ADVANCED ALGORITHMS

NOTES FOR CMU 15-850 (FALL 2020)

LECTURER: ANUPAM GUPTA

About this document

This document contains the course notes for 15-850: Advanced Algorithms, a graduate-level course taught by Anupam Gupta at Carnegie Mellon University in Fall 2020. Parts of these notes were written by the students of previous versions of the class (based on the lectures) and then edited by the professor. The names of the student scribes will appear soon, as will missing details and bug fixes, more chapters, exercises, and (better) figures.

The notes have not been thoroughly checked for accuracy, especially attributions of results. They are intended to serve as study resources and not as a substitute for professionally prepared publications. We apologize for any inadvertent inaccuracies or misrepresentations.

More information about the course, including problem sets and references, can be found on the course website:

`https://www.cs.cmu.edu/~15850/`

The style files (as well as the text on this page!) are mildly adapted from the ones developed by Yufei Zhao (MIT), for his notes on Graph Theory and Additive Combinatorics. As some of you may guess, the LaTeX template used for these notes is called `tufte-book`.

Contents

Part I

Discrete Algorithms

1
Minimum Spanning Trees

1.1 Minimum Spanning Trees: History

In minimum spanning tree problem, the input is an undirected connected graph $G = (V, E)$ with n nodes and m edges, where the edges have weights $w(e) \in \mathbb{R}$. The goal is to find a spanning tree of the graph with the minimum total edge-weight. If the graph G is disconnected, we get a *spanning forest* As a classic (and important) problem, it's been tackled many times. Here's a brief, not-quite-comprehensive history of its optimization, all without making any assumptions on the edge weights other that they can be compared in constant time:

A spanning tree/forest is defined to be an acyclic subgraph T that is inclusion-wise maximal, i.e., adding any edge in $G \setminus T$ would create a cycle.

- Otakar Borůvka [1] gave the first known MST algorithm in 1926; it was independently discovered by Gustave Choquet, Georges Sollin, and others. Vojtěch Jarník [2] gave his algorithm in 1930, and it was independently discovered by Robert Prim ('57) and Edsger Dijkstra ('59), among others. Joseph Kruskal gave his algorithm in '56; this was rediscovered by Loberman and Weinberger in 57. All these can easily be implemented in $O(m \log n)$ time; we will discuss these in this lecture.

[1]

[2]

J.B. Kruskal, Jr. (1956)

Loberman and Weinberger (1957)

Both Prim and Kruskal refer to Borůvka's paper, but say it is "unnecesarily elaborate". However, while Borůvka's paper *is* written in a complicated fashion, but his essential ideas are very clean.

- In 1975, Andy Yao [3] achieved a runtime of $O(m \log \log n)$. His algorithm builds on Borůvka's algorithm (which he attributes to Sollin), and uses as a subroutine the linear-time algorithm for median-finding, which had only recently been invented in 1974. We will work through Yao's algorithm in HW#1.

[3]

- In 1984, Michael Fredman and Bob Tarjan gave an $O(m \log^* n)$ time algorithm, based on their Fibonacci heaps data structure. Here $\log^*$ is the *iterated logarithm* function, and denotes the number of times we must take logarithms before the argument becomes smaller than 1. The actual runtime is a bit more nuanced, which we will not bother with today.

Fredman and Tarjan (1987)

This result was soon improved by Gabow, Galil, Spencer, and Tarjan ('86) to get an $O(m \log \log^* n)$ runtime—note the logarithm *applied to* the iterated logarithm.

Gabow, Galil, Spencer, and Tarjan (1986)

- In 1995, David Karger, Phil Klein and Bob Tarjan finally got the holy grail of $O(m)$ time! ... but it was a randomized algorithm, so the search for a deterministic linear-time algorithm continued.

Karger, Klein, and Tarjan (1995)

- In 1997, Bernard Chazelle gave an $O(m\alpha(n))$-time deterministic algorithm. Here $\alpha(n)$ is the inverse Ackermann function (defined in §1.6). This function grows extremely slowly, even slower than the iterated logarithm function. However, it still goes to infinity as $n \to \infty$, so we still don't have a deterministic linear-time MST algorithm.

Chazelle (1997)

- In 1998, Seth Pettie and Vijaya Ramachandran gave an *optimal* algorithm for computing minimum spanning trees—however, we don't know its runtime! More formally, they show that if there exists an algorithm which uses $MST^*(m, n)$ comparisons to find MSTs on all graphs with m edges and n nodes, the Pettie-Ramachandran algorithm will run in time $O(MST^*(m, n))$.)

Pettie and Ramachandran (1998)

This was part of Seth's Ph.D. thesis, and Vijaya was his advisor.

In this chapter, we'll go through the three classics (Jarnik/Prim's, Kruskal's, and Borůvka's). Then we will discuss Fredman and Tarjan's algorithm, and finally present Karger, Klein, and Tarjan's randomized algorithm. This will lead us to discuss another intriguing question: ***how do we verify whether a given tree is an MST?***

For the rest of this chapter, assume that the edge weights are *distinct*. This does not change things in any essential way, but it ensures that the MST is unique (Exercise: prove this!), and hence simplifies some statements. Also assume the graph is simple, and hence $m = O(n^2)$; you can delete all self-loops and remove all-but-the-lightest from any collection of parallel edges, all by preprocessing the graph in linear time.

1.1.1 *The Cut and Cycle Rules*

Most of these algorithms rely on two rules: the ***cut rule*** (known in Tarjan's book as the blue rule) and the ***cycle rule*** (or the red rule). Recall that a *cut* in the graph is a partition of the vertices into two non-empty sets $(S, \bar{S} = V \setminus S)$, and an edge *crosses* this cut if its two endpoints lie in different sets.

Tarjan (1983)

Theorem 1.1 (Cut Rule). *For any cut of the graph, the minimum-weight edge that crosses the cut must be in the MST. This rule helps us determine what to add to our MST.*

Proof. Let $S \subsetneq V$ be any nonempty proper subset of vertices, let $e = \{u, v\}$ be the minimum-weight edge that crosses the cut defined by $(S, \bar{S})$ (W.l.o.g., $u \in S$, $v \notin S$), and let T be a spanning tree not containing e. Then $T \cup \{e\}$ contains a unique cycle C. Since C crosses the cut $(S, \bar{S})$ once (namely at e), it must also cross at another edge e'. But $w(e') > w(e)$, so $T' = (T - \{e'\}) \cup \{e\}$ is a lower-weight tree than T, so T is not the MST. Since T was an arbitrary spanning tree not containing e, the MST must contain e. $\square$

Theorem 1.2 (Cycle Rule). *For any cycle in G, the heaviest edge on that cycle cannot be in the MST. This helps us determine what we can remove in constructing the MST.*

Proof. Let C be any cycle, let e be the heaviest edge in C. For a contradiction, let T be an MST that contains e. Dropping e from T gives two components. Now there must be some edge e' in $C \setminus \{e\}$ that crosses between these two components, and hence $T' := (T - \{e'\}) \cup \{e\}$ is a spanning tree. (Make sure you see why.) By the choice of e we have $w(e') < w(e)$, so T' is a lower-weight spanning tree than T, a contradiction. $\square$

To find a minimum spanning tree, we repeated apply whichever of these rules we like. E.g., we choose some cut, use the cut rule to designate the lightest edge in it as belonging to the MST by coloring it blue (hence the name). [4] Or we choose a cycle which contains no red edge, use the cycle rule to mark the heaviest edge as not being in the MST, and color it red. (Again, this edge cannot already be blue for similar reasons.) And if either of the rules is not applicable, we are done. Indeed, if we cannot apply the blue rule, the blue edges cross every cut, and hence form a spanning tree, which must be the MST. Similarly, once the non-red edges do not contain a cycle, they form a spanning tree, which must be the MST. All known algorithms differ only in their choice of cut/cycle, and how they find these fast. Indeed, all the deterministic algorithms we discuss today will just use the cut rule, whereas the randomized algorithm will use the cycle rule as well.

[4] This edge e cannot have previously been colored red—this follows from the above lemmas. Or more directly, any cycle crosses any cut an even number of times, so a cycle containing e also contains another edge f in the cut, which is heavier.

1.2 The Classical Algorithms

1.2.1 Kruskal's Algorithm

For Kruskal's Algorithm, first sort all the edges such that $w(e_1) < w(e_2) < \cdots < w(e_m)$. This takes $O(m \log m) = O(m \log n)$ time. Start with all edges being uncolored, and iterate through the edges in the sorted order, coloring an edge blue if and only if it connects two vertices which are not currently in the same blue component. Figure 1.1 gives an example of how edges are added.

To keep track of which vertex is in which component, use a ***disjoint set union-find*** data structure. This data structure has three operations:

- `makeset`$(elem)$, which takes an element *elem* and creates a new singleton set for it,
- `find`$(elem)$, which finds the canonical representative for the set containing the element *elem*, and
- `union`$(elem_1, elem_2)$, which merges the two sets that $elem_1$ and $elem_2$ are in.

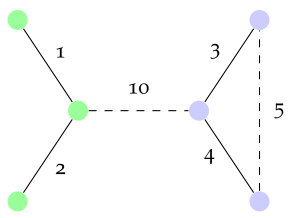


Figure 1.1: Dashed lines are not yet in the MST. Note that 5 will be analyzed next, but will not be added. 10 will be added. Colors designate connected components.

There is an implementation of this data structure which allows us to do m operations in $O(m\,\alpha(m))$ ***amortized*** time, where $\alpha(\cdot)$ is the inverse Ackermann function mentioned above. Note that the naïve implementation of Kruskal's algorithm spends $O(m \log m) = O(m \log n)$ time to sort the edges, and then performs n `makeset`s, m `find`s, and $n-1$ `union` operations, the total runtime is $O(m \log n + m\,\alpha(m))$, which is dominated by the $O(m \log n)$ term.

1.2.2 *The Jarnik/Prim Algorithm*

For the Jarnik/Prim algorithm, first take an arbitrary root vertex r to start our MST T. At each iteration, take the cheapest edge connecting of our current tree T of blue edges to some vertex not yet in T, and color it blue—thereby adding this edge to T and increasing its size by one. Figure 1.2 below shows an example of how we edges are added.

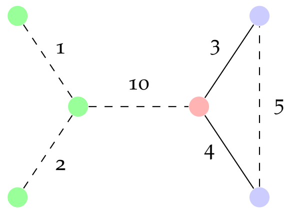


Figure 1.2: Dashed lines are not yet in the MST. We started at the red node, and the blue nodes are also part of T right now.

We'll use a ***priority queue*** data structure which keeps track of the lightest edge connecting T to each vertex not yet in T. A priority queue data structure is equipped with (at least) three operations:

- `insert`$(elem, key)$ inserts the given $(element, key)$ pair into the queue,
- `decreasekey`$(elem, newkey)$ changes the key of the element *elem* from its current key to $\min(originalkey, newkey)$, and
- `extractmin`$()$ removes the element with the minimum key from the priority queue, and returns the $(elem, key)$ pair.

Note that by using the standard ***binary heap*** data structure we can get $O(\log n)$ worst-case time for each priority queue operation above.

To implement the Jarnik/Prim algorithm, we initially insert each vertex in $V \setminus \{r\}$ into the priority queue with key ∞, and the root r with key 0. The key of an node v denotes the weight of the least-weight edge from a node in T to v; it is zero if $v \in T$, and ∞ if there are no edges yet from nodes in T to v. At each step, use `extractmin` to find the vertex u with smallest key, and add u to the tree using this edge. Then for each neighbor of u, say v, do `decreasekey`$(v, w(\{u, v\}))$. Overall we do m `decreasekey` operations, n `insert`s, and n `extractmin`s, with the `decreasekey`s supplying the dominating $O(m \log n)$ term.

We can optimize slightly by inserting a vertex into the priority queue only when it has an edge to the current tree T. This does not seem particularly useful right now, but will be crucial in the Fredman-Tarjan proof.

1.2.3 *Borůvka's Algorithm*

Unlike Kruskal's and Jarnik/Prim's algorithms, Borůvka's algorithm adds many edges in parallel, and can be implemented without any non-trivial data structures. In a "round", simply take the lightest edge out of each vertex and color it blue; these edges are guaranteed to form a forest if edge-weights are distinct. (Exercise: why?)

Now contract the blue edges and recurse on the resulting graph. At the end, when the resulting graph is a single vertex, uncontract all the edges to get the MST. Each round can be implemented in $O(m)$ work: we will work out the details of this in HW #1. Moreover, we're guaranteed to shrink away at least half of the nodes (as each node at least pairs up with one other node), and maybe many more if we are lucky. So we have at most $\lceil \log_2 n \rceil$ rounds of computation, leaving us with $O(m \log n)$ total work.

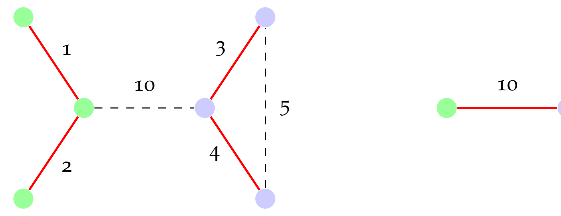


Figure 1.3: The red edges will be chosen and contracted in a single step, yielding the graph on the right, which we recurse on. Colors designate components.

1.2.4 *A Slight Improvement on Jarnik/Prim*

We can actually easily improve the performance of Jarnik/Prim's algorithm by using a more sophisticated data structure, namely by using ***Fibonacci heaps*** instead of binary heaps to implement the priority queue. Fibonacci heaps (invented by Fredman and Tarjan) implement the `insert` and `decreasekey` operations in constant amortized time, and `extractmin` in amortized $O(\log H)$ time, where H is the maximum number of elements in the heap during the execution. Since we do n `extractmin`s, and $O(m + n)$ of the other two operations, and the maximum size of the heap is at most n, this gives us a total cost of $O(m + n \log n)$.

Note that this is linear time on graphs with $m = \Omega(n \log n)$ edges; however, we'd like to get linear-time on all graphs. So the remaining cases are the graphs with $m = o(n \log n)$ edges.

1.3 *Fredman and Tarjan's $O(m \log^* n)$-time Algorithm*

Fredman and Tarjan's algorithm builds on Jarnik/Prim's algorithm: the crucial observation uses the following crucial facts.

The amortized cost of `extractmin` operations in Fibonacci heaps is $O(\log H)$, where H is the maximum size of the heap. Moreover, in Jarnik/Prim's algorithm, the size of the heap is just the number of nodes that are adjacent to the current tree T. So if the current tree always has a "small boundary", the `extractmin` cost will be low.

How can we maintain the boundary to be smaller than some threshold K? Simple: Once the boundary exceeds K, stop growing the Prim tree, and begin Jarnik/Prim's algorithm anew from a different vertex. Do this until we have a forest where all vertices lie in some tree; then contract these trees (much like Borůvka), and recurse on the smaller graph. Before we formally define the algorithm, here's an example.

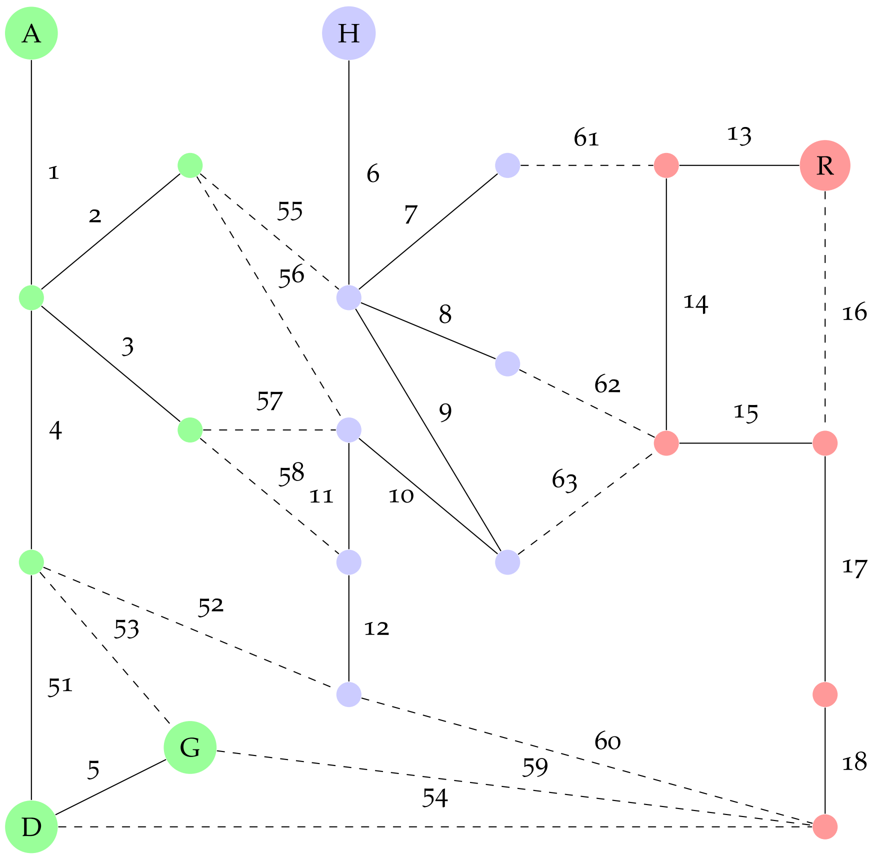


Figure 1.4: We begin at vertices A, H, R, and D (in that order) with $K = 6$. Although D begins as its own component, it stops when it joins with tree A. Dashed edges are not chosen in this step (though they may be chosen in the next recursive call), and colors denote trees.

Formally, in each round of the algorithm, all vertices start as unmarked.

1. Pick an arbitrary unmarked vertex and start Jarnik/Prim's algorithm from it, creating a tree T. Keep track of the lightest edge from T to each vertex in the neighborhood $N(T)$ of T, where $N(T) := \{v \in V - T \mid \exists u \in T \text{ s.t. } \{u, v\} \in E\}$. Note that $N(T)$ may contain vertices that are marked.

2. If at any time $|N(T)| \geq K$, or if T has just added an edge to some vertex that was previously marked, stop and mark all vertices in the current T, and go to step 1.

3. Terminate when each node belongs to some tree.

Let's first note that the runtime of one round of the algorithm is $O(m + n \log K)$. Each edge is considered at most twice, once from each endpoint, giving us the $O(m)$ term. Each time we grow the current tree in step 1, the number of connected components decreases by 1, so there are at most n such steps. Each step calls `findmin` on a heap of size at most K, which takes $O(\log K)$ times. Hence, at the end of this round, we've successfully identified a forest, each edge of which is part of the final MST, in $O(m + n \log K)$ time.

Let d_v be the degree of the vertex v in the graph we consider in this round. We claim that every marked vertex u belongs to a component C such that $\sum_{v \in C} d_v \geq K$. Indeed, if u became marked because the neighborhood of its component had size at least K, then this is true. Otherwise, u became marked because it entered a component C of marked vertices. Since the vertices of C were marked, $\sum_{v \in C} d_v \geq K$ before u joined, and this sum only increased when u (and other vertices) joined. Thus, if $C_1, \ldots, C_l$ are the components at the end of this routine, we have

$$2m = \sum_v d_v = \sum_{i=1}^{l} \sum_{v \in C_i} d_v \geq \sum_{i=1}^{l} K \geq Kl$$

Thus $l \leq \frac{2m}{K}$, i.e. this routine produced at most $\frac{2m}{K}$ trees.

The choice of K will change over the course of the algorithm. How should we set the thresholds K_i? Say we start round i with n_i nodes and $m_i \leq m$ edges. One clean way is to set

$$K_i := 2^{\frac{2m}{n_i}}$$

which ensures that

$$O(m_i + n_i \log K_i) = O\left(m_i + n_i \cdot \frac{2m}{n_i}\right) = O(m).$$

In turn, this means the number of trees, and hence the number of nodes n_{i+1} in the next round, is at most $\frac{2m_i}{K_i} \leq \frac{2m}{K_i}$. The number of edges is $m_{i+1} \leq m_i \leq m$. Rewriting, this gives

$$K_i \leq \frac{2m}{n_{i+1}} = \lg K_{i+1} \implies K_{i+1} \geq 2^{K_i}.$$

Hence the threshold value exponentiates in each step. Hence after $\log^* n$ rounds, the value of K would be at least n, and we would

The threshold increases "tetrationally".

just run Jarnik/Prim's algorithm to completion, ending with a single tree. This means we have at most $\log^* n$ rounds, and a total of $O(m \log^* n)$ work.

In retrospect, I don't know whether to consider the Fredman-Tarjan algorithm as being trivial (once we have Fibonacci heaps) or being devilishly clever. I think it is the latter (and that is the beauty of the best algorithms). Indeed, there's a lovely idea—of keeping the neighborhoods small at the beginning when there's a lot of work to do, but allow them to grow quickly, as the graph collapses. It is quite non-obvious at the start, and obvious in hindsight. And once you see it, you cannot un-see it!

1.4 A Linear-Time Randomized Algorithm

Another algorithm that is extremely clever but almost obvious in hindsight is the the Karger-Klein-Tarjan randomized MST algorithm, which runs in $O(m + n)$ expected time. The new idea here is to compute a "rough approximation" to the MST, use that to throw away many edges using the *cycle rule*, and then recurse on the rest of the graph.

Karger, Klein, and Tarjan (1995)

A version of this algorithm was proposed by Karger in 1992, but he only obtained an $O(m + n \log n)$ runtime. The enhancement to linear time was given by Klein and Tarjan at the STOC 1994 conference; the combined paper is cited above.

1.4.1 Heavy & light edges

The crucial definition is that of edges being *heavy* and *light* with respect to some forest F.

Definition 1.3. Let F be a forest that is a subgraph of G. An edge $e \in E(G)$ is ***F*-heavy** if e creates a cycle when added to F, and moreover it is the heaviest edge in this cycle. Otherwise, we say edge e is ***F*-light**.

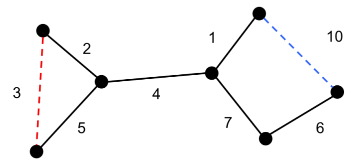


Figure 1.5: Fix this figure, make it interesting. Every edge in F is F-light, as are the edges on the left, and also those going between the components. The edge on the right is F-heavy.

The next facts follow from the definition:

Fact 1.4. Edge e is F-light $\iff$ $e \in \text{MST}(F \cup \{e\})$.

Fact 1.5 (Completeness). If T is an MST of G then edge $e \in E(G)$ is T-light if and only if $e \in T$.

Fact 1.6 (Soundness). For any forest F, the F-light edges contain the MST of the underlying graph G. In other words, any F-heavy edge is also heavy with respect to the MST of the entire graph.

This suggests a clear strategy: pick a forest F from the current edges, and discard all the F-heavy edges. Hopefully the number of edges remaining is small. By Fact 1.6 these edges contain the MST of G, so repeat the process on them. To make this idea work, we want a forest F with many F-heavy edges. The catch is that a forest has many heavy edges if it has small weight, if there are many off-forest edges forming cycles where they are the heaviest edges. Indeed, one such forest in the MST T^* of G: Fact 1.5 shows there are $m-(n-1)$ many T^*-heavy edges, the maximum possible. How do we find some similarly good tree/forest, but in linear time?

A second issue is to classify edges as light/heavy, given a forest F. It is easy to classify a single edge e in linear time, but the following remarkable theorem is also true:

Theorem 1.7 (MST Verification). *Given a forest $F \subseteq G$, we can output the set of all F-light edges in G in time $O(m+n)$.*

This MST verification algorithm itself uses several interesting ideas; we discuss some of them in Section 1.5. But for now, let us use it to give the randomized linear-time MST algorithm.

1.4.2 *The Randomized MST Algorithm*

The idea is simple and elegant: randomly choose half of the edges and find the minimum-weight spanning forest F on this "half-of-a-graph". This forest F should have many F-heavy edges; we discard these and recursively find the MST on the remaining graph. Since both the recursive calls are on smaller graphs, hopefully the runtime will be linear.

The random subgraph may not be connected, so the maximum spanning forest is obtained by finding the MST for each of its connected components.

The actual algorithm below has just one extra step: we first run a few rounds of Borůvka's algorithm to force a reduction in the number of vertices, and then do the steps above.

Algorithm 1: KKT(G)

```
1.1 Run 3 rounds of Borůvka's Algorithm on $G$, contracting the
      chosen edges to get a graph $G' = (V', E')$ with $n' \leq n/8$
      vertices and $m' \leq m$ edges.
1.2 If $G'$ has a single vertex, return any chosen edges.
1.3 $E_1 \leftarrow$ random sample of $E'$, each edge picked indep. w.p. $1/2$.
1.4 $F_1 \leftarrow$ KKT($G_1 = (V', E_1)$).
1.5 $E_2 \leftarrow$ all the $F_1$-light edges in $E'$.
1.6 $F_2 \leftarrow$ KKT($G_2 = (V', E_2)$).
1.7 return $F_2$ (combined with Borůvka edges chosen in Step 1).
```

Theorem 1.8. *The KKT algorithm returns MST(G).*

Proof. This follows from Fact 1.6, that discarding heavy edges of any forest F in a graph does not change the MST. Indeed, the MST on G_2 is the same as the MST on G', since the discarded F_1-heavy edges cannot be in $MST(G')$ because of Fact 1.6. Adding back the edges picked by Borůvka's algorithm in Step 1 gives the MST on G, by the cut rule. □

Now we need to bound the running time. The following two claims formalize the intuition that we recurse on "smaller" subgraphs:

Claim 1.9. $\mathbb{E}[\#E_1] = \frac{1}{2}m'$.

Claim 1.10. $\mathbb{E}[\#E_2] \leq 2n'$.

The first claim is easy to prove, using linearity of expectations, and that each edge is picked with probability 1/2. The proof of Claim 1.10 is also short, but before we prove it, let us complete the proof of the linear running time.

Theorem 1.11. *The KKT algorithm, run on a graph with m edges and n vertices, terminates in expected time $O(m+n)$.*

Proof. Let T_G be the expected running time on graph G, and

$$T_{m,n} := \max_{G=(V,E),|V|=n,|E|=m} \{T_G\}.$$

In the KKT algorithm, Step 1, 2, 4 and 6 can each be done in linear time: indeed, the only non-trivial part is Step 4, for which we use Theorem 1.7. Let the total time for these steps be at most cm. Steps 3 and 5 requires time T_{G_1} and T_{G_2} respectively. Then we have

$$T_G \leq cm + \mathbb{E}[T_{G_1} + T_{G_2}] \leq cm + \mathbb{E}[T_{m_1,n'} + T_{m_2,n'}],$$

where $m_1 = \#E_1$ and $m_2 = \#E_2$ are both random variables. Inductively assume that $T_{m,n} \leq c(2m+n)$, then

$$\begin{aligned} T_G &\leq cm + \mathbb{E}[c(2m_1 + n')] + \mathbb{E}[c(2m_2 + n')] \\ &\leq c(m + m' + 6n') \\ &\leq c(2m+n) \end{aligned}$$

The second inequality holds because $\mathbb{E}[m_1] \leq \frac{1}{2}m'$ and $\mathbb{E}[m_2] \leq 2n'$. The last inequality holds because $n' \leq n/8$ and $m' \leq m$. Indeed, we shrunk the graph using Borůvka's algorithm in the first step just to ensure $n' \leq 8n$ and hence give us some "breathing room". ☐

Now we prove Claim 1.10. Recall that we randomly subsample the edges of G' to get G_1, compute its maximum spanning forest F_1, and now we want to bound the expected number of edges in G' that are F_1-light. The key to the proof is to do all these steps together, deferring the random decisions to when we really need them. This makes it apparent which edges are light, making them easy to count.

This idea to defer looking at the random choices of the algorithm is often called the ***principle of deferred decisions***.

Proof of Claim 1.10. For the sake of the proof, we can use any correct algorithm to compute F_1, so let us use Kruskal's algorithm. Moreover, let's run a *lazy* version as follows: first sort all the edges in E', and not just those in $E_1 \subseteq E'$, and consider then in increasing order of weights. Now if the currently considered edge e_i connects two different trees in the current blue forest, call e_i *useful* and flip an independent unbiased coin: if the coin comes up "heads", color e_i blue and add it to F_1, else color e_i red. The crucial observation is that this process produces a forest from the same distribution as first choosing G_1 and then computing F_1 by running Kruskal's algorithm on it.

Now, let us consider the lazy process again: which edges are F_1-light? We claim that these are precisely the useful edges. Indeed, any non-useful edge e_j forms a cycle with the previously chosen blue edges in F_1, and it is the heaviest edge on that cycle. Hence e_j does not belong to $MST(F_1 \cup \{e_j\})$, so it is F_1-heavy by Fact 1.4. And a useful edge e_i would belong to $MST(F_1 \cup \{e_i\})$, since running Kruskal's algorithm on $F_1 \cup \{e_i\}$ would see that e_i connects two different blue components and hence would pick it.

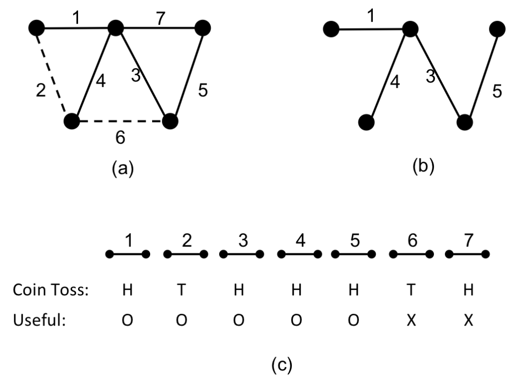


Figure 1.6: Illustration of another order of coin tossing

Finally, how many useful edges are there, in expectation? Let's abstract away the details: we're running a process that periodically asks us to flip an independent unbiased coin. Since each time we see a heads, we add an edge to the forest, so we definitely stop when we see $n' - 1$ heads. (We may stop earlier, in case the process runs out of edges, but then we can pad the random sequence to flip some more coins.) Since the coins are independent and unbiased, the expected number of flips until we see $n' - 1$ heads is exactly $2(n' - 1)$. This proves Claim 1.10. $\square$

That's it. The algorithm and proof are both short and slick and beautiful: this result is a real gem. I think it's an algorithm from ***The Book***. The one slight annoyance with the algorithm is the relative complexity of the MST verification algorithm, which we use to find the F_1-light edges in linear time. Nonetheless, these verification algorithms also contain many nice ideas, which we now discuss.

Paul Erdős claimed that God has "The Book" which contains the most elegant proof of each mathematical theorem.

The current verification algorithms are deterministic; can we use randomness to simplify these as well?

1.5 *Optional: MST Verification*

We now come back to the implementation of the MST verification procedure. Here we only consider only trees (not forests), since we can run this algorithm separately on each tree in the forest and incur only a linear extra cost. Let us refine Theorem 1.7 as follows.

Theorem 1.12 (MST Verification)**.** *Given a tree $T = (V, E)$ where $|V| = n$, and m pairs of vertices (y_i, z_i) in T, we can find the heaviest edge on the unique y_i-to-z_i path in T for all i, in $O(m + n)$ time.*

Since the edge $\{y_i, z_i\}$ is T-heavy precisely if it is heavier than the heaviest edge on the corresponding tree path, this also proves Theorem 1.7. Observe that the query pairs are given up-front: there is an inverse-Ackermann-type lower bound for the problem where the queries arrive online.

Pettie (2006)

How do we get such a linear-time algorithm? *A priori*, it is not easy to even show a ***query-complexity upper bound***: that there exists a procedure that performs *a linear number of edge-weight comparisons* to solve the MST verification problem. This problem was solved by János Komlós. His result was subsequently made algorithmic ("how do you find (in linear time) which linear number of queries to make?") by Brendan Dixon, Monika Rauch (now Monika Henzinger) and Bob Tarjan. This algorithm was futher simplified by Valerie King [5], and by Thomas Hagerup [6]. We will just discuss Komlós's query-complexity bound.

Komlos (1985)

[5]
[6]

1.5.1 A Simpler Case

To start developing the algortihm, it helps to consider special cases: e.g., what if the tree is a complete binary tree? Let's assume something slightly less restrictive than a complete binary tree: suppose tree T is rooted at some node r, all internal nodes have at least 2 children, and all its leaves are at the same level. Moreover, all queries $\{y_i, z_i\}$ are for pairs where y_i is a leaf and z_i its ancestor.

A node v is an ***ancestor*** of u if v lies on the unique path from u to the root; then u is a ***descendent*** of v.

Now for an edge (u, v) of the tree, where v is the parent and u the child, consider all queries starting within subtree T_u and ending at vertex v or higher. Say these queries go from some leaves inside T_v up to $w_1, w_2, \ldots, w_k$, where w_1 is closest to the root. Define the "query string"

$$Q_e := (w_1, w_2, \ldots, w_k).$$

We want to calculate the "answer string"

$$A_e := (a_1, a_2, \cdots, a_k),$$

where a_i is the largest weight among the edges between w_i and u.

Now given the answer string $A_{(b,a)}$, we can get the answer string for a child edge. In the example, say the query string for edge (c, b) is $Q_{(c,b)} = (w_1, w_4, b)$. We have lost some queries that were in $Q_{(b,a)}$, (e.g., for w_3) but we now have a query ending at b. To get $A_{(b,a)}$ we can drop the lost queries, add in the entry for b, and also take the component-wise maximum with the weight of (c, b) itself. E.g., if (c, b) has weight t, then

$$A_{(c,b)} = (\max\{a_1, t\}, \max\{a_4, t\}, t) = (\max\{6, 5\}, \max\{4, 5\}, 5).$$

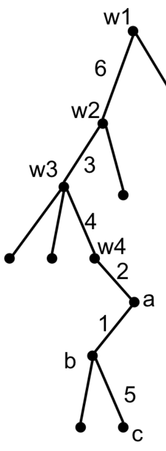


Figure 1.7: Query string $Q_{(b,a)} = (w_1, w_3, w_4)$ means there are three queries starting from vertices in T_b and ending at w_1, w_3, w_4. The answer string is $A_{(b,a)} = (a_1, a_3, a_4) = (6, 4, 4)$.

Naïvely this would require us to compare the weight $w_{(c,b)}$ with all the entries in the answer string, incurring $|A_{e'}|$ comparisons. The crucial observation is this: since the nodes in the query string are sorted from top to bottom, the answers must be non-increasing: i.e., $a_1 \geq a_2 \geq \cdots \geq a_k$. Therefore we can do binary search to reduce the number of comparisons between edge-weights. Indeed, given the answer string for some edge e, we can compute answers $A_{e'}$ for a child edge e' using at most $\lceil \log(|A_{e'}| + 1) \rceil$ comparisons. This will be enough to prove the result.

Claim 1.13. The total number of comparisons for all queries is at most

$$\sum_e \log\left(|Q_e| + 1\right) \leq O(n + n \log \frac{m+n}{n}) = O(m+n).$$

Proof. Let the number of edges at height i be n_i, where height 1 corresponds to edges incident to the leaves.

$$\begin{aligned}\sum_{e \in \text{height } i} \log_2(1 + |Q_e|) &= n_i \operatorname{avg}_{e \in \text{height } i}(\log_2(1 + |Q_e|)) \\ &\leq n_i \log_2 \left(1 + \operatorname{avg}_{e \in \text{height } i}(|Q_e|)\right) \\ &\leq n_i \log_2 \left(1 + \frac{m}{n_i}\right) \\ &= n_i \left(\log_2 \frac{m+n}{4n} + \log_2 \frac{4n}{n_i}\right).\end{aligned}$$

The first inequality uses concavity of the function $\log_2(1+x)$, and Jensen's inequality. The second holds because each of the m queries can only appear on at most one edge, so the average "load" is at most m/n_i. Summing the first term over all heights gives $n \log_2 \frac{m+n}{4n} = O(m)$.

Jensen's inequality says that for any convex function f and any random variable X, $\mathbb{E}[f(X)] \geq f(\mathbb{E}[X])$. Concavity requires flipping the sign, of course.

To bound the second term (summed over all heights), recall that each node has at least two children, so the number of edges at least doubles each time the height decreases. Hence, $n_i \leq n/2^{i-1}$, and

$$\sum_{i \geq 1} n_i \log_2 \frac{4n}{n_i} \leq \sum_{i \geq 1} \frac{n}{2^{i-1}} \log_2 \frac{4n}{n/2^{i-1}} = n \cdot \sum_{i \geq 1} \frac{O(i)}{2^i} = O(n).$$

$$\begin{aligned} S &= \sum_{i \geq 0} \frac{i}{2^i} \\ 2S &= \sum_{i \geq 0} \frac{i}{2^{i-1}} = \sum_{i \geq 0} \frac{i+1}{2^i} \\ \implies 2S - S &= \sum_{i \geq 0} \frac{(i+1) - i}{2^i} = \sum_{i \geq 0} \frac{1}{2^i} = 2. \end{aligned}$$

The inequality above uses that $x \log(4n/x)$ is increasing for $x \leq n$. ☐

Converting this into an algorithm that runs in $O(m+n)$ time requires quite a bit more work. The essential idea is to store each query string $Q_{(u,v)}$ as a bit vector of length $\log_2 n$, indicating which nodes on the path from v to the root belong to it $Q_{(u,v)}$. Now the answers $A_{(u,v)}$ can be stored by encoding the locations of the successive maxima. And answers for a child edge can be computed from that of the parent edge using some tricky bit operations (e.g., by precomputing solutions on bit-strings of length, say $(\log_2 n)/3$, of which there are only $n^{1/3} \times n^{1/3} = n^{2/3}$). If you are interested, check out these lecture slides by Uri Zwick.

1.5.2 *Solving the General Case*

Finally, we reduce a general instance of MST verification to the special instances considered in §1.5.2. First we reduce to a "branching" tree with the special properties we asked for, then we alter the queries to become leaf-ancestor queries.

To achieve this reduction, run Borůvka's algorithm on the tree T. After the i^{th} round of edge selection and contraction, let V_i be the remaining vertices, so that $V_0 = V$ is the original set of nodes. Define a new tree T' whose vertex set V' is the disjoint union $V_0 \uplus V_1 \uplus \cdots$. A node $u \in V_i$ has an edge in T' to $v \in V_{i+1}$ if the component containing u was contracted into the new vertex v; the weight of this edge in T' is the weight of the minimum-weight edge chosen by u in this round. Moreover, if r is the single vertex corresponding to the entire tree T at the end of the run of Borůvka's algorithm, then root tree T' at r.

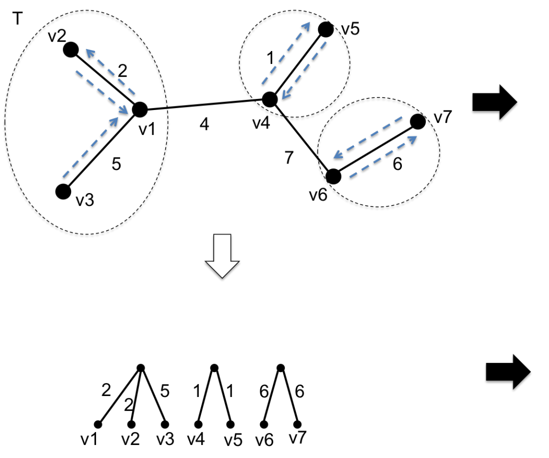


Figure 1.8: Illustration of balancing a tree. We have $\texttt{maxwt}_T(v_1, v_7)$ is 7 which is the weight of edge (v_4, v_6). We can check that $\texttt{maxwt}_{T'}(v_1, v_7)$ is also 7.

Exercise 1.14. Show that each node in T' has at least two children, and all leaves belong to the same level. There are n leaves (corresponding to the nodes in T), and at most $2n - 1$ nodes in T'. Also show how to construct T' in linear time.

Exercise 1.15. For nodes u, v in a tree T, let $\texttt{maxwt}_T(u, v)$ be the maximum weight of an edge on the (unique) path between u, v in the tree T. Show that all $u, v \in V$, $\texttt{maxwt}_T(u, v) = \texttt{maxwt}_{T'}(u, v)$.

This exercise means arbitrary queries (y_i, z_i) in the original tree T can be reduced to leaf-leaf queries in T'. To make these leaf-ancestor queries, we simply find the least-common ancestor $\ell_i := \text{lca}(y_i, z_i)$ for each pair, and replace the original query by the maximum of two queries $(y_i, \ell_i), (z_i, \ell_i)$. To show that we can find the least-common ancestors in linear time, we defer to a theorem of David Harel and Bob Tarjan:

Harel and Tarjan (1984)

Theorem 1.16. *Given a tree T, we can preprocess it in $O(n)$ time, so that all subsequent least-common ancestor queries for T can be answered in $O(1)$ time.*

Interestingly, this algorithm also proceeds by solving the least-common ancestor problem for complete balanced binary trees, and then extending the solution to general trees. For a survey of algorithms for this problem, see the paper of Alstrup et al.

Alstrup et al. (2004)

This completes Komlós' proof that the MST verification problem can be solved using $O(m + n)$ comparisons. An outstanding open problem is to get a really simple linear-time algorithm for this problem. (An algorithm that runs in time $O(m\alpha(n))$ can be given using the disjoint set union-find data structure.)

1.6 *The Ackermann Function*

Wilhelm Ackermann defined a fast-growing function that is totally computable but not primitive recursive. Today, we use the term ***Ackermann function*** $A(m,n)$ to refer to one of many variants that are rapidly-growing and have similar propeties. It seems to arise often in algorithm analysis, so let's briefly discuss it here.

Ackermann (1928)

A similar function was defined by Gabriel Sudan, a Romanian mathematician, in 1927.

For illustrative purposes, it is cleanest to define $A(m,n) : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ recursively as

$$A(m,n) = \begin{cases} 2n & : \quad m = 1 \\ 2 & : \quad m \geq 1, xn = 1 \\ A(m-1, A(m,n-1)) & : \quad m \geq 2, n \geq 2 \end{cases}$$

Here are the values of $A(m,n)$ for $m,n \leq 4$:

	1	2	3	4	...	n
1	2	4	6	8	...	$2n$
2	2	4	8	16	...	2^n
3	2	4	2^{2^2}	$2^{2^{2^2}}$	...	$2^{2^{\cdot^{\cdot^{2}}}}$
4	2	4	65536	!!!	...	huge!

We can define the ***inverse Ackermann function*** $\alpha(\cdot)$ to be a functional inverse of the diagonal $A(n,n)$; by construction, $\alpha(\cdot)$ grows extremely slowly. For example, $\alpha(m) \leq 4$ for all $m \leq 2^{2^{\cdot^{\cdot^{2}}}}$ where the tower has height 65536.

1.7 *Matroids*

To come here. See HW1 as well.

2
Arborescences: Directed Spanning Trees

Greedy algorithms worked vey well for minimum weight spanning tree problem, as we saw in Chapter 1. In this chapter, we define arborescences which are a notion of spanning trees for rooted directed graphs. We will see that a naïve greedy approach no longer works, but it requires just a slightly more sophisticated algorithm to efficiently find them. We give two proofs of correctness for this algorithm. The first is a direct inductive proof, but the second makes use of linear programming duality, and highlights its use in analyzing the performance of algorithms. This will be a theme that return to multiple times in this course.

2.1 *Arborescences*

Consider a graph $G = (V, A, w)$: here V is a set of vertices, and A a set of directed edges, also known as ***arcs***. The function $w : A \to \mathbb{R}$ gives a weight to every arc. Let $|V| = n$ and $|A| = m$. Once we root G at a node $r \in V$, we can define a "directed spanning tree" with r being the *sink/root*.

We will use "arcs" instead of "edges" to emphasize the directedness of the graph.

Definition 2.1. An ***r-arborescence*** is a subgraph $T = (V, A')$ with $A' \subseteq A$ such that

1. Every vertex has a directed path in T to the root r, and
2. Each vertex except r has one outgoing arc; r has none.

A ***branching*** is the directed analog of a forest; it drops the first reachability requirement, and asks only for all non-root vertices to have an outgoing edge.

Remark 2.2. Observe that T forms a spanning tree in the undirected sense. This property (along with either property 1 or property 2) can alternatively be used to define an arborescence.

Remark 2.3. It's easy to check if an r-arborescence exists. We can reverse the arcs and run a depth-first search from the root. If all vertices are reached, we have produced an r-arborescence.

The focus of this chapter is to find the minimum-weight r-arborescence. We can simplify things slightly by assuming that all of the weights are non-negative. Because no outgoing arcs from r will be part of any arborescence, we can assume no such arcs exist in G either. For brevity, we fix r and simply say arborescence when we mean r-arborescence.

If there are negative arc weights, add a large positive constant M to every weight. This increases the total weight of each arborescence by $M(n-1)$, and hence the identity of the minimum-weight one remains unchanged.

2.1.1 *The Limitations of Greedy Algorithms*

It's natural to ask if greedy algorithms like those in Chapter 1 for the directed case. E.g., we can try picking the lightest incoming arc into the component containing r, as in Prim's algorithm, but this fails, for example in Figure 2.1. Or we could emulate Kruskal's algorithm and consider arcs in increasing order of weight, adding them if they don't close a directed cycle. (Exercise: give an example where it fails.) The problem is that greedy algorithms (that consider the arcs in some linear order and irrevocably add them in) don't see to work. However, the algorithm we eventually get will feel like Borůvka's algorithm, but one where we are allowed to revoke some of our past decisions.

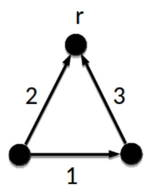


Figure 2.1: A Prim-like algorithm will select the arc with weight 2 and 3, whereas the optimal choices are the arcs with weights 3 and 1.

2.2 *The Chu-Liu/Edmonds/Bock Algorithm*

The algorithm we present was discovered independently by Yoeng-Jin Chu and Tseng-Hong Liu [1], Jack Edmonds, and F. Bock [2]. We will follow Karp's presentation of Edmonds' algorithm.

[1] Edmonds (1967)

[2] Karp (1971)

Definition 2.4. For a vertex $v \in V$ or subset of vertices $S \subseteq V$, let $\partial^+ v$ and $\partial^+ S$ denote the set of arcs leaving the node v and the set S, respectively.

Definition 2.5. For a vertex $v \in V$ in graph G, define $M_G(v) := \min_{a \in \partial^+ v} w(a)$ be the minimum weight among arcs leaving v in G.

The first step is to create a new graph G' by subtracting some weight from each outgoing arc from a vertex, such that there is at least one arc of weight 0. That is, set $w(a') \leftarrow w(a) - M_G(v)$ for all $a \in \partial^+ v$ and each $v \in V$.

Claim 2.6. T is a min-weight arborescence in $G \iff T$ is a min-weight arborescence in G'.

Proof. Each arborescence has exactly one arc leaving each vertex. Decreasing the weight of every arc exiting v by $M_G(v)$ decreases the weight of every possible arborescence by $M_G(v)$ as well. Thus, the set of min-weight arborescences remains unchanged. □

Now each vertex has at least one 0-weight arc leaving it. Now, for each vertex, pick an arbitrary 0-weight arc out of it. If this choice is an arborescence, this must be the minimum-weight arborescence, since all arc weights are still nonnegative. Otherwise, the graph consist of some connected components, each of which has one directed cycle along with some acyclic incoming components, as shown in the figure.

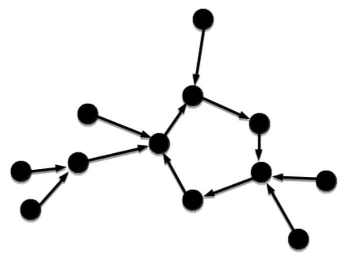


Figure 2.2: An example of a possible component after running the first step of the algorithm

For the second step of the algorithm, consider one such 0-weight cycle C, and construct a new graph $G'' := G'/C$ by contracting the cycle C down to a single new node v_C, removing arcs within C, and replacing parallel arcs by the cheapest of these arcs. Let $\mathrm{OPT}(G)$ denote the weight of the min-weight arborescence on G.

Claim 2.7. $\mathrm{OPT}(G') = \mathrm{OPT}(G'')$.

Proof. To show $\mathrm{OPT}(G') \leq \mathrm{OPT}(G'')$, we exhibit an arborescence in G' with weight at most $\mathrm{OPT}(G'')$. Indeed, let T'' be a min-weight arborescence in G''. Consider arborescence T' in G' obtained by expanding v_C back to the cycle C, and removing one arc in the cycle. Since the cycle has weight 0 on all its arcs, T' has the same weight as T''. (See Figure 2.3.)

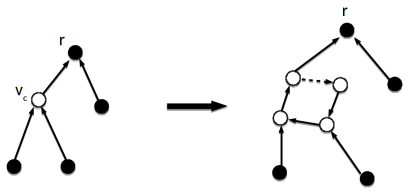


Figure 2.3: The white node is expanded into a 4-cycle, and the dashed arrow is the arc that is removed after expanding.

Now to show $\mathrm{OPT}(G'') \leq \mathrm{OPT}(G')$, take a min-weight arborescence T' of G', and identify the nodes in C down to get a vertex v_C. The resulting graph is clearly connected, with each vertex having a directed path to the root. Now remove some arcs to get an arborescence of G'', e.g., as in Figure 2.4. Since arc weights are non-negative, we can only lower the weight by removing arcs. Therefore $\mathrm{OPT}(G'') \leq \mathrm{OPT}(G')$. □

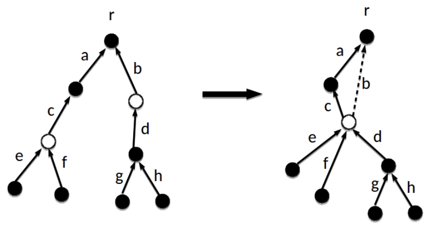


Figure 2.4: Contracting the two white nodes down to a cycle, and removing arc *b*.

The proof also gives an algorithm for finding the min-weight arborescence on G' by contracting the cycle C (in linear time), recursing on G'', and the "lifting" the solution T'' back to a solution T'. Since we recurse on a graph which has at least one fewer nodes, there are at most n recursive calls. Moreover, the weight-reduction, contraction, and lifting steps in each recursive call take $O(m)$ time, so the runtime of the algorithm is $O(mn)$.

Remark 2.8. This is not the best known run-time bound: there are many optimizations possible. Tarjan [3] presents an implementation of the above algorithm using priority queues in $O(\min(m \log n, n^2))$ time, and Gabow, Galil, Spencer and Tarjan [4] give an algorithm to solve the min-weight arborescence problem in $O(n \log n + m)$ time. The best runtime currently known is $O(m \log \log n)$ due to Mendelson et al. [5].

[3]

[4]

[5]

Open problem 2.9. Is there a linear-time (randomized or deterministic) algorithm to find a min-weight arborescence in a digraph G?

2.3 *Linear Programming Methods*

Let us now see an alternate proof of correctness of the algorithm above, this time using linear programming duality. This is how Edmonds originally proved his algorithm to be optimal.

If you have access to the Chu-Liu or Bock papers, I would love to see them.

2.3.1 *Linear Programming Review*

Before we actually represent the arborescence problem as a linear program, we first review some standard definitions and results from linear programming.

Definition 2.10. For some number of variables (a.k.a. dimension) $n \in \mathbb{N}$, number of constraints $m \in \mathbb{N}$, objective vector $c \in \mathbb{R}^n$, constraint matrix $A \in \mathbb{R}^{n \times m}$, and right-hand side $b \in \mathbb{R}^m$, a (minimization) *linear program* (LP) is

This form of the LP is called the *standard form*. More here.

$$\text{minimize } c^\intercal x \quad \text{subject to } Ax \geq b \text{ and } x \geq 0$$

Note that $c^\intercal x$ is the inner product $\sum_{i=1}^n c_i x_i$.

The constraints of a linear program form a ***polyhedron***, which is the convex body formed by the intersection of a finite number of half spaces. Here we have $m + n$ half spaces. There are m of them corresponding to the constraints $\{a_i^\intercal x \geq b_i\}_{i=1}^m$, where $a_i \in \mathbb{R}^n$ is the vector corresponding to the i^{th} row of the matrix A. Moreover, we have n non-negativity constraints $\{x_j \geq 0\}_{j=1}^n$. If the polyhedron is bounded, we call it a ***polytope***.

Whenever we write a vector, we imagine it to be a column vector.

Definition 2.11. A vector $x \in \mathbb{R}^n$ is called *feasible* if it satisfies the constraints: i.e., $Ax \geq b$ and $x \geq 0$.

Definition 2.12. Given a linear program $\min\{c^\intercal x \mid Ax \leq b, x \geq 0\}$, the *dual linear program* is

$$\text{maximize } b^\intercal y \quad \text{subject to } A^\intercal y \leq c \text{ and } y \geq 0$$

The dual linear program has a single variable y_i for each constraint in the original (primal) linear program. This variable can be thought of as giving an importance weight to the constraint, so that taking a linear combination of constraints with these weights shows that the primal cannot possibly surpass a certain value for $c^\intercal x$. This purpose is exemplified by the following theorem.

Theorem 2.13 (Weak Duality)**.** *If x and y are feasible solutions to the linear program* $\min\{c^\intercal x \mid Ax \leq b, x \geq 0\}$ *and its dual, respectively, then* $c^\intercal x \geq b^\intercal y$.

Proof. $c^\intercal x \geq (A^\intercal y)^\intercal x = y^\intercal Ax \geq y^\intercal b = b^\intercal y$. □

This principle of weak duality tells us that if we have feasible solutions x, y where $c^\intercal x = b^\intercal y$, then we know that both x and y are optimal solutions. Our approach will be to give a linear program that models min-weight arborescences, use the algorithm above to write a feasible solution to the primal, and then to exhibit a feasible solution to the dual such that the primal and dual values are the same—hence both must be optimal!

See the ***strong duality theorem*** in add reference for a converse to this theorem. For now, weak duality will suffice.

2.3.2 *Arborescences via Linear Programs*

To analyze the algorithm, we first need to come up with a linear program that "captures" the min-weight arborescence problem. Since we want to find a set of arcs forming an arborescence T, we have one variable x_a for each arc $a \in A$. Ideally, each variable will be an *indicator* for the arc being in the arborescence: i.e., it will binary values: $x_a \in \{0, 1\}$, with $x_a = 1$ if and only if $a \in T$. This choice of variables allows us to express our objective to minimize the total weight: $w^\intercal x := \sum_{a \in A} w(a) x_a$.

Next, we need to come up with a way to express the constraint that T is a valid arborescence. Let $S \subseteq V - \{r\}$ be a set of vertices not containing the root, and some vertex $v \in S$. Every vertex must be able to reach the root by a directed path. If $\partial^+ S \cap T = \emptyset$, there is no arc in T leaving the set S, and hence we have no path from v to r. We conclude that, at a minimum, $\partial^+ S \cap T \neq \emptyset$. We represent this constraint by ensuring that the number of arcs out of S is non-zero, i.e.,

$$\sum_{a \in \partial^+ S} x_a \geq 1.$$

We write an integer linear programming (ILP) formulation for min-weight arborescences as follows:

$$\begin{aligned}
\text{minimize} \sum_{a \in A} w(a) x_a & \\
\text{subject to} \sum_{a \in \partial^+ S} x_a \geq 1 & \quad \forall S \subseteq V - \{r\} \\
\sum_{a \in \partial^+ v} x_a = 1 & \quad \forall v \neq r \\
x_a \in \{0, 1\} & \quad \forall a \in A.
\end{aligned} \tag{2.1}$$

The following lemma is easy to verify:

Lemma 2.14. *T is an arborescence of G with $x_a = \mathbf{1}_{a \in T}$ if and only if x is feasible for the integer LP (2.2). Hence the optimal solution to the ILP (2.1) is exactly the min-weight arborescence.*

Relaxing the Boolean integrality constraints gives us the linear programming relaxation:

$$\begin{aligned}
&\text{minimize} \sum_{a\in A} w(a)x_a \\
&\text{subject to} \sum_{a\in\partial^+ S} x_a \geq 1 \quad \forall S \subseteq V - \{r\} \\
&\qquad\qquad \sum_{a\in\partial^+ v} x_a = 1 \quad \forall v \neq r \\
&\qquad\qquad x_a \geq 0 \qquad\quad \forall a \in A.
\end{aligned} \tag{2.2}$$

Since we have *relaxed* the constraints, the optimal solution to the (fractional) LP (2.2) can only have less value than the ILP (2.1), and hence the optimal value of the LP is at most $\mathrm{OPT}(G)$. In the following, we show that it is in fact *equal* to $\mathrm{OPT}(G)$!

Exercise 2.15. Suppose all the arc weights are non-negative. Show that the optimal solution to the linear program remains unchanged even if drop the constraints $\sum_{a\in\partial^+ v} x_a = 1$.

2.3.3 *Showing Optimality*

The output T of the Chu-Liu/Edmonds/Bock algorithm is an arborescence, and hence the associated solution x (as defined in Lemma 2.14) is feasible for ILP (2.1) and hence for LP (2.2). To show that x is optimal, we now exhibit a vector y feasible for the dual linear program with objective equal to $w^\intercal x$. Now weak duality implies that both x and y must be optimal primal and dual solutions.

The dual linear program for (2.2) is

$$\begin{aligned}
&\text{maximize} \sum_{S\subseteq V-\{r\}} y_S \\
&\text{subject to} \sum_{S:a\in\partial^+ S} y_S \leq w(a) \quad \forall a \in A \\
&\qquad\qquad y_S \geq 0 \quad \forall S \subseteq V - \{r\}, |S| > 1.
\end{aligned} \tag{2.3}$$

Observe that y_S is unconstrained when $|S| = 1$, i.e., S corresponds to a singleton non-root vertex.

We think of y_S as *payments* raised by vertices inside set S so that we can buy an arc leaving S. In order to buy an arc a, we need to raise $w(a)$ dollars. We're trying to raise as much money as possible, while not overpaying for any single arc a.

Lemma 2.16. *If arc weights are non-negative, there exists a solution for the dual LP (2.3) such that $w^\intercal x = \mathbb{1}^\intercal y$, where all y_e values are non-negative.*

Proof. The proof is by induction over the execution of the algorithm.

- The base case is when the chosen zero-weight arcs out of each node form an arborescence. In this case we can set $y_S = 0$ for all S; since all arc weights are non-negative, this is a feasible dual solution. Moreover, both the primal and dual values are zero.

- Suppose we subtract $M := M_G(v)$ from all arcs leaving vertex v in graph G so that v has at least one zero-weight arc leaving it. Let G' be the graph with the new weights, and let T' be the optimal solution on G'. By induction on G', let y' be a non-negative solution such that $\sum_{a \in T'} w'_e = \sum_S y'_S$. Define $y_v := y'_v + M$ and $y_S = y'_S$ for all other subsets; this is the desired feasible dual solution for the same tree $T = T'$ on the original graph G. Indeed, for one of the arcs $a = (v, u)$ out of the node v, we have
$$\begin{aligned}\sum_{S:a\in\partial^+S} y_S &= \sum_{S:a\in\partial^+S,|S|=1} y_S + \sum_{S:a\in\partial^+S,|S|\geq 2} y_S \\ &= (y'_{\{u\}} + M) + \sum_{S:a\in\partial^+S,|S|\geq 2} y'_S \\ &\leq M + w'(a) = M + (w(a) - M) = w(a).\end{aligned}$$
Moreover, the value of the dual increases by M, the same as the increase in the weight of the arborescence.

- Else, suppose the chosen zero-weight arcs contain a cycle C, which we contract down to a node v_C. Using induction for this new graph G', let y' be the feasible dual solution. For any subset S' of nodes in G' that contains the new node v_C, let $S = (S' \setminus \{v_C\}) \cup C$, and define $y_S = y'_{S'}$. For all other subsets S in G' not containing v_C, define $y_S = y'_S$. Moreover, for all nodes $v \in C$, define $y_{\{v\}} = 0$. The dual value remains unchanged, as does the weight of the solution T obtained by lifting T'. The dual constraint changes only arcs of the form $a = (v, u)$, where $v \in C$ and $u \notin C$. But such an arc is replaced by an arc $a' = (v_C, u)$, whose weight is at most $w(a)$. Hence
$$\sum_{S:a\in\partial^+S} y_S = y'_{\{v_C\}} + \sum_{S':a'\in\partial^+S',S'\neq\{v_C\}} y'_S \leq w(a') \leq w(a).$$
This completes the inductive proof

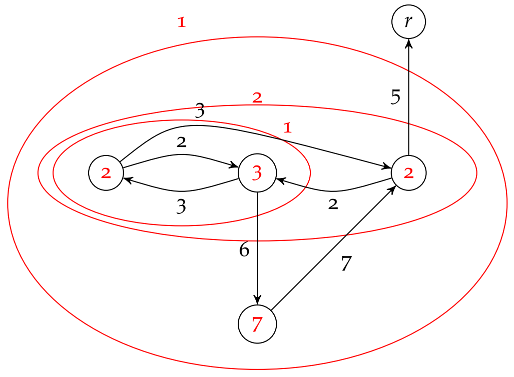


Figure 2.5: An optimal dual solution: vertex sets are labeled with dual values, and arcs with costs.

Notice that the sets with non-zero weights correspond to singleton nodes, or to the various cycles contracted during the algorithm. Hence these sets form a *laminar* family; i.e., any two sets S, S' with non-zero value in y are either disjoint, or one is contained within the other. $\square$

By Lemma 2.16 and weak duality, we conclude that the solution x and the associated arborescence T is optimal. It is easy to extend the argument to potentially negative arc weights.

Corollary 2.17. *There exists a solution for the dual LP (2.3) such that $w^\intercal x = \mathbb{1}^\intercal y$. Hence the algorithm produces an optimal arborescence even for negative arc weights.*

Proof. If some arc weights are negative, add M to all arc weights to get the new graph G' where all arc weights are positive. Let y' be the optimal dual for G' from Lemma 2.16; define $y_S = y'_S$ for all sets of size at least two, and $y_{\{v\}} = y'_{\{v\}} - M$ for singletons. Note that the weight of the optimal solution on G is precisely $M(n-1)$ smaller than on G'; the same is true for the total dual value. Moreover, for arc $e = (u, v)$, we have

$$\sum_{S: a \in \partial^+ S} y_S = \sum_{S: a \in \partial^+ S, |S| \geq 2} y'_S + (y'_{\{u\}} - M) \leq (w_e + M) - M = w_e.$$

The inequality above uses that y' is a feasible LP solution for the graph G' with inflated arc weights. Finally, since the only non-negative values in the dual solution are for singleton sets, all constraints in (2.2) are satisfied for the dual solution y, this completes the proof. □

2.3.4 *Integrality of the Polytope*

The result of Corollary 2.17 is quite exciting: it says that no matter what the objective function of the linear program (i.e., the arc weights $w(a)$), there is an optimal *integral* solution to the linear program, which our combinatorial algorithm finds. In other words, the optimal solutions to the LP (2.2) and the ILP (2.1) are the same.

We will formally discuss this later in the course, but let us start playing with these kinds of ideas. A good start is to visualize this geometrically: let $\mathcal{A} \subseteq \mathbb{R}^{|A|}$ be the set of all solutions to the ILP (which correspond to the characteristic vectors of all valid r-arborescences). This is a finite set of points, and let K_{arb} be the convex hull of these points. (It can be shown that K_{arb} is a polytope, though we don't do it here.) If we optimize a linear function given by some weight vector w over this polytope, we get the optimal arborescence for this weight. This is the solution to ILP (2.1).

Moreover, let $K \subseteq \mathbb{R}^{|A|}$ be the polytope defined by the constraints in the LP relaxation (2.2). Note that each point in $\mathcal{A}$ is contained within K, therefore so is their convex hull K. I.e.,

$$K_{arb} \subseteq K.$$

In general, the two polytopes are not equal. But in this case, Corollary 2.17 implies that for this particular setting, the two are indeed equal. Indeed, a geometric hand-wavy argument is easy to make — if K were strictly bigger than K_{arb}, there would be some direction in which K extends beyond K_{arb}. But each direction corresponds to a weight-vector, and hence for that weight vector the optimal solution within K (which is the solution to the LP) would differ from the optimal solution within K_{arb} (which is the solution to the ILP). This contradicts Corollary 2.17.

2.4 *Matroid Intersection*

More to come here, maybe just a forward pointer to a later lecture.

3
Dynamic Algorithms for Graph Connectivity

Dynamic algorithms is the study of algorithmic problems in the setting where the input changes over time. At each timestep, we get either an ***update*** which tells us how the input has changed, or a ***query*** which demands outputting some aspect of the solution to the underlying problem. In this chapter focus on some basic graph questions, but the area is much broader, and we can consider efficient dynamic algorithms for most algorithmic questions.

For graph problems, the most common model is the *edge-update* model. Hence, the update operations are InsertEdge(e) and DeleteEdge(e), which add and delete some edge e from the graph respectively. (It is useful to also allow InsertNode(v) and DeleteNode(v), which add and delete isolated nodes in the graph; we cannot delete a node that has an edge incident to it.) If we can handle both kinds of edge-update operations, we are said to be in the *fully-dynamic* case, else we may be in the *insert-only* or *delete-only* case. How can we maintain solutions to basic graph problems in these models?

3.1 Dynamic Connectivity

We restrict our attention to the most basic of all graph problems: graph connectivity. As the graph changes via a sequence of Insert and Delete operations, we want to support two types of queries:

- Connected(u, v): are the vertices u, v in the same connected component of the current graph?
- Connected(G): is the current graph G connected.

Here are two naïve approaches for this problem:

1. We keep track of the updates, say using an adjacency matrix, and then run depth-first search each time we get a query. This takes $O(1)$ time for each update, and $O(m + n)$ time for each query.

2. We explicitly maintain the connected components of graph by running depth-first search after each update. Now each query takes $O(1)$ time, but each update may take $O(m)$ time.

If there are no deletions, we can use ***disjoint set union-find*** data structure to implement both updates and queries in $O(\alpha(n))$ amortized time.

These two extreme approaches suggest a trade-off between update time and query time: how can we balance between them to get small query and update times?

3.1.1 *Some Issues to Consider*

The quality of our algorithms will depend on the power we allow them. For example,

- ***worst-case*** versus ***amortized*** update and query times: does the algorithm take at most B time on each step? Or can some of the steps require more time and others less, as long as any sequence of T operations requires at most BT time?
- ***deterministic*** versus ***randomized*** algorithms: does the algorithm flip coins or not?

  Even among randomized algorithms, we may consider questions like ***Las Vegas*** versus ***Monte Carlo*** algorithms: is the algorithm always correct, or is it just correct for each output with high probability? Or that of ***adaptive*** versus ***oblivious*** adversaries: can the request at some step depend on the coin-tosses made by the algorithm in previous steps? We defer these subtler distinctions for now, and just focus on either deterministic algorithms, or on Monte Carlo algorithms against oblivious adversaries, where the input is assumed to be fixed before we flip any coins.

3.1.2 *And Some Results*

The dynamic graph connectivity problem has been studied along all of these axes, which makes it exhausting to give a complete list of the current best results. Let us just list the results we will discuss.

1. For deterministic algorithms with worst-case runtimes, Greg Frederickson showed how to get $O(\sqrt{m})$ update time and $O(1)$ query time. This was improved to $O(\sqrt{n})$ update times by Eppstein et al., but then progress stalled for quite some time. We will discuss Frederickson's approach, and see Eppstein et al.'s extension in a homework.

Only very recently, Chuzhoy et al. broke through the logjam, and obtained $n^{o(1)}$ worst-case update time; the team includes CMU almunus Richard Peng and current student Jason Li.

2. For deterministic algorithms with amortized runtimes, Holm, de Lichtenberg, and Thorup showed $O(\log^2 n)$ update times and $O\left(\frac{\log n}{\log\log n}\right)$ query times.

3. For randomized algorithms with worst-case bounds, Kapron, King, Mountjoy gave an algorithm with poly-logarithmic update and query times.

All three of these algorithms illustrate some clever algorithmic ideas; the running times are not the point here.

3.1.3 *Oh, and a Lower Bound*

A quick word about lower bounds. Sadly, we cannot achieve constant update and query times: a lower bound of Mihai Pătraşcu and Erik Demaine says that if t_u denotes the update time and t_q denotes the query time (both in a deterministic? amortized sense), then

$$t_q \log(t_u/t_q) = \Omega(\log n)$$
$$t_u \log(t_q/t_u) = \Omega(\log n)$$

This implies, for instance, that $\max\{t_u, t_q\} = \Omega(\log n)$. More details to come.

3.2 *Frederickson's algorithm*

We describe a weaker version of Frederickson's algorithm, which has an $O(m^{2/3})$ update time, and $O(1)$ query time. As we mentioned, the algorithm maintains a spanning forest of the graph. To find a replacement edge fast, it clusters the forest into "clusters" that are roughly the same size. Now it keeps track of which clusters have edges going between them. So, when searching for a replacement edge, it can just look at pairs of clusters, instead of at all nodes.

3.2.1 *The Basic Operations*

To avoid worrying about lower-level details, assume we have a DYNAMIC FOREST data structure that supports the following operations:

1. CREATENODE(v) creates an isolated node,
2. TREE(v) returns the identity of the tree containing v,
3. LINK(u, v) adds the edge uv to the trees containing u and v (assuming these trees are distinct), and
4. CUT(u, v) removes the edge uv from its tree.

For simplicity, we assume that these can be done *in constant time*. In practice, these can be implemented, e.g., using the *Euler Tree (ET)* data structure of Monika Henzinger and Valerie King, in $O(\log n)$ worst-case time, or using the *link-cut* data structure of Danny Sleator and Bob Tarjan, in amortized logarithmic time.

3.2.2 *A First Attempt*

Using this data structure, a naïve algorithm to maintain a spanning forest would be the following:

- When an edge uv is added, check if $\textsc{Tree}(u) = \textsc{Tree}(v)$. If so, ignore it, else $\textsc{Link}(u, v)$.

- When an edge uv is deleted and does not belong to the current spanning forest, do nothing. Else, if it belongs to the spanning forest, $\textsc{Cut}(u, v)$. This creates two trees T_u and T_v. *However, there may be edges in G crossing between these trees, so we need to find a replacement edge.*

Iterating over the edges incident to one these trees (say T_u) would take time $\sum_{x \in T_u} \deg(x)$. How can we do it faster? All the algorithms of this section address this replacement edge question in different ways.

3.2.3 *Reduction to the Sub-Cubic Case*

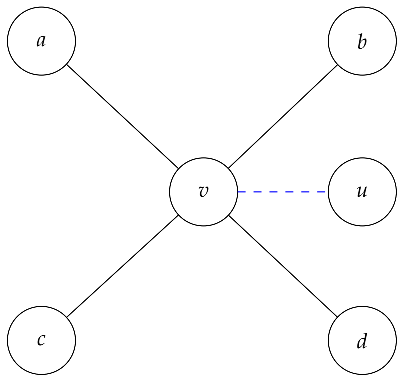
We first reduce to the case where the maximum degree of any node is at most 3. Given updates to a graph $G = (V, E)$, we instead maintain a graph $H = (V', E')$ with $V \subseteq V'$, such that the answer to the queries on G and H are the same. The graph H has $O(m)$ nodes and edges.

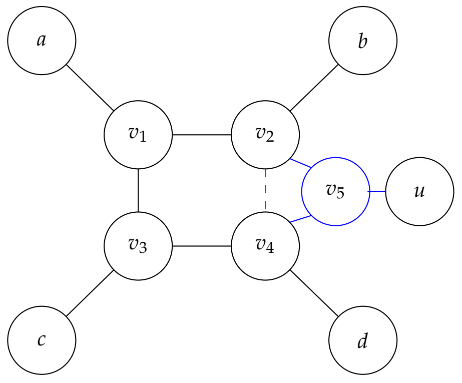
Any constant degree greater than three would suffice.

The mapping is simple: pick any vertex v in G with degree $d \geq 2$, create a cycle $v = v_1, v_2, \ldots, v_d$ in H, and connect each vertex on this cycle to a unique neighbor of v. A vertex of degree one is unchanged. Moreover, this mapping can be maintained dynamically: if an edge $e = uv$ is inserted into G, we may have to increase the size of the cycles for both endpoints (or create a cycle, in case the degree has gone from 1 to 2), and then add an edge. This requires a constant number of $\textsc{InsertNode}$ and $\textsc{InsertEdge}$ operations in H (and maybe one $\textsc{DeleteEdge}$ as well), so the number of updates is maintained up to a constant. Deleting an edge in G is the exact inverse of this process.

We don't need the cycle for nodes that have degree at most 3 in G, but it is easier to enforce a uniform rule.

We need to remember this mapping between elements of G and H: this can be done using hash-tables, and we omit the details here.





3.2.4 *Clustering a Tree*

The advantage of a sub-cubic graph is that any spanning forest F also has maximum degree 3. This allows us to use the following elementary clustering algorithm:

Lemma 3.1 (Tree Clustering)**.** *Given a positive integer z, and any tree T with at least z nodes and maximum degree 3, we can partition its vertex set into clusters such that each cluster (a) induces a connected subtree, and (b) contains between z and $3z$ nodes.*

Proof. Root the tree T at some leaf node. Since the maximum degree is 3, each node has at most two children. Find some node u such that the subtree T_u rooted at u has at least z nodes, but each of its children has strictly less than z nodes in its subtrees. Since there are at most two children, the total number of nodes in T_u is at most $2(z-1)+1 \;\leq\; 2z$, the last $+1$ to account for u itself. Put all the nodes in T_u in a cluster, and remove them from T, and repeat. If this process leaves T with fewer than z nodes at the end, add these remaining nodes to one of its adjacent clusters, making its size at most $3(z-1)+1 \leq 3z$. □

Important Exercise: if T has n_T nodes, a naïve application of this theorem may take $O(n_T^2)$ time. Give an implementation that clusters T in $O(n_T)$ time.

3.2.5 *The Algorithm and Analysis*

To answer connectivity queries, we maintain a spanning forest F of the dynamically changing sub-cubic graph H. Moreover, for each tree in F, we also maintain a clustering like in Lemma 3.1, for a value of z to be choose later. (If some tree in F has size less than z, it forms a *trivial* cluster by itself.) Note that there are $O(m/z)$ non-trivial clusters. Moreover, since all vertices have degree at most 3 in H, each cluster is incident to $O(z)$ edges from H.

Concretely, we use a dynamic tree data structure to maintain a sub-forest of the spanning forest F induced by just the in-cluster edges. Moreover, we use the data structure to also maintain the ***cluster forest***, obtained from the spanning forest F by shrinking each cluster down to a single node. For example, figure to come soon. Moreover, for each pair of clusters C_i, C_j, we keep track of all edges not in F that go between C_i and C_j. This list is non-empty only for clusters in the same tree. Finally, each vertex remembers its (at most three) neighbors in both F and H.

Now we describe how to process updates and queries, first without worrying about the sizes of the clusters:

1. INSERT(uv): We call TREE(u) and TREE(v) to find the clusters C_u and C_v that contain them. Then we call TREE(C_u) and TREE(C_v) on the cluster forest to find their trees T_u and T_v in F. If u, v belong to the same tree in F, update the crossing-edge information for the

pair (C_u, C_v). Else, u, v are in different trees in F, so merge these trees by calling Link(C_u, C_v) in the cluster tree.

2. Delete(uv): If edge uv is not in the spanning forest F, then this deletion does not change F. Just remove uv from the list of cross-cluster edges for the pair of clusters C_u, C_v if needed. Otherwise $uv \in T$ for some tree T in the spanning forest F, and we must find a replacement edge. There are two cases:

   (a) Edge uv goes between two clusters C_u and C_v. Then perform Cut(C_u, C_v) in the cluster forest, which breaks the tree T into T_u, T_v, say. For all clusters $C_i \in T_u$ and $C_j \in T_v$, check the cross-cluster information to see if there is an edge xy from C_i to C_j. Upon finding the first such edge, add it to F, Link the corresponding clusters in the cluster forest, and remove the connecting edge from the list of non-forest edges between C_i and C_j. (You can imagine having just deleted this edge xy from H, and then immediately added it back in.) The total runtime is $O(m^2/z^2)$, in order to check all pairs of clusters (C_i, C_j), plus a constant number of dynamic tree operations.

   (b) Now suppose edge uv lies within some cluster C. Then performing Cut(u, v) breaks the cluster into two parts, say C_u, C_v. First, check if there is an edge within C from C_u to C_v: this takes $O(z)$ time, because there are $O(z)$ nodes in C, each of degree at most 3. (This is where we use the bounded cluster size!) If such an edge xy is found, add it to F and use Link(x, y) to add the in-cluster edge, thereby reconnecting cluster C. There is no need to change any of the other metadata we are maintaining.

   However, if there is no such edge, expand the "search area": look at the two parts T_u, T_v of tree T created by removing uv, and try to find an edge from T_u to T_v as in part (a), which still takes $O(m^2/z^2)$. Hence the total runtime in this case is $O(z + m^2/z^2)$.

One remaining issue to address: the changes above may have caused a constant number of non-trivial clusters to have sizes that are too big or too small. If there is a cluster smaller than z, merge it with an adjacent cluster (perhaps making the result be too big). If there is a cluster that is bigger than $3z$, use the clustering from Lemma 3.1 to recluster with sizes being in the range $[z, 3z]$. Since each cluster only has $O(z)$ edges, using a linear-time implementation uses only $O(z)$ time. Moreover, these changes need to be reflected in the cluster forest, which may require $O(z)$ of the edges in that forest to be changed. Hence, the total overhead of this process is only $O(z)$.

To minimize this worst-case runtime bound of $O(z + m^2/z^2)$, set $z = m^{2/3}$ to balance the two terms and get $O(m^{2/3})$ time for both InsertEdge and DeleteEdge. The InsertNode and DeleteNode operations can be implemented in $O(1)$ time.

3. Query: for queries of type Query(u,v), use Tree(Tree(u)) to get the tree containing u, do the same for v, and check if they are identical. This takes a constant number of dynamic tree operations.

3.2.6 Improvements and Extensions

As mentioned above, the $O(m^{2/3})$ worst-case update time was improved by Frederickson by reducing the $O(m^2/z^2)$ time to search over all pairs of clusters down to $O(m/z)$, using additional features of the Euler tree data structure. Setting $z = \sqrt{m}$ in this case to balance terms gives the $O(m^{1/2})$ update time.

Moreover, the homeworks ask you to derive a cool and fairly generic way to sparsify graphs for dynamic algorithms, which improves the algorithm's dependence on the number of edges m down to a dependence on the number of vertices n.

Finally, the big open question: can we improve on the deterministic worst-case update times of $O(\sqrt{n})$? This problem has only recently seen an improvement: work by Chuzhoy et al. gives $n^{o(1)}$ update times, using the idea of *expander decompositions*, which we hope to cover in a later chapter.

Chuzhoy, Gao, Li, Nanongkai, Peng, Saranurak (2019)

3.3 An Amortized, Deterministic Algorithm

The clustering approach of Frederickson's algorithm was to focus the search on pairs of clusters, instead on doing the naïve search. However, if we allow amortized update times instead of worst-case ones, we can avoid this clustering approach, and instead use a charging scheme to keep track of the work we do.

In this charging scheme, we associate a ***level*** with each edge of the dynamic graph G, which is zero when the edge is just inserted. Now when T breaks into L, R because some edge is deleted, we want to scan the edges incident to one of the sides (say the side L with fewer vertices) to check if they remain within L, or if they go to R. (Because T was a tree in a spanning forest, there can be no other options: do you see why?) When we scan an edge e and it fails to connect L, R, we increase its level by 1—this "pays" for the failed work we did in scanning e.

Of course, such a charging scheme is completely useless if the edge levels can increase unboundedly. So we maintain some invariants that ensure the level of any edge stays at most $\log_2 n$.

3.3.1 *A Collection of Layered Spanning Forests*

Here is how we do it. We imagine there being many graphs

$$G = G_0 \supseteq G_1 \supseteq \cdots \supseteq G_i \cdots ,$$

where graph G_i consists of the edges with *level i or higher*. We maintain a spanning forest F_0 of G_0, with the additional property that $F_i := F_0 \cap G_i$ is also a spanning forest of G_i. An equivalent way to think of this is as follows:

($\star$) F_0 is a max-weight spanning forest of G_0, w.r.t. the edge-levels.

Adding an edge is easy: set its level to zero, and add it to F_0 if it does not create a cycle in F_0. Deleting an edge not in F_0 is also easy: just delete it. Finally, if an edge $e \in F_0$ (say with level ℓ) is deleted, we need to search for a replacement. Such a replacement edge can only be at level ℓ or lower—this follows from property ($\star$), and the Cut Rule from **??**. Moreover, to maintain property ($\star$), we should also add a replacement edge of the highest level possible. So we consider off-tree edges from level ℓ downwards.

Remember, when we scan an edge, we want to raise its level, to charge to it. That could mess with probability ($\star$), because this may cause off-tree edges to have higher level/weight than tree edges. To avoid this problem, we first raise the levels of some of the tree edges. Specifically, we do the following steps:

1. Say deleting e breaks tree $T \in F_\ell$ into L_ℓ, R_ℓ. Let $|L_\ell| \leq |R_\ell|$; then scan the edges incident to L_ℓ. (This choice of the smaller side will help us bound the number of levels by $\log_2 n$.) Raise all level-ℓ edges in L_ℓ to have level $\ell + 1$.

2. Scan level-ℓ edges incident to L_ℓ. If some edge e connects to R_ℓ, this is a replacement edge. Add e to F_0 (and hence to $F_1, \ldots, F_\ell$). Do not raise its level, though. Stop: you are done.

   Else, if the edge stays within L_ℓ, raise its level, and try another.

If we fail to find a replacement edge at level ℓ, we lower ℓ by one, and try the steps again. Finally if we don't find a replacement edge between L_0 and R_0, we stop—there is no replacement edge, and the deletion has increased the number of components in the graph. (Convince yourself that ($\star$) remains satisfied when we do all this.)

3.3.2 *Bounding the Maximum Level*

It just remains to show that no edge can have level more than $\log_2 n$—this is what all the steps above have been leading up to.

Lemma 3.2. *Each tree in forest F_ℓ has at most $\lfloor n/2^\ell \rfloor$ nodes. Hence, the level ℓ of any edge is at most $\log_2 n$.*

Proof. This is clearly true for F_0. Now a tree in F_ℓ is formed by raising the levels of some "left" piece $L_{\ell-1}$. It was the smaller half of some tree in level $\ell - 1$, which by induction has size at most $\lfloor n/2^{\ell-1} \rfloor$. Half of that is at most $\lfloor n/2^\ell \rfloor$. Finally, since each tree must have size at least 2, the level ℓ of any edge can be at most $\log_2 n$. □

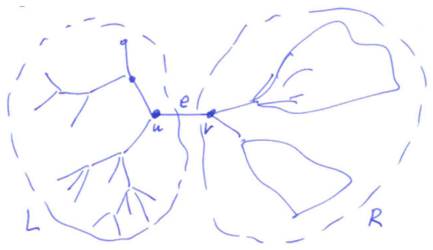
Observe: a linear number of edges can have levels $\approx \log_2 n$, but they must lie in small components. I find this particular charging quite clever: however, variants of this idea (of "charging to the smaller side") arise all the time.

Let us summarize the ideas in this algorithm: we raise levels of edges to pay for the work we do in finding a replacement edge. Each edge has level $O(\log n)$, so the amortized number of scans is $O(\log n)$. We did not talk about the details of the implementation here, but each of these operations of raising levels and maintaining spanning forests in amortized $O(\log n)$ time, using a dynamic tree data structure. This gives an amortized update time of $O(\log^2 n)$. The current best amortized update-time bound for deterministic algorithms is only slightly better, at $O(\frac{\log^2 n}{\log \log n})$[1].

1

3.4 A Randomized Algorithm for Dynamic Graph Connectivity

Finally, let's sketch a randomized approach to finding replacement edges, which leads to a Monte-Carlo randomized (albeit worst-case) update time algorithm due to Bruce Kapron, Valerie King, and Alan Mountjoy. To simplify matters, we consider the case where a *single* edge-deletion occurs: this will be enough to highlight two interesting techniques of their algorithm.



3.4.1 Giving Nodes a Bit Representation, and the Power of XOR

Suppose we delete edge $e = uv$, causing the tree T to split into L and R. As a thought experiment, suppose there is a *single* replacement edge f in G that goes between L and R: all other edges incident to L and R do not cross. How would we find f?

We begin by assigning to each node v a $(\log_2 n)$-bit label $\ell(v)$. Assume some total ordering on the nodes, so the edge uv (where $u \prec v$ in this ordering) has a $2\log_2 n$-bit label that concatenates the names of its two vertices in that order:

$$\ell(e) := [\ell(u), \ell(v)].$$

Definition 3.3 (Node Fingerprint)**.** The *fingerprint* $\mathcal{F}(v)$ of node v is defined to be the exclusive-or of labels of all edges incident to v in G

$$cF(v) := \bigoplus_{e \in \partial_G v} \ell(e) .$$

Note: we consider all neighbors in the graph G, not just in the spanning forest.

It is easy and fast to maintain the fingerprint $\mathcal{F}(v)$ for each vertex v as edges change. Now when we delete an edge uv (and if there is a unique replacement edge f), consider the following:

Fact 3.4. The label of the unique replacement edge is given by the exclusive-or of the fingerprints of all nodes in L. That is,

$$\bigoplus_{v \in L} \mathcal{F}(v) = \ell(f)$$

Proof. Indeed, each edge with both endpoints in L will be present twice in the exclusive-or, and hence will be zeroed out. The only contribution remaining will be $\ell(f)$. □

3.4.2 *Multiple Edges? Subsample!*

What if there are multiple possible replacement edges going from L to R? The result of the calculation in Fact 3.4 will be the XOR of the labels of all these edges. If this is non-zero, it would signal that there is a replacement edge, but not tell us its name. But the XOR may even be zero if there are crossing edges. However, all is not lost—our secret weapon will be randomness, which we have not used yet.

Let us make an weaker assumption this time: *suppose there are some k replacement edges, and we know this value k.* We now keep a subsampled graph G' by picking each edge in G with probability $1/k$. Then for any $k \geq 1$,

$$\Pr(\text{exists unique } L\text{-to-}R \text{ edge in } G') = k \cdot \frac{1}{k}\left(1 - \frac{1}{k}\right)^{k-1} \geq \frac{1}{e}. \quad (3.1)$$

So with constant probability, there is a unique replacement edge in G'. So if we define the fingerprints in terms of edges in G', the same idea would work. Note that we don't need to know k precisely: using any value in $[k/2, 2k]$ would give some constant probability of having a unique replacement edge. Moreover, by repeating this process $O(\log n)$ times independently—i.e., keeping $O(\log n)$ independent sets of fingerprints—we can ensure that one of these repetitions will give a unique crossing edge with probability $1 - 1/\operatorname{poly}(n)$.

Finally, what if we don't even have a crude estimate of k? We can try subsampling at rates $1/2, 1/4, \ldots, 1/2^i, \ldots, 1/n$, i.e., at *all powers* *of* 2 between $1/2$ and $1/n$. There are $\log_2 n$ such values, and one of them will be the correct one, up to a factor of 2. Specifically, we keep $O(\log n)$ different data structures, each one using a different one of these $O(\log n)$ sampling rates; at least one of them (the "right" one) will succeed with probability $1 - \frac{O(\log n)}{\text{poly}(n)}$, by a trivial union bound.

3.4.3 *Wrapping Up*

Many of the random experiments would have multiple L-to-R edges: the answers for those would not make sense: they may give names of non-edges, or of edges that do not cross between L and R. Hence, the algorithm above needs a mechanism to check that the answer is indeed an L-to-R edge. Details of this can be found in the Kapron, King, and Mountjoy paper.

More worryingly, what about multiple deletions? We might be tempted to say that if the input sequence is oblivious to the algorithm's randomness, we can just take a union bound over all timesteps. However, the algorithm's behavior—the structure of the current spanning forest F, and hence which cut is queried during later edge deletions—depends on the replacement edges found in previous steps, which are correlated with the randomness. Hence, we cannot claim independence for the calculations in (3.1). To handle this, Kapron et al. construct a multi-level data structure; see their paper for details.

Finally, we did not talk about any of the implementation details here. For instance, how can we compute the XOR of the set of nodes in L quickly? How do we check if the replacement edge names are valid? All these can be done using a dynamic tree data structure. Putting all this toget, the update time becomes $O(\log^5 n)$, still in the worst-case. As an aside, if we allow randomization and amortized bounds, the update times can be improved to within $\text{poly}(\log \log n)$ factors of the lower bound of $O(\log n)$; see this work by [2].

[2]

A closely related algorithm is due to Ahn, Guha, and MacGregor. Discussion of future developments to come soon.

4
Shortest Paths in Graphs

In this chapter, we look at another basic algorithmic construct: given a graph where edges have weights, find the shortest path between two specified vertices in it. Here the weight of a path is the sum of the weights of the edges in it. Or given a source vertex, find shortest paths to all other vertices. Or find shortest paths between all pairs of vertices in the graph. Of course, each harder problem can be solved by multiple calls of the easier ones, but can we do better?

Let us give some notation. The input is a graph $G = (V, E)$, with each edge $e = uv$ having a weight/length $w_{uv} \in \mathbb{R}$. For most of this chapter, the graphs will be directed: in this case we use the terms *edges* and *arcs* interchangeably, and an edge uv is imagined as being directed from u to v (i.e., from left to right). Given a *source* vertex s, the ***single-source shortest paths (SSSP)*** asks for the weights of shortest paths from s to each other vertex in V. The ***all-pairs shortest paths (APSP)*** problem asks for shortest paths between each pair of vertices in V. (In the worst-case, algorithms for the s-t-shortest-path problem also solve the SSSP, so we do not consider these any further.) We will consider both these variants, and give multiple algorithms for both.

Given the graph and edge-weights, the weight of a shores path between u and v is often called their ***distance***.

There is another source of complexity: whether the edge-weights are all non-negative, or if they are allowed to take on negative values. In the latter case, we disallow cycles of negative weight, else the shortest-path is not well-defined, since such a cycle allows for ever-smaller shortest paths as we can just run around the cycle to reduce the total weight arbitrarily.

We could ask for a shortest ***simple*** path. However, this problem is NP-hard in general, via a reduction from Hamilton path.

4.1 *Single-Source Shortest Path Algorithms*

The ***single-source shortest path problem*** (SSSP) is to find a shortest path from a single source vertex s to every other vertex in the graph. The output of this algorithm can either be the $n - 1$ numbers giving the weights of the $n - 1$ shortest paths, or (some compact representation of) these paths. We first consider Dijkstra's algorithm for the case of non-negative edge-weights, and give the Bellman-Ford algorithm that handles negative weights as well.

4.1.1 Dijkstra's Algorithm for Non-negative Weights

Dijkstra's algorithm keeps an estimate `dist` of the distance from s to every other vertex. Initially the estimate of the distance from s to itself is set to 0 (which is correct), and is set to ∞ for all other vertices (which is typically an over-estimate). All vertices are unmarked. Then repeatedly, the algorithm finds an umarked vertex u with the smallest current estimate, marks this vertex (thereby indicating that this estimate is correct), and then updates the estimates for all vertices v reachable by arcs uv thus:

This update step is often said to ***relax*** the edges out of u, which has a nice physical interpretation. Indeed, any edge uv for which the $\text{dist}(v)$ is strictly bigger than $\text{dist}(u) + w_{uv}$ can be imagined to be over-stretched, which this update fixes.

$$\text{dist}(v) \leftarrow min\{\text{dist}(v), \text{dist}(u) + w_{uv}\}$$

We keep all the vertices that are not marked and their estimated distances in a priority queue, and extract the minimum in each iteration.

Algorithm 2: Dijkstra's Algorithm

```
Input: Digraph $G = (V, E)$ with edge-weights $w_e \geq 0$ and
       source vertex $s \in G$
Output: The shortest-path distances from each vertex to $s$
2.1 add $s$ to heap with key 0
2.2 for $v \in V \setminus \{s\}$ do
2.3     add $v$ to heap with key $\infty$
2.4 while heap not empty do
2.5     $u \leftarrow$ deletemin
2.6     for $v$ a neighbor of $u$ do
2.7         key($v$) $\leftarrow$ min{ key($v$), key($u$) + $w_{uv}$ }        // relax $uv$
```

To prove the correctness of the algorithm, it suffices to show that each time we extract a vertex u with the minimum estimated distance from the priority queue, the estimate for that vertex u is indeed the distance from s to u. This can be proved by induction on the number of marked vertices, and left as an exercise. Also left as an exercise are the modifications to return the shortest-path tree from node s.

The time complexity of the algorithm depends on the priority queue data structure. E.g., if we use binary heap, which incurs $O(\log n)$ for `decrease-key` as well as `extract-min` operations, we incur a running time of $O(m \log n)$. But just like for spanning trees, we can do better with Fibonacci heaps, which implement the ***decrease-key*** operation in constant amortized time, and `extract-min` in $O(\log n)$ time. Since Dijkstra's algorithm uses n inserts, n deletemins, and m decrease-keys, this improves the running time to $O(m + n \log n)$.

There have been many other improvements since Dijkstra's original work. If the edge-weights are integers in $\{0,\ldots,C\}$, a clever priority queue data structure of Peter van Emde Boas[1] can be used instead; this implements all operations in time $O(\log\log C)$. Carefully using it can give us runtimes of $O(m\log\log C)$ and $O(m+n\sqrt{\log C})$ (see Ahuja et al. [2]). Later, [3] showed a faster implementation for the case that the weights are integer, which has the running time of $O(m+n\log\log(n))$ time. Currently, latest results to come here.

(Dijkstra (1959)

The paper also gives his version of the Járnik/Prim MST algorithm. The two algorithms are not that different, since the MST algorithm merely changes the update rule to $\text{dist}(v) \leftarrow \min\{\text{dist}(v), w_{uv}\}$.

1

2

3

4.1.2 *The Bellman-Ford Algorithm*

Dijkstra's algorithm does not work on instances with negative edge weights. For example, it will return a distance of 2 for the vertex a, whereas the correct shortest-path from s to a goes via b, and has weight $3-2=1$. For such instances, a correct SSSP algorithm must either return the distances from s to all other vertices, or else find a negative-weight cycle in the graph.

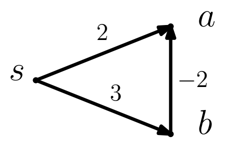


Figure 4.1: Example in which Dijkstra's algoirthm does not work on

The most well-known algorithm for this case is the Shimbel-Bellman-Ford algorithm. [4] Just like Dijkstra's algorithm, this algorithm also starts with an overestimate of the shortest path to each vertex. However, instead of relaxing the out-arcs from each vertex once (in a careful order), this algorithm relaxes the out-arcs of all the vertices $n-1$ times, in round-robin fashion. Formally, the algorithm is the following. (A visualization can be found [5].)

[4] This algorithm also has a complicated history. The algorithm was first stated by Shimbel in 1954, then Moore in '57, Woodbury and Dantzig in '57, and finally by Bellman in '58. Since it used Ford's idea of relaxing edges, the algorithm "naturally" came to be known as Bellman-Ford.

5

Algorithm 3: The Bellman-Ford Algorithm

```
  Input: A digraph $G=(V,E)$ with edge weights $w_e \in \mathbb{R}$, and
         source vertex $s \in V$
  Output: The shortest-path distances from each vertex to $s$, or
          report that a negative-weight cycle exists
3.1 $dist(s) = 0$                        // the source has distance 0
3.2 for $v \in V$ do
3.3     $dist(v) \leftarrow \infty$
3.4 for $|V|$ iterations do
3.5     for edge $e=(u,v) \in E$ do
3.6         $dist(v) \leftarrow \min\{dist(v), dist(u)+weight(e)\}$
3.7 If any distances changed in the last ($n^{th}$) iteration, output "$G$
      has a negative weight cycle".
```

The proof relies on the following lemma, which is easily proved by induction on i.

Lemma 4.1. *After i iterations of the algorithm, $dist(v)$ equals the weight of the shortest-path from s to v containing at most i edges. (This is defined to be ∞ if there are no such paths.)*

If there is no negative-weight cycle, then the shortest-paths are well-defined and simple, so a shortest-path contains at most $n-1$ edges. Now the algorithm is guaranteed to be correct after $n-1$ iterations by Lemma 4.4; moreover, none of the distances will change in the n^{th} iteration.

However, suppose the graph contains a negative cycle that is reachable from the source. Then the labels $dist(u)$ for vertices on this cycle continue to decrease in each subsequent iteration, because we may reach to any point on this cycle and by moving in that cycle we can accumulate negative distance; therefore, the distance will get smaller and smaller in each iteration. Specifically, they will decrease in the n^{th} iteration, and this decrease signals the existence of a negative-weight cycle reachable from s. (Note that if none of the negative-weight cycles C are reachable from s, the algorithm outputs a correct solution despite C's existence, and it will produce the distance of ∞ for all the vertices in that cycle.)

The runtime is $O(mn)$, since each iteration of Bellman-Ford looks at each edge once, and there are n iterations. This is still the fastest algorithm known for SSSP with general edge-weights, even though faster algorithms are known for some special cases (e.g., when the graph is planar, or has some special structure, or when the edge weights are "well-behaved"). E.g., for the case where all edge weights are integers in the range $[-C, \infty)$, we can compute SSSP in time $O(m\sqrt{n}\log C)$, using an idea we may discuss in Homework #1.

4.2 The All-Pairs Shortest Paths Problem (APSP)

The obvious way to do this is to run an algorithm for SSSP n times, each time with a different vertex being the source. This gives an $O(mn + n^2 \log n)$ runtime for non-negative edge weights (using n runs of Dijkstra), and $O(mn^2)$ for general edge weights (using n runs of Bellman-Ford). Fortunately, there is a clever trick to bypass this extra loss, and still get a runtime of $O(mn + n^2 \log n)$ with general edge weights. This is known as Johnson's algorithm, which we discuss next.

4.2.1 Johnson's Algorithm and Feasible Potentials

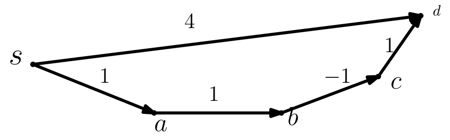


Figure 4.2: A graph with negative edges in which adding positive constant to all the edges will change the shortest paths

The idea behind this algorithm is to (a) re-weight the edges so that they are nonnegative yet preserve shortest paths, and then (b) run n instances of Dijkstra's algorithm to get all the shortest-path distances. A simple-minded hope (based on our idea for MSTs) would be to add a positive number to all the weights to make them positive. Although this preserves MSTs, it doesn't preserve shortest paths. For instance, the example on the right has a single negative-weight edge. Adding

1 to all edge weights makes them all have non-negative weights, but the shortest path from s to d is changed.

Don Johnson gave a algorithm that does the edge re-weighting in a slightly cleverer way, using the idea of ***feasible potentials***. Loosely, it runs the Bellman-Ford algorithm once, then uses the information gathered to do the re-weighting. At first glance, the concept of a *feasible potential* does not seem very useful. It is just an assignment of weights ϕ_v to each vertex v of the graph, with some conditions:

Johnson (1977)

Definition 4.2. For a weighted digraph $G = (V, A)$, a function $\phi : V \to \mathbb{R}$ is a *feasible potential* if for all edges $e = uv \in A$,

$$\phi(u) + w_{uv} - \phi(v) \geq 0.$$

Given a feasible potential, we can transform the edge-weights of the graph from w_{uv} to

$$\widehat{w}_{uv} := w_{uv} + \phi(u) - \phi(v).$$

Observe the following facts:

1. The new weights $\widehat{w}$ are all positive. This comes from the definition of the feasible potential.

2. Let P_{ab} be a path from a to b. Let $\ell(P_{ab})$ be the length of P_{ab} when we use the weights w, and $\hat{\ell}(P_{ab})$ be its length when we use the weights $\widehat{w}$. Then

$$\widehat{\ell}(P_{ab}) = \ell(P_{ab}) + \phi_a - \phi_b.$$

   The change in the path length is $\phi_a - \phi_b$, which is independent of the path. So the new weights $\widehat{w}$ preserve the shortest a-to-b paths, only changing the length by $\phi_a - \phi_b$.

This means that if we find a feasible potential, we can compute the new weights $\widehat{w}$, and then run Dijkstra's algorithm on the remaining graph. But how can we find feasible potentials? Here's the short answer: Bellman-Ford. Indeed, suppose there some ***source*** vertex $s \in V$ such that every vertex in V is reachable from s. Then, set $\phi(v) = \text{dist}(s, v)$.

It is cleaner (and algorithmically simpler) to just add a new vertex s and add zero-weight edges from it to all the original vertices. This does not change any of the original distances, or create any new cycles.

Lemma 4.3. *Given a digraph $G = (V, A)$ with vertex s such that all vertices are reachable from s, $\phi(v) = dist(s, v)$ is a feasible potential for G.*

Proof. Since every vertex is reachable from s, $dist(s, v)$ and therefore $\phi(v)$ is well-defined. For an edge $e = uv \in A$, taking the shortest path from s to u, and adding on the arc uv gives a path from s to v, whose length is $\phi(u) + w_{uv}$. This length is at least $\phi(v)$, the length of the shortest path from s to v, and the lemma follows. ☐

In summary, the algorithm is the following:

Algorithm 4: Johnson's Algorithm

```
Input: A weighted digraph $G = (V, A)$
Output: A list of the all-pairs shortest paths for $G$
4.1 $V' \leftarrow V \cup \{s\}$                  // add a new source vertex
4.2 $A' \leftarrow E \cup \{(s, v, 0) \mid v \in V\}$
4.3 $dist \leftarrow \text{BellmanFord}((V', A'))$
                                         // set feasible potentials
4.4 for $e = (u, v) \in A$ do
4.5     $weight(e) += dist(u) - dist(v)$
4.6 $L = []$                                        // the result
4.7 for $v \in V$ do
4.8     $L += \text{Dijkstra}(G, v)$
4.9 return $L$
```

We now bound the running time. Running Bellman-Ford once takes $O(mn)$ time, computing the "reduced" weights $\widehat{w}$ requires $O(m)$ time, and the n Dijkstra calls take $O(n(m + n \log n))$, if we use Fibonacci heaps. Therefore, the overall running time is $O(mn + n^2 \log n)$—almost the same as one SSSP computation, except on very sparse graphs with $m = o(n \log n)$.

4.2.2 *More on Feasible Potentials*

How did we decide to use the shortest-path distances from s as our feasible potentials? Here's some more observations, which give us a better sense of these potentials, and which lead us to the solution.

1. If all edge-weights are non-negative, then $\phi(v) = 0$ is a feasible potential.

2. Adding a constant to a feasible potential gives another feasible potential.

3. If there is a negative cycle in the graph, there can be no feasible potential. Indeed, the sum of the new weights along the cycle is the same as the sum of the original weights, due to the telescoping sum. But since the new weights are non-negative, so the old weight of the cycle must be, too.

4. If we set $\phi(s) = 0$ for some vertex s, then $\phi(v)$ for any other vertex v is an underestimate of the s-to-v distance. This is because for all the paths from s to v we have

$$0 \leq \widehat{\ell}(P_{sv}) = \ell(P_{sv}) - \phi_v + \phi_s = \ell(P_{sv}) - \phi_v,$$

giving $\ell(P_{sv}) \geq \phi_v$. Now if we try to set $\phi(s)$ to zero and try to maximize summation of $\phi(v)$ for other vertices subject to the feasible potential constraints we will get an LP that is the dual of the shortest path LP.

$$\begin{aligned} \text{Maximize} \quad & \sum_{x \in V} \phi_x \\ \text{Subject to} \quad & \phi_s = 0 \\ & w_{vu} + \phi_v - \phi_u \geq 0 \qquad \forall (v,u) \in E \end{aligned}$$

4.2.3 *The Floyd-Warshall Algorithm*

The Floyd-Warshall algorithm is perhaps best introduced via its strikingly simple pseudocode. It first puts down estimates $\text{dist}(u,v)$ for the distances thus:

The naming of this algorithm does not disappoint: it was discovered by Bernard Roy, Stephen Warshall, Bob Floyd, and others. The name tells only a small part of the story.

$$\text{dist}_{ij} = \begin{cases} w_{ij}, & i,j \in E \\ \infty & i,j \notin E, i \neq j \\ 0, & i = j \end{cases} .$$

Then it runs the following series of updates.

Algorithm 5: The Floyd-Warshall Algorithm

```
Input: A weighted digraph $D = (V, A)$
Output: A list of the all-pairs shortest paths for $D$
5.1 set $d(x,y) \leftarrow w_{xy}$ if $(x,y) \in E$, else $d(x,y) \leftarrow \infty$
5.2 for $z \in V$ do
5.3     for $x, y \in V$ do
5.4         $d(x,y) \leftarrow \min\{d(x,y), d(x,z) + d(z,y)\}$
```

Importantly, we run over the "inner" index z in the outermost loop. The proof of correctness is similar to, yet not that same as that of Algorithm 3, and is again left as a simple exercise in induction.

Lemma 4.4. *After we have considered vertices $V_k = \{z_1, \ldots, z_k\}$ in the outer loop, $dist(u,v)$ equals the weight of the shortest x-y path that uses only the vertices from V_k as internal vertices. (This is ∞ if there are no such paths.)*

The running time of Floyd-Warshall is clearly $O(n^3)$—no better than Johnson's algorithm. But it does have a few advantages: it is simple, and it is quick to implement with minimal errors. (The most common error is nesting the for-loops in reverse.) Another advantage is that Floyd-Warshall is also parellelizable, and very cache efficient.

Actually, this paper of Hide, Kumabe, and Maehara (2019) shows that even if you get the loops wrong, but you run the algorithm a few more times, it all works out in the end. But that proof requires a bit more work.

4.3 *Min-Sum Products and APSPs*

A conceptually different way to get shortest-path algorithms is via matrix products. These may not seem relevant, ***a priori***, but they lead to deep insights about the APSP problem.

Recall the classic definition of matrix multiplication, for two real-valued matrices $A, B \in \mathbb{R}^{n \times n}$

$$(AB)_{ij} = \sum_{k=0}^{n} (A_{ik} * B_{kj}).$$

Hence, each entry of the product AB is a sum of products, both being the familar operations over the field $(\mathbb{R}, +, *)$. But now, what if we change the constituent operations, to replace the sum with the `min` operation, and the product with a sum? We get the ***Min-Sum Product***(MSP): given matrices $A, B \in \mathbb{R}^{n \times n}$, the new product is

$$(A \circledcirc B)_{ij} = \min_k \{A_{ik} + B_{kj}\}.$$

This is the usual matrix multiplication, but over the ***semiring*** $(\mathbb{R}, \min, +)$.

A ***semiring*** has a notion of addition and one of multiplication. However, neither the addition nor the multiplication operations are required to have inverses.

It turns out that computing Min-Sum Products is precisely the operation needed for the APSP problem. Indeed, initialize a matrix D exactly as in the Floyd-Warshall algorithm:

$$D_{ij} = \begin{cases} w_{ij}, & i,j \in E \\ \infty & i,j \notin E, i \neq j \\ 0, & i = j \end{cases}.$$

Now $(D \circledcirc D)_{ij}$ represents the cheapest i-j path using at most 2 hops! (It's as though we made the outer-most loop of Floyd-Warshall into the inner-most loop.) Similarly, we can compute

$$D^{\circledcirc k} := \underbrace{D \circledcirc D \ \circledcirc D \cdots \circledcirc D}_{k-1 \text{ MSPs}},$$

whose entries give the shortest i-j paths using at most k hops (or at most $k-1$ intermediate nodes). Since the shortest paths would have at most $n-1$ hops, we can compute $D^{\circledcirc n-1}$.

How much time would this take? The very definition of MSP shows how to implement it in $O(n^3)$ time. But performing it $n-1$ times would be $O(n)$ worse than all other approaches! But here's a classical trick, which probably goes back to the Babylonians: for any integer k,

$$D^{\circledcirc 2k} = D^{\circledcirc k} \circledcirc D^{\circledcirc k}.$$

(Here we use that the underlying operations are associative.) Now it is a simple exercise to compute $D^{\circledcirc n-1}$ using at most $2\log_2 n$ MSPs. This a runtime of $O(MSP(n) \log n)$, where $MSP(n)$ is the time it takes to compute the min-sum product of two $n \times n$ matrices. Now using the naive implementation of MSP gives a total runtime of $O(n^3 \log n)$, which is almost in the right ballpark! The natural question is: can we implement MSPs faster?

In fact, with some more work, we can implement APSP in time $O(MSP(n))$; you will probably see this in a homework.

4.3.1 Faster Algorithms for Matrix Multiplication

Can we get algorithms for MSP that run in time $O(n^{3-\varepsilon})$ for some constant $\varepsilon > 0$? To answer this question, we can first consider the more common case, that of matrix multiplication over the reals (or over some field)? Here, the answer is yes, and this has been known for now over 50 years. In 1969, Volker Strassen showed that one could multiply $n \times n$ matrices over any field $\mathbb{F}$, using $O(n^{\log_2 7}) = O(n^{2.81}$ additions and multiplications. (One can allow divisions as well, but Strassen showed that divisions do not help asymptotically.)

Mike Paterson has a beautiful but still mysterious geometric interpretation of the sub-problems Strassen comes up with, and how they relate to Karatsuba's algorithm to multiply numbers.

If we define ***the exponent of matrix multiplication*** $\omega > 0$ to be smallest real such that two $n \times n$ matrices over any field $\mathbb{F}$ can be multiplied in time $O(n^{\omega})$, then Strassen's result can be phrased as saying:

$$\omega \leq \log_2 7.$$

This value, and Strassen's idea, has been refined over the years, to its current value of 2.3728 due to François Le Gall (2014). (See this survey by Virginia for a discussion of algorithmic progress until 2013.) There has been a flurry of work on lower bounds as well, e.g., by Josh Alman and Virginia Vassilevska Williams showing limitations for all known approaches.

The big improvements in this line of work were due to Arnold Schönhage (1981), Don Coppersmith and Shmuel Winograd (1990), with recent refinements by Andrew Stothers, CMU alumna Virginia Vassilevska Williams, and François Le Gall (2014).

But how about $MSP(n)$? Sadly, progress on this has been less impressive. Despite much effort, we don't even know if it can be done in $O(n^{3-\epsilon})$ time. In fact, most of the recent work has been on giving evidence that getting sub-cubic algorithms for MSP and APSP may not be possible. There is an interesting theory of ***hardness within P*** developed around this problem, and related ones. For instance, it is now known that several problems are equivalent to APSP, and truly sub-cubic algorithms for one will lead to sub-cubic algorithms for all of them.

Yet there is some interesting progress on the positive side, albeit qualitatively small. As far back as 1976, Fredman had shown an algorithm to compute MSP in $O(n^3 \frac{\log \log n}{\log n})$ time. He used the fact that the *decision-tree complexity* of APSP is sub-cubic (a result we will discuss in §4.5) in order to speed up computations over nearly-xlogarithmic-sized sub-instances; this gives the improvement above. More recently, another CMU alumnus Ryan Williams improved on this idea quite substantially to $O\left(\frac{n^3}{2^{\sqrt{\log n}}}\right)^6$, using very interesting ideas from circuit complexity. We will discuss this result in a later section, if we get a chance.

6

4.4 *Undirected APSP Using Fast Matrix Multiplication*

One case were we have seen improvements in APSP algorithms is that of graphs with small integer edge-weights. Our focus will be on ***undirected, unweighted*** graphs: we present a beautiful algorithm of Raimund Seidel that runs in time $O(n^\omega \log n)$, assuming that $\omega > 2$.

To begin, the adjacency matrix A for graph G is the symmetric matrix

$$A_{ij} = \begin{cases} 1 & ij \in E \\ 0 & ij \notin E \end{cases}.$$

Now consider the graph G^2, the ***square of** G*, which has the same vertex set as G but where an edge in G^2 corresponds to being at most two hops away in G—that is, $uv \in E(G^2) \iff d_G(u,v) \leq 2$. If we consider A as a matrix over the finite field $(\mathbb{F}_2, +, *)$, then the adjacency matrix of G^2 has a nice formulation:

In this field over $\{0,1\}$, observe that the multiplication operation behaves just like the Boolean AND function, and the addition like the Boolean XOR.

$$A_{G^2} = A_G * A_G + A_G.$$

This shows how to get the adjacency matrix of G^2 given one for G, having spent one Boolean matrix multiplication and one matrix addition. Suppose we recursively compute APSP on G^2: how can we translate this result back to G? The next lemma shows that the shortest-path distances in G^2 are nicely related to those in G.

Lemma 4.5. *If d_{xy} and D_{xy} are the shortest-path distances between x, y in G and G^2 respectively, then*

$$D_{xy} = \left\lceil \frac{d_{xy}}{2} \right\rceil.$$

Proof. Any u-v path in G can be written as

$$u, a_1, b_1, a_2, b_2, \ldots, a_k, b_k, v$$

if the path has odd length; an even-length path can be written as

$$u, a_1, b_1, a_2, b_2, \ldots, a_k, b_k, a_{k+1}, v.$$

In either case, G^2 has edges $ub_1, b_1b_2, \ldots, b_{k-1}b_k, b_kv$, and thus a u-v path of length $\lceil \frac{d_{xy}}{2} \rceil$ in G^2. Therefore $D_{xy} \leq \lceil \frac{d_{xy}}{2} \rceil$.

To show equality, suppose there is a u-v path of length $\ell < \lceil \frac{d_{xy}}{2} \rceil$ in G^2. Each of these ℓ edges corresponds to either an edge or a 2-edge path in G, so we can find a u-v path of length at most $2\ell < d_{xy}$ in G, a contradiction. □

Lemma 4.5 implies that

$$d_{uv} \in \{2D_{uv}, 2D_{uv} - 1\}.$$

But which one? The following lemmas give us simple rule to decide. Let $N_G(v)$ denote the set of neighbors of v in G.

Lemma 4.6. *If $d_{uv} = 2D_{uv}$, then for all $w \in N_G(v)$ we have $D_{uw} \geq D_{uv}$.*

Proof. Assume not, and let $w \in N_G(v)$ be such that $D_{uw} < D_{uv}$. Since both of them are integers, we have $2D_{uw} < 2D_{uv} - 1$. Then the shortest u-w path in G along with the edge wv forms a u-v-path in G of length at most $2D_{uw} + 1 < 2D_{uv} = d_{uv}$, which is in contradiction with the assumption that d_{uv} is the shortest path in G. $\square$

Lemma 4.7. *If $d_{uv} = 2D_{uv} - 1$, then $D_{uw} \leq D_{uv}$ for all $w \in N_G(v)$; moreover, there exists $z \in N_G(v)$ such that $D_{uz} < D_{uv}$.*

Proof. For any $w \in N_G(v)$, considering the shortest u-v path in G along with the edge vw implies that $d_{uw} \leq d_{uv} + 1 = (2D_{uv} - 1) + 1$, so Lemma 4.5 gives that $D_{uw} = \lceil d_{uw}/2 \rceil = D_{uv}$. For the second claim, consider a vertex $z \in N_G(v)$ on a shortest path from u to v. Then $d_{uz} = d_{uv} - 1$, and Lemma 4.5 gives $D_{uz} < D_{uv}$. $\square$

Where did we use that G was undirected? In Lemma 4.6 we used that $w \in N_G(v) \implies wv \in E$. And in Lemma 4.7 we used that $w \in N_G(v) \implies vw \in E$.

These lemmas can be summarized thus:

$$d_{uv} = 2D_{uv} \iff \frac{\sum_{w \in N(v)} D_{uw}}{\deg(v)} \geq D_{uv}, \tag{4.1}$$

where $\deg(j) = |N_G(j)|$ is the degree of j. Given D, the criterion on the right can be checked for each uv in time $\deg(v)$ by just computing the average, but that could be too slow—how can we do better? Define the ***normalized adjacency matrix*** of G to be $\widehat{A}$ with

$$\widehat{A}_{wv} = \mathbb{1}_{wv \in E} \cdot \frac{1}{\deg(v)}.$$

Now if D is the distance matrix of G^2, then

$$(D\widehat{A})_{uv} = \sum_{w \in V} D_{uw}\widehat{A}_{wv} = \frac{\sum_{w \in N_G(v)} D_{uw}}{\deg(v)},$$

which is conveniently the expression in (4.1). Let $\mathbb{1}_{(D\widehat{A}<D)}$ be a matrix with the uv-entry being 1 if $(D\widehat{A})_{uv} < D_{uv}$, and zero otherwise. Then the distance matrix for G is

$$2D - \mathbb{1}_{(D\widehat{A}<D)}.$$

This completes the algorithm, which we now summarize:

Algorithm 6: Seidel's Algorithm

```
    Input: Unweighted undirected graph $G = (V, E)$ with
           adjacency matrix $A$
    Output: The distance matrix for $G$
6.1 if $A = J$ then
6.2     return $A$                // If $A$ is all-ones matrix, done!
6.3 else
6.4     $A' \leftarrow A * A + A$                  // Boolean operations
6.5     $D \leftarrow \text{Seidel}(A')$
6.6     return $2D - \mathbb{1}_{(D\widehat{A}<D)}$
```

Each call to the procedure above performs one Boolean matrix multiplication in step (6.4), one matrix multiplication with rational entries in step (6.6), plus $O(n^2)$ extra work. The diameter of the graph halves in each recursive call (by Lemma 4.5), and the algorithm hits the base case when the diameter is 1. Hence, the overall running time is $O(n^\omega \log n)$.

Ideas similar to these can be used to find shortest paths graphs with small integer weights on the edges: if the weights are integers in the interval $[0, W]$, Avi Shoshan and Uri Zwick [7] give an $\tilde{O}(Wn^\omega)$-time algorithm. In fact, Zwick [8] also extends the ideas to directed graphs, and gives an algorithm with runtime $\tilde{O}\left(W^{\frac{1}{4-\omega}} n^{2+\frac{1}{4-\omega}}\right)$.

7

8

4.4.1 *Finding the Shortest Paths*

How do we find the shortest paths themselves, and not just their lengths? For the previous algorithms, modifying the algorithms to output the paths is fairly simple. But for Seidel's algorithm, things get tricky. Indeed, since the runtime of Seidel's algorithm is strictly sub-cubic, how can we write down the shortest paths in n^ω time, since the total length of all these paths may be $\Omega(n^3)$? We don't: we just write down the ***successor pointers***. Indeed, for each pair u, v, define $S_v(u)$ to be the *second* node on a shortest u-v path (the first node being u, and the last being v). Then to get the entire u-v shortest path, we just follow these pointers:

$$u, S_v(u), S_v(S_v(u)), \ldots, v.$$

So there is a representation of all shortest paths that uses at most $O(n^2 \log n)$ bits.

The main idea for computing the successor matrix for Seidel's algorithm is to solve the ***Boolean Product Matrix Witness*** problem: given $n \times n$ Boolean matrices A, B, compute an $n \times n$ matrix W such that $W_{uv} = k$ if $A_{ik} = B_{kj} = 1$, and $W_{ij} = 0$ if no such k exists. We will hopefully see (and solve) this problem in a homework.

4.5 Optional: Fredman's Decision-Tree Complexity Bound

Given the algorithmic advances, one may wonder about lower bounds for the APSP problem. There is the obvious $\Omega(n^2)$ lower bound from the time required to write down the answer. Maybe even the decision-tree complexity of the problem is $\Omega(n^3)$? Then no algorithm can do any faster, and we'd have shown the Floyd-Warshall and the Matrix-Multiplication methods are optimal.

However, thanks to a result of Fredman [9], we know this is not the case. If we just care about the decision-tree complexity, we can get much better. Specifically, Fredman shows

9

Theorem 4.8. *The Min-Sum Product of two $n \times n$ matrices A, B can be deduced in $O(n^{2.5})$ additions and comparisons.*

Proof. The proof idea is to split A and B into rectangular sub-matrices, and compute the MSP on the sub-matrices. Since these sub-matrices are rectangular, we can substantially reduce the number of comparisons needed for each one. Once we have these sub-MSPs, we can simply compute an element-wise minimum for find the final MSP.

Fix a parameter W which we determine later. Then divide A into n/W $n \times W$ matrices $A_1, \dots, A_{n/W}$, and divide B into n/W $W \times n$ submatrices $B_1, \dots, B_{n/W}$. We will compute each $A_i \circledcirc B_i$. Now consider $(A \circledcirc B)_{ij} = \min_{k \in [W]}(A_{ik} + B_{kj}) = \min_{k \in [W]}(A_{ik} + B^T_{jk})$ and let k^* be the minimizer of this expression. Then we have the following:

$$A_{ik^*} - B^T_{jk^*} \leq A_{ik} - B^T_{jk} \ \forall k \tag{4.2}$$

$$A_{ik^*} - A_{ik} \leq -(B^T_{jk^*} - B^T_{jk}) \ \forall k \tag{4.3}$$

Now for every pair of columns, p, q from A_i, B_i^T, and sort the following $2n$ numbers

$$A_{1p} - A_{iq}, A_{2p} - A_{2q}, \dots, A_{np} - A_{nq}, -(B_{1p} - B_{1q}), \dots, -(B_{np} - B_{nq})$$

We claim that by sorting W^2 lists of numbers we can compute $A_i \circledcirc B_i$. To see this, consider a particular entry $(A \circledcirc B)_{ij}$ and find a k^* such that for every $k \in [W]$, $A_{ik^*} - A_{ik}$ precedes every $-(B^T_{jk^*} - B^T_{jk})$ in their sorted list. By (4.3), such a k^* is a minimizer. Then we can set $(A \circledcirc B)_{ij} = A_{ik^*} + B_{k^*j}$.

This computes the MSP for A_i, B_i, but it is possible that another $A_j \circledcirc B_j$ produces the actual minimum. So, we must take the element-wise minimum across all the $(A_i \circledcirc B_i)$. This produces the MSP of A, B.

Now for the number of comparisons. We have n/W smaller products to compute. Each sub-product has W^2 arrays to sort, each of which can be sorted in $2n \log n$ comparisons. Finding the minimizer requires $W^2 n$ comparisons.So, computing the sub-products requires $n/W * 2W^2 n \log n = 2n^2 W \log n$ comparisons. Then, reconstructing the final MSP requires n^2 element-wise minimums between $n/W - 1$ elements, which requires n^3/W comparisons. Summing these bounds gives us $n^3/W + 2n^2 W \log n$ comparisons. Optimizing over W gives us $O(n^2 \sqrt{n \log n})$ comparisons. $\square$

This result does not give us a fast algorithm, since it just counts the number of comparisons, and not the actual time to figure out which comparisons to make. Regardless, many of the algorithms that achieve $n^3/\operatorname{poly}\log n$ time for APSP use Fredman's result on tiny instances (say of size $O(\operatorname{poly}\log n)$, so that we can find the best decision-tree using brute-force) to achieve their results.

5 Low-Stretch Spanning Trees

Given that shortest paths from a single source node s can be represented by a single shortest-path tree, can we get an analog for all-pairs shortest paths? Given a graph can we find a tree T that gives us the shortest-path distances between every pair of nodes? Does such a tree even exist? Sadly, the answer is negative—and it remains negative even if we allow this tree to stretch distances by a small factor, as we will soon see. However, we show that allowing randomization will allow us to circumvent the problems, and get low-stretch spanning trees in general graphs.

In this chapter, we consider *undirected* graphs $G = (V, E)$, where each edge e has a *non-negative* weight/length w_e. For all u, v in V, let $d_G(u, v)$ be the *distance* between u, v, i.e., the length of a shortest path in G from u to v. Observe that the set V along with the distance function d_G forms a metric space.

A metric space is a set V with a distance function d satisfying *symmetry* (i.e., $d(x, y) = d(y, x)$ for all $x, y \in V$) and the *triangle inequality* ($d(x, y) \leq d(x, z) + d(z, y)$ for all $x, y, z \in V$). Typically, the definition also asks for $x = y \iff d(x, y) = 0$, but we will merely assume $d(x, x) = 0$ for all x.

5.1 Towards a Definition

The study of low-stretch spanning trees is guided by two high level hopes:

1. Graphs have spanning trees that preserve their distances. That is, given G there exists a subtree $T = (V, E_T)$ with $E_T \subseteq E$ such that

$$d_G(u, v) \approx d_T(u, v) \qquad \text{for all } u, v \in V.$$

We assume that the weights of edges in E_T are the same as those in G.

2. Many **NP**-hard problems are much easier to solve on trees.

Supposing these are true, we have a natural recipe for designing algorithms to solve problems that depend only on distances in G: (1) find a spanning tree T preserving distances in G, (2) solve the problem on T, and then (3) return the solution (or some close cousin) with the hope that it is a good solution for the original graph.

5.1.1 An All-Pairs Shortest Path Tree?

The boldest hope would be to find an *all-pairs shortest path tree* T, i.e., one that ensures $d_T(u,v) = d_G(u,v)$ for all u,v in V. However, such a tree may not exist: consider K_n, the clique of n nodes, with unit edge lengths. The distance d_G satisfies $d_G(x,y) = 1$ for all $x \neq y$, and zero otherwise. But any subtree T contains only $n-1$ edges, so most pairs of vertices $x,y \in V$ lack an edge between them in T. Any such pair has a shortest-path distance $d_T(x,y) \geq 2$, whereas $d_G(x,y) = 1$.

5.1.2 A First Relaxation: Low-Stretch Spanning Trees

To remedy the snag above, let us not require distances in T be equal to those in G, but instead be within a small multiplicative factor $\alpha \geq 1$ of those in G.

Definition 5.1. Let T be a spanning tree of G, and let $\alpha \geq 1$. We call T a (deterministic) *α-stretch spanning tree* of G if

Exercise: show that if T is any subtree of G with the same edge weights, then $d_G(x,y) \leq d_T(x,y)$.

$$d_G(u,v) \leq d_T(u,v) \leq \alpha\, d_G(u,v).$$

holds for all $u,v \in V$.

Supposing we had such a low-stretch spanning tree, we could try our meta-algorithm out on the ***traveling salesperson problem (TSP)***: given a graph, find a closed tour that visits all the vertices, and has the smallest total length. This problem is **NP**-hard in general, but let us see how an α-stretch spanning tree of G gives us an an α-approximate TSP solution for G. The algorithm is simple:

Algorithm 7: TSP via Low-Stretch Spanning Trees

```
7.1 Find an $\alpha$-stretch spanning tree $T$ of $G$.
7.2 Solve TSP on $T$ to get an ordering $\pi_T$ on the vertices.
7.3 return the ordering $\pi_T$.
```

Solving the TSP problem on a tree T is trivial: just take an Euler tour of T, and let π_T be the order in which the vertices are visited. Let us bound the quality of this solution.

Claim 5.2. π_T is an α-approximate solution to the TSP problem on G.

Proof. Suppose that the permutation π_G minimizes the length of the TSP tour for G. The length of the resulting tour is

$$OPT_G := \sum_{i\in[n]} d_G(\pi_G(i), \pi_G(i+1)).$$

Since distances in the tree T are stretched by only a factor of α,

$$\sum_{i\in[n]} d_T(\pi_G(i),\pi_G(i+1)) \leq \alpha \cdot \sum_{i\in[n]} d_G(\pi_G(i),\pi_G(i+1)). \tag{5.1}$$

Now, since π_T is the optimal ordering for the tree T, and π_G is some other ordering,

$$\underbrace{\sum_{i\in[n]} d_T(\pi_T(i),\pi_T(i+1))}_{OPT_T} \leq \sum_{i\in[n]} d_T(\pi_G(i),\pi_G(i+1)). \tag{5.2}$$

Finally, since distances were only stretched in going from G to T,

$$\sum_{i\in[n]} d_G(\pi_T(i),\pi_T(i+1)) \leq \sum_{i\in[n]} d_T(\pi_T(i),\pi_T(i+1)). \tag{5.3}$$

Putting it all together, the length of the tour given by π_T is

$$\sum_{i\in[n]} d_G(\pi_T(i),\pi_T(i+1)) \leq \alpha \cdot \sum_{i\in[n]} d_G(\pi_G(i),\pi_G(i+1)),$$

which is $\alpha \cdot OPT_G$. □

Hence, if we had low-stretch spanning trees T with $\alpha \leq 1.49$, we would get the best approximation algorithm for the TSP problem. (Assuming we can find T, but we defer this for now.) However, you may have already noticed that the K_n example above shows that $\alpha < 2$ is impossible. But can we achieve $\alpha = 2$? Indeed, is there any "small" value for α such that for any graph G we can find an α-stretch spanning tree of G?

Sadly, things are terrible: take the cycle C_n, again with unit edge weights. Now any subtree T is missing one edge from C_n, say uv. The endpoints of this edge are at distance 1 in C_n, but $d_T(u,v) = n-1$, since we have to go all the way around the cycle. Hence, getting $\alpha < (n-1)$ is impossible in general.

Exercise: show how to find, for any graph G, a spanning tree T with stretch $\alpha \leq n-1$.

5.1.3 *A Second Relaxation: Randomization to the Rescue*

Since we cannot get trees with small stretch deterministically, let us try to get trees with small stretch "on average". We amend our definition as follows:

Definition 5.3. A ***(randomized) low-stretch spanning tree*** of *stretch α* for a graph $G = (V,E)$ is a probability distribution $\mathcal{D}$ over spanning trees of G such that for all $u,v \in V$, we have

Henceforth, all references to low-stretch trees will only refer to this randomized version, unless otherwise specified.

$$\begin{aligned} d_G(u,v) &\leq d_T(u,v) \qquad \text{for all } T \text{ in the support of } \mathcal{D}\text{, and} \\ \mathbb{E}_{T\sim\mathcal{D}}[d_T(u,v)] &\leq \alpha\, d_G(u,v) \end{aligned} \tag{5.4}$$

Observe that the first property must hold with probability 1 (i.e., it holds for all trees in the support of the distribution), whereas the second property holds only on average. Is this definition any good for our TSP example above? If we change the algorithm to sample a tree T from the distribution and then return the optimal tour for T, we get a randomized algorithm that is good in expectation. Indeed, (5.1) becomes

$$\sum_{i\in[n]} \mathbb{E}[d_T(\pi_G(i),\pi_G(i+1))] \leq \alpha \cdot \sum_{i\in[n]} d_G(\pi_G(i),\pi_G(i+1)), \qquad (5.5)$$

because the stretch guarantees hold in expectation (and linearity of expectation). The rest of the inequalities hold unchanged, including (5.3)—which requires the probability 1 guarantee of Definition 5.6 (Do you see why?). Hence, we get

$$\underbrace{\sum_{i\in[n]} \mathbb{E}[d_G(\pi_T(i),\pi_T(i+1))]}_{\text{expected algorithm's tour length}} \leq \alpha \cdot \underbrace{\sum_{i\in[n]} d_G(\pi_G(i),\pi_G(i+1))}_{OPT_G}. \qquad (5.6)$$

Even a randomized better-than-1.49 approximation for TSP would still be amazing! And the algorithmic template here works not just for TSP: any **NP**-hard problem whose objective is a linear function of distances (e.g., many other vehicle routing problems, or the k-median clustering problem) can be solved in this way. Indeed, the first approximation algorithms for many such problems came via low-stretch spanning trees.

Moreover, (randomized) low-stretch spanning trees arise in many different contexts, some of which are not obvious at all. E.g., they can be used to more efficiently solve "Laplacian" linear systems of the form $A\vec{x} = \vec{b}$, where A is the ***Laplacian matrix*** of some graph G. To do this, we let P be the Laplacian matrix of a low-stretch spanning tree of G, and then we solve the system $P^{-1}A\vec{x} = P^{-1}\vec{x}$ instead. This is called *preconditioning* with P. It turns out that this preconditioning allows certain algorithms for solving linear systems to converge faster to a solution. Time permitting, we will discuss this application later in the course.

5.2 *Low-Stretch Spanning Tree Construction*

But first, given a graph G, how can we find a randomized low-stretch spanning tree for G with a small value of α (and efficiently)? As a sanity check, let us check what we can do on the two examples from before:

1. For the complete graph K_n, choose a star graph centered at a uniformly random vertex of G. For any pair of vertices u, v, they are

A natural first attempt (at least for unweighted graphs) would be to try a uniformly random spanning tree. This does not work very well (which I think is not that surprising), even for the complete graph K_n (which I think is somewhat surprising). A result of Moon and Moser shows that for any pair of vertices $u, vinV(K_n)$, if we choose T to be one of the n^{n-2} spanning trees uniformly at random, the expected distance is $d_T(u,v) = \Theta(\sqrt{n})$.

at distance 1 in this star if either u or v is the center, else they are at distance 2. Hence the expected distance is $\frac{2}{n} \cdot 1 + \frac{n-2}{n} \cdot 2 = 2 - \frac{2}{n}$.

2. For the cycle C_n, choose a tree by dropping a single edge uniformly at random. For any edge uv in the cycle, there is only a 1 in n chance of deleting the edge from u to v. But when it is deleted, u and v are at distance $n-1$ in the tree. So

$$\mathbb{E}[d_T(u,v)] = \frac{n-1}{n} \cdot 1 + \frac{1}{n} \cdot (n-1) = 2 - \frac{2}{n}.$$

And what about an arbitrary pair of nodes u, v in C_n? We can use the exercise on the right to show that the stretch on other pairs is no worse!

Exercise: Given a graph G, suppose the stretch on all edges is at most α. Show that the stretch on all pairs of nodes is at most α. (Hint: linearity of expectation.)

While we will not manage to get $\alpha < 1.49$ for general graphs (or even for the above examples, for which the bounds of $2 - \frac{2}{n}$ are the best possible), we show that $\alpha \approx O(\log n)$ can indeed be achieved. The following theorem is the current best result, due to Ittai Abraham and Ofer Neiman:

Theorem 5.4. *For any graph G, there exists a distribution $\mathcal{D}$ over spanning trees of G with stretch $\alpha = O(\log n \log\log n)$. Moreover, the construction is efficient: we can sample trees from this distribution $\mathcal{D}$ in $O(m \log n \log\log n)$ time.*

Moreover, the stretch bound of this theorem is almost optimal, up to the $O(\log\log n)$ factor, as the following lower bound due to Alon, Peleg, Karp, and West shows.

Theorem 5.5. *For infinitely many n, there exist graphs G on n vertices such that any α-stretch spanning tree distribution $\mathcal{D}$ on G must have $\alpha = \Omega(\log n)$. In fact, G can be taken to be the n-vertex square grid, the n-vertex hypercube, or any n-vertex constant-degree expander.*

5.3 Bartal's Construction

The algorithm underlying Theorem 5.4 is quite involved, but we can give the entire construction of low-stretch trees for finite metric spaces.

Definition 5.6. A (randomized) low-stretch tree with stretch α for a metric space $M = (V, d)$ is a probability distribution $\mathcal{D}$ over trees over the vertex set V such that for all $u, v \in V$, we have

$$\begin{aligned} d(u,v) &\le d_T(u,v) \qquad \text{for all } T \text{ in the support of } \mathcal{D}\text{, and} \\ \mathbb{E}_{T\sim\mathcal{D}}[d_T(u,v)] &\le \alpha\, d(u,v). \end{aligned} \tag{5.7}$$

The difference of this definition from Definition 5.6 is slight: we now have a metric space instead of a graph, and we are allowed to output any tree on the vertex set V (since the concept of subtrees doesn't make sense now). Note that given a graph G, we can compute its shortest-path metric (V, d_G) and then find a distribution over (non-spanning) trees that approximate the distance in G. So if we don't really need the spanning aspect in our low-stretch trees—e.g., as in the TSP example—we can use results for this definition.

We need one more piece of notation: for a metric space $M = (V, d)$, define its ***aspect ratio*** Δ to be

$$\Delta_M := \frac{\max_{u \neq v \in V} d(u,v)}{\min_{u \neq v \in V} d(u,v)}.$$

We will show the following theorem, due to Yair Bartal:

Theorem 5.7. *For any metric space $M = (V, d)$, there exists an efficiently sampleable α_B-stretch spanning tree distribution $\mathcal{D}_B$, where*

$$\alpha_B = O(\log n \log \Delta_M).$$

The proof works in two parts: we first show a good *low-diameter decomposition*. This will be a procedure that takes a metric space and a diameter bound D, and randomly partitions the metric space into clusters of diameter $\leq D$, in such a way that close-by points are unlikely to be separated. Then we show how such a low-diameter decomposition can be used recursively to constuct a low-stretch tree.

The ***diameter*** of a set S is $\max_{u,v \in S} d(u,v)$, i.e., the maximum distance between any two points in it.

5.3.1 Low-Diameter Decompositions

The notion of a low-diameter decomposition has become ubiquitous in algorithm design, popping up in approximation and online algorithms, and also in distributed and parallel algorithms. It's something worth understanding well.

Definition 5.8 (Low-Diameter Decomposition)**.** A ***low-diameter decomposition scheme*** (or LDD scheme) with parameter β for a metric $M = (V, d)$ is a randomized algorithm that, given a bound $D > 0$, partitions the point set V into "clusters" $C_1, \ldots, C_t$ such that

(i) for all $i \in \{1, \ldots, t\}$, the *diameter* of C_i is at most D, and
(ii) for all $x, y \in V$ such that $x \neq y$, we have

$$\Pr[x, y \text{ in different clusters}] \leq \beta \cdot \frac{d(x,y)}{D}.$$

Let's see a few examples, to get a better sense for the definition:

1. Consider a set of points on the real line. One way to partition the line into pieces of diameter D is simple: imagine making notches

on the line at distance D from each other, and then randomly shifting them. Formally, pick a random value $R \in [0, D]$ uniformly at random, and partition the line into intervals of the form $[Di + R, D(i+1) + R)$, for $i \in \mathbb{Z}$. A little thought shows that points x, y are separated with probability exactly $\frac{d(x,y)}{D}$.

2. The infinite 2-dimensional square grid with unit edge-lengths. One way to divide this up is to draw horizontal and vertical lines which are $D/2$ apart, and randomly shift as above. A pair x, y is separated with probability exactly $\frac{d(x,y)}{D/2}$ in this case. Indeed, this approach works for k-dimensional hypergrids (and k-dimensional ℓ_1-space) with probability $k \cdot \frac{d(x,y)}{D}$ — in this case the β parameter is at most the dimension of the space.

3. What about lower bounds? One can show that for the k-dimensional hypergrid, we cannot get $\beta = o(k)$. Or for a constant-degree n-vertex expander, we cannot get $\beta = o(\log n)$. Details to come soon.

Since the aspect ratio of the metric space is invariant to scaling all the edge lengths by the same factor, it will be convenient to assume that the smallest non-zero distance in d is 1, so the largest distance is Δ. The basic algorithm is then quite simple:

```
Algorithm 8: LDD($M = (V, d), D$)
8.1 $p \leftarrow \min(1, \frac{4 \log n}{D})$.
8.2 while there exist unmarked point do
8.3     $v \leftarrow$ any unmarked point.
8.4     sample $R_v \sim \text{Geometric}(p)$.
8.5     cluster $C_v \leftarrow \{\text{unmarked } u \mid d(v,u) < R_v\}$.
8.6     mark points in $C_v$.
8.7 return the resulting set of clusters.
```

Lemma 5.9. *The algorithm above ensures that*

1. *the diameter of every cluster is at most D with probability at least $1 - 1/n$, and*

2. *any pair $x, y \in V$ is separated with probability at most $2p\, d(x,y)$.*

Proof. To show the diameter bound, it suffices to show that $R_v \leq D/2$ for each cluster C_v, because then the triangle inequality shows that for any $x, y \in C_v$,

$$d(x,y) \leq d(x,v) + d(v,y) < D/2 + D/2 = D.$$

Now the probability that $R_v > D/2$ for one particular cluster is

We use that $1 - z \leq e^z$ for all $z \in \mathbb{R}$.

$$\Pr[R_v > D/2] = (1-p)^{D/2} \le e^{-pD/2} \le e^{-2\log n} = \frac{1}{n^2}.$$

By a union bound, there exists a cluster with diameter $> D$ with probability

$$1 - \Pr[\exists v \in V,\, R_v > D/2] \ge 1 - \frac{n}{n^2} = 1 - \frac{1}{n}.$$

To bound the probability of some pair u, v being separated, we use the fact that sampling from the geometric distribution with parameter p means repeatedly flipping a coin with bias p and counting the number of flips until we see the first heads. Recall this process is memoryless, meaning that even if we have already performed k flips without having seen a heads, the time until the first heads is still geometrically distributed.

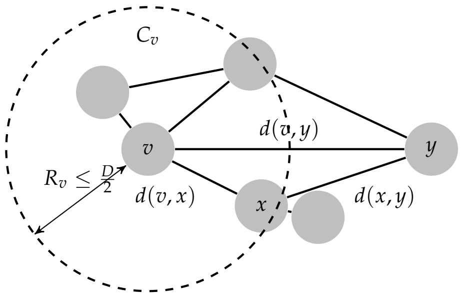


Figure 5.1: A cluster forming around v in the LDD process, separating x and y. To reduce clutter, only some of the distances are shown.

Hence, the steps of drawing R_v and then forming the cluster can be viewed as starting from v, where the cluster is a unit-radius ball around v. Each time we flip a coin of bias p: it is comes up heads we set the radius R_v to the current value, form the cluster C_v (and mark its vertices) and then pick a new unmarked point v; on seeing tails, we just increment the radius of v's cluster by one and flip again. The process ends when all vertices lie in some cluster.

For x, y, consider the first time when one of these vertices lies inside the current ball centered at some point, say, v. (This must happen at some point, since all vertices are eventually marked.) Without loss of generality, let the point inside the current ball be x. At this point, we have performed $d(v,x)$ flips without having seen a heads. Now we will separate x, y if we see a heads within the next $\lceil d(v,y) - d(v,x) \rceil \le \lceil d(x,y) \rceil$ flips—beyond that, both x, y will have been contained in v's cluster and hence cannot be separated. But the probability of getting a heads among these flips is at most (by a union bound)

$$\lceil d(x,y) \rceil\, p \le 2d(x,y)\, p \le 8 \log n\, \frac{d(x,y)}{D}.$$

(Here we used that the minimum distance is 1, so rounding up distances at most doubles things.) This proves the claimed probability of separation. $\square$

Recall that we wanted the diameter bound with probability 1, whereas Lemma 5.9 only ensures it with high probability. Here's a quick fix to this problem: repeat the above process until the returned partition has clusters of diameter at most D. The probability of any pair u, v being separated by this last run of Algorithm 8 is at most the probability of u, v being separated by any of the runs, which is at most $p\, d(u,v)$ times the expected number of runs,

$$p\, d(u,v) \cdot (1/(1-1/n)) \le 2p\, d(u,v) = O(\log n)\, \frac{d(u,v)}{D}.$$

Lemma 5.10. *The low-diameter decomposition scheme above achieves parameter $\beta = O(\log n)$ for any metric M on n points.*

5.3.2 *Low-Stretch Trees Using LDDs*

Now we can use the low-diameter decomposition scheme to get a low-stretch tree (LST). Here's the high-level idea: given a metric with diameter Δ, use an LDD to decompose it into clusters with diameter $D \leq \Delta/2$. Build a tree recursively for each of these clusters, and then combine these trees into one tree for the entire metric.

Recall we assumed that the metric had minimum distance 1 and maximum distance Δ. Formally, we invoke the procedure LST below with the parameters LST(metric M, $\lceil \log_2 \Delta \rceil$).

Algorithm 9: LST(metric M = (V,d), $D = 2^\delta$)

```
    Input: Invariant: diameter$(M) \leq 2^\delta$
9.1 if $|V| = 1$ then
9.2     return tree containing the single point in $V$.
9.3 $C_1, \dots, C_t \leftarrow$ LDD$(M, D = 2^{\delta-1})$.
9.4 for $j$ in $\{1, \dots, t\}$ do
9.5     $M_j \leftarrow$ metric $M$ restricted to the points in $C_j$.
9.6     $T_j \leftarrow$ LST$(M_j, \delta - 1)$.
9.7 Add edges of length $2^\delta$ from root $r_1$ for tree $T_1$ to the roots of
      $T_2, \dots, T_t$.
9.8 return resulting tree rooted at $r_1$.
```

We are ready to prove Theorem 5.7; we will show that the tree has expected stretch $O(\beta \log \Delta)$, and that it does not shrink any distances. In fact, we show a slightly stronger guarantee.

Lemma 5.11. *If the random tree T returned by some call LDD(M', δ) has root r, then (a) every vertex x in T has distance $d(x, r) \leq 2^{\delta+1}$, and (b) the expected distance between any $x, y \in T$ has $\mathbb{E}[d_T(x, y)] \leq 8\delta\beta\, d(x, y)$.*

Proof. The proof is by induction on δ. For the base case, the tree has a single vertex, so the claims are trivial. Else, let x lie in cluster C_j, so inductively the distance to the root of the tree T_i is $d(x, r_i) \leq 2^{(\delta-1)+1}$. Now the distance to the new root r is at most 2^δ more, which gives $2^\delta + 2^\delta = 2^{\delta+1}$ as claimed.

Moreover, any pair x, y is separated by the LDD with probability $\beta\, \frac{d(x,y)}{2^{\delta-1}}$, in which case their distance is at most

$$d(x, r) + d(r, y) \leq 2^{\delta+1} + 2^{\delta+1} = 4 \cdot 2^\delta.$$

Else they lie in the same cluster, and inductively have expected distance at most $8(\delta-1)\beta\, d(x,y)$. Hence the expected distance is

$$\begin{aligned}\mathbb{E}[d(x,y)] &\le \Pr[x,y \text{ separated}]\cdot 4\cdot 2^{\delta} + \\ &\quad \Pr[x,y \text{ not separated}]\cdot 8(\delta-1)\beta\, d(x,y) \\ &\le \beta\,\frac{d(x,y)}{2^{\delta-1}}\cdot 4\, 2^{\delta} + 8(\delta-1)\beta\, d(x,y) \\ &= 8\,\delta\,\beta\, d(x,y). \quad \square\end{aligned}$$

This proves Theorem 5.7 because $\beta = O(\log n)$, and the iniitial call on the entire metric defines $\delta = O(\log \Delta)$. In fact, if we have a better LDD (with smaller β), we immediately get a better low-stretch tree. For example, shortest-path metrics of planar graphs admit an LDD with parameter $\beta = O(1)$; this shows that planar metrics admit (randomized) low-stretch trees with stretch $O(\log \Delta)$.

It turns out this factor of $O(\log n \log \Delta)$ can be improved to $O(\log n)$—this was done by Fakcharoenphol, Rao, and Talwar. Moreover, the bound of $O(\log n)$ is tight: the lower bounds of Theorem 5.5 continue to hold even for low-stretch non-spanning trees.

5.4 *Metric Embeddings: a.k.a. Simplifying Metrics*

We just how to approximate a finite metric space with a simpler metric space, defined over a tree. (Loosely, "every metric space is within $O(\log n)$ of some tree metric".) And since trees are simpler metrics, both conceptually and algorithmically, such an embedding can help design algorithms for problems on metric spaces.

This idea of approximating metric spaces by simpler ones has been extensively studied in various forms. For example, another famous result of Jean Bourgain (with an extension by Jirka Matoušek) shows that any finite metric space on n points can be embedded into ℓ_p-space with $O((\log n)/p)$ distortion [1]. Moreover, the ***Johnson-Lindenstrauss Lemma***, which we will see in a future chapter, shows that any n point-submetric of Euclidean space can be embedded into a (low-dimensional) Euclidean space of dimension at most $O(\log n/\epsilon^2)$, such that distances between points are distorted by a factor of at most $1\pm\epsilon$ [2]. Since geometric spaces, and particularly, low-dimensional Euclidean spaces, are easier to work with and reason about, these can be used for algorithm design as well.

[1]

[2]

5.4.1 *Historical Notes*

To be cleaned up. Elkin et al. [3] gave the first polylog-stretch *spanning* trees, which took eight years following Bartal's construction. (The first low-stretch spanning trees had stretch $2^{O(\sqrt{\log n \log\log n})}$ by Alon et al. [4], which is smaller than n^{ϵ} for any $\epsilon > 0$ but larger than

[3]

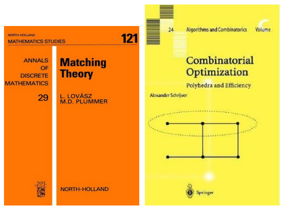
[4]

polylogarithmic, i.e., $(\log n)^C$ for any $C > 0$.)

6 Graph Matchings I: Combinatorial Algorithms

Another fundamental graph problem is to find ***matchings***: these are subsets of edges that do not share endpoints. Matchings arise in various contexts: matching tasks to workers, or advertisements to slots, or roommates to each other. Moreover, matchings have a rich combinatorial structure. The classical results can be found in *Matching Theory* by Laci Lovász and Michael Plummer [1], though Lex Schrijver's *Combinatorial Optimization: Polyhedra and Efficiency* might be easier to find, and contains more recent developments as well.

[1]



Several different and interesting algortihmic techniques can be used to find large matchings in graphs; we will discuss them over the next few chapters. This chapter discusses the simplest, combinatorial algorithms.

6.1 Notation and Definitions

Consider an undirected (simple and connected) graph $G = (V, E)$ with $|V| = n$ and $|E| = m$ as usual. The graph is unweighted; we will consider weighted versions of matching problems in later chapters. When considering ***bipartite*** graphs, where the vertex set has parts $V = L \uplus R$ (the "left" and "right", and the edges $E \subseteq L \times R$, we may denote the graph as $G = (L, R, E)$.

Definition 6.1 (Matching)**.** A matching in graph G is a subset of the edges $M \subseteq E$ which have no endpoints in common. Equivalently, the edges in M are disjoint, and hence every vertex in (V, M) has maximum degree 1.

Given a matching M in G, a vertex v is *open* or *exposed* or *free* if no edge in the matching is incident to v, else the vertex is *closed* or *covered* or *matched*. Observe: the empty set of edges is a matching. Moreover, any matching can have at most $|V|/2$ edges, since each edge covers two vertices, and each vertex can be covered by at most one edge.

Definition 6.2 (Perfect Matching)**.** A ***perfect matching*** M is a matching such that $|M| = |V|/2$. Equivalently, every vertex is matched in the matching M.

Definition 6.3 (Maximum Matching)**.** A ***maximum cardinality matching*** (or simply maximum matching) in G is a matching with largest possible cardinality. The size of the maximum matching in graph G is denoted $MM(G)$.

Definition 6.4 (Maximal Matching)**.** A ***maximal matching*** on a graph is a matching that is inclusion-wise maximal; that is, no additional edges can be added to M while maintaining the matching propert. Hence, $M \cup \{e\}$ is not a matching for all edges $e \notin M$.

The last definition is given to mention something we will *not* be focusing on; our interest is in perfect and maximum matchings. That being said, it is a useful exercise to show that any maximal matching in G has at least $MM(G)/2$ edges.

6.1.1 *Augmenting Paths for Matchings*

Since we want to find a maximum matching, a question we may ask is: *given a matching M, can we (efficiently) decide if it is a maximum matching?* One answer to this was suggested by Berge, who gave a characterization of maximum matchings in terms of "augmenting" paths.

Definition 6.5 (Alternating Path)**.** For matching M, an ***M-alternating path*** is a path in which edges in M alternate with those not in M.

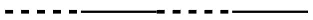


Figure 6.1: An alternating path P (dashed edges are not in P, solid edges are in P)

Definition 6.6 (Augmenting Path)**.** For matching M, an ***M-augmenting path*** is an M-alternating path with both endpoints open.

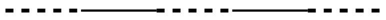


Figure 6.2: An augmenting path

Berge (1957)

Given sets S, T, their symmetric difference is denoted

$$S \triangle T := (S \setminus T) \cup (T \setminus S).$$

The following theorem explains the name for augmenting paths.

Theorem 6.7 (Berge's Optimality Criterion)**.** *A matching M is a maximum matching in graph G if and only if there are no M-augmenting paths in G.*

Proof. If there is an M-augmenting path P, then $M' := M \triangle P$ is a larger matching than M. (Think of getting M' by toggling the dashed edges in the path to solid, and *vice versa*). Hence if M is maximum matching, there cannot exist an M-augmenting path.

Conversely, suppose M is not a maximum matching, and matching M' has $|M'| > |M|$. Consider their symmetric difference $S := M \triangle M'$.

Every vertex is incident to at most 2 edges in S (at most one each from M and M'), so S consists of only paths and cycles, all of them having edges in M alternating with edges in M'. Any cycle with this alternating structure must be of even length, and any path has at most one more edge from one matching than form the other. Since $|M'| > |M|$, there must exists a path in S with one more edge from M' than from M. But this is an M-augmenting path. $\square$

If we could efficiently find an M-augmenting path (if one exists), we could repeatedly augment the current matching until we have a maximum matching. However, Berge's theorem does not immediately give an efficient algorithm: finding an M-augmenting path could naively take exponential time. We now give algorithms to efficiently find augmenting paths, first in bipartite graphs, and then in general graphs.

6.2 *Bipartite Graphs*

Finding an M-augmenting path (if one exists) in bipartite graphs is an easier task, though it still requires cleverness. A first step is to consider a "dual" object, which is called a vertex cover.

Definition 6.8 (Vertex Cover)**.** A ***vertex cover*** in G is a set of vertices C such that every edge in the graph has at least one endpoint in C.

Note that the entire set V is trivially a vertex cover, and the challenge is to find small vertex covers. We denote the size of the smallest cardinality vertex cover of graph G as $VC(G)$. Our motivation for calling it a "dual" object comes from the following fundamental theorem from the early 20th century:

Theorem 6.9 (König's Minimax Theorem)**.** *In a bipartite graph, the size of the largest possible matching equals the cardinality of the smallest vertex cover:*

$$MM(G) = VC(G).$$

Dénes König (1916)

This theorem is a special case of the max-flow min-cut theorem, which you may have seen before. It is first of many min-max relationships, many of which lead to efficient algorithms. Indeed, the algorithm for finding augmenting paths will come out of the proof of this theorem.

Exercise: Use König's theorem to prove P. Hall's theorem: *A bipartite graph has a matching that matches all vertices of L if and only for every subset $S \subseteq L$ of vertices,* $|N(S)| \geq |S|$. Here $N(S)$ denotes the "neighborhood" of S, i.e., those vertices with a neighbor inside S.

Proof. In many such proofs, there is one easy direction. Here, it is proving that $MM(G) \leq VC(G)$. Indeed, the edges of any matching share no endpoints, so covering a matching of size $MM(G)$ requires at least as many vertices. The minimum vertex cover size is therefore at least $MM(G)$.

Next, we prove that $MM(G) \geq VC(G)$. To do this, we give a linear-time algorithm that takes as input an arbitrary matching M, and either returns an M-augmenting path (if such a path exists), or else returns a vertex cover of size $|M|$. Since a maximum matching M admits no M-augmenting path by Berge's theorem, we would get back a vertex cover of size $MM(G)$, thereby showing $VC(G) \leq MM(G)$.

The proof is an "alternating" breadth-first search: it starts with all open nodes among the left vertex set L, and places them at level 0. Then it finds all the (new) neighbors of these nodes reachable using non-matching edges, and then all (new) neighbors of those nodes using matching edges, and so on. Formally, the algorithm is as follows, where we use $X_{\leq j}$ to denote $X_0 \cup \ldots \cup X_j$.

```
9.1 $X_0 \leftarrow$ all open vertices in $L$
9.2 for $i$ = 0, 1, 2, . . . do
9.3     $X_{2i+1} \leftarrow \{v \mid$ exists $u \in X_{2i}$ s.t. $uv \notin M$, and $v \notin X_{\leq 2i}\}$
9.4     $X_{2i+2} \leftarrow \{v \mid$ exists $u \in X_{2i+1}$ s.t. $uv \in M$, and $v \notin X_{\leq 2i+1}\}$
```

Let us make a few observations about the procedure. First, since the graph is bipartite, X_i is a subset of L for even levels i, and of R for odd levels i. Next, all vertices in $X_2 \cup X_4 \cup \ldots$ are matched vertices, since they are reached from the previous level using an edge in the matching. Moreover, if some odd level X_{2i+1} contains an open node v, we have found an M-alternating path from an open node in X_0 to v, and hence we can stop and return this augmenting path.

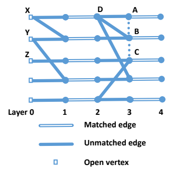


Figure 6.3: Illustration of the process to find augmenting paths in a bipartite graph. Mistakes here, to be fixed!

Hence, suppose we do not find an open node in an even level, and stop when some X_j is empty. Let $X = \cup_j X_j$ be all nodes added to any of the sets X_j; we call these ***marked*** nodes. Define the set C to be the vertices on the left which are *not marked*, plus the vertices on the right which are ***marked***. That is,

$$C := (L \setminus X) \cup (R \cap X)$$

We claim that C is a vertex cover of size $|M|$.

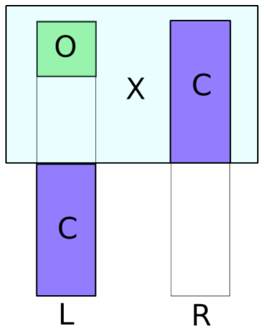


Figure 6.4: X = set of marked vertices, O = marked open vertices, C = claimed vertex cover of G. To be changed.

Claim 6.10. C is a vertex cover.

Proof. G is a bipartite graph, and C hits all edges that touch $R \cap X$ and $L \setminus X$. Hence we must show there are no edges between $L \cap X$ and $R \setminus X$, i.e., between the top-left and bottom-right of the figure.

1. There can be no unmatched edge from the *open* vertices in $L \cap X$ to $R \setminus X$, else that vertex would be reachable from X_0 and so belong to X_1. Moreover, an open vertex has no unmatched edges, by definition. Hence, any "offending edges" out of $L \cap X$ must come from a covered vertex.

2. There can be no non-matching edge from a covered vertex in $L \cap X$ to some node u in $R \setminus X$, else this node u would have been added to some level X_{2i+1}.

3. Finally, there can be no matching edge between a covered vertex in $L \cap X$ and some vertex in $R \setminus X$. Indeed, every covered node in $L \cap X$ (i.e., those in $X_2, X_4, \ldots$) was reached via a matching edge from some node in $R \cap X$. There cannot be another matching edge from some node in $R \setminus X$ incident to it.

This shows that C is a vertex cover. $\square$

Claim 6.11. $|C| \leq |M|$.

We use a simple counting argument:

- Every vertex in $R \cap X$ has a matching edge incident to it; else it would be open, giving an augmenting path.
- Every vertex in $L \setminus X$ has an incident edge in the matching, since no vertices in $L \setminus X \subseteq L \setminus X_0$ are open.
- There are no matching edges between $L \setminus X$ and $R \cap X$, else they would have been explored and added to X.

Hence, every vertex in $C = (L \setminus X) \cup (R \cap X)$ corresponds to a unique edge in the matching, and $|C| \leq |M|$. $\square$

Observe that the proof of König's theorem is algorithmic, and can be implemented to run in $O(m)$ time. Now, starting from some trivial matching, we can use this linear-time algorithm to repeatedly augment until we have a maximum matching. This means that maximum matching on bipartite graphs has an $O(mn)$-time algorithm.

Observe: this algorithm also gives a "proof of optimality" of the maximum matching M, in the form of a vertex cover of size $|M|$. By the easy direction of König's theorem, this is a vertex cover of *minimum cardinality*. Therefore, while finding the smallest vertex cover is NP-hard for general graphs, we have just solved the minimum vertex cover problem on bipartite graphs.

One other connection: if you have seen the Ford-Fulkerson algorithm for computing maximum flows, the above algorithm may seem familiar. Indeed, modeling the maximum matching problem in bipartite graphs as that of finding a maximum integer s-t flow, and running the Ford-Fulkerson "augmenting paths" algorithm results in the same result. Moreover, the minimum s-t cut corresponds to a vertex cover, and the max-flow min-cut theorem proves König's theorem. The figure to the right illustrates this on an example. Figure needs fixing.

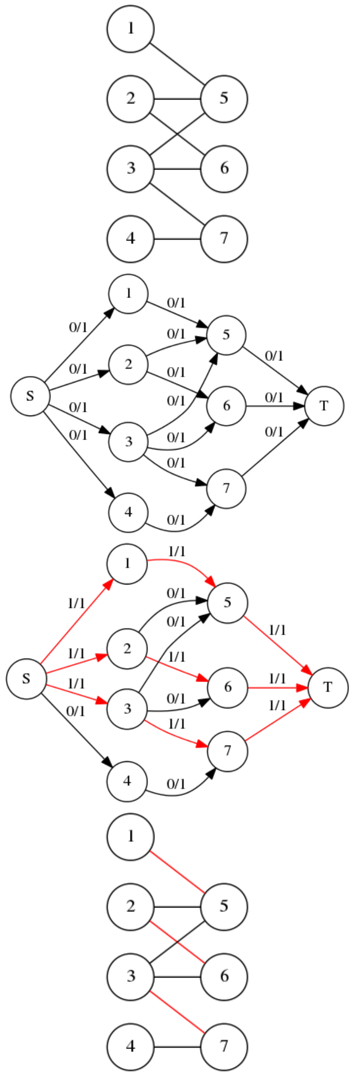


Figure 6.5: Use Ford-Fulkerson algorithm to find a matching

6.2.1 Other algorithms

There are faster algorithms to find maximum matchings in bipartite graphs. For a long time, the fastest one was an algorithm by John Hopcroft and Dick Karp, which ran in time $O(m\sqrt{n})$. It finds many augmenting paths at once, and then combines them in a clever way. There is also a related algorithm of Shimon Even and Bob Tarjan, which runs in time $O(\min(m\sqrt{m}, mn^{2/3}))$; in fact, they compute maximum flows on unit-capacity graphs in this running time.

Hopcroft and Karp (1973)

Even and Tarjan (1975)

There was remarkably little progress on the maximum matching problem until 2016, when Aleksander Madry gave an algorithm that runs in time $\tilde{O}(m^{10/7})$ time—in fact the algorithm also solves the unit-capacity maximum-flow problem in that time. It takes an interior-point algorithm for solving general linear programs, and specializes it to the case of maximum matchings. We may discuss this max-flow algorithm in a later chapter. The current best runtime for the unit-capacity maximum-flow problem is $m^{4/3+o(1)}$, due to Yang Liu and Aaron Sidford[2].

Madry (2016)

[2]

6.3 General Graphs: The Tutte-Berge Theorem

The matching problem on general (non-bipartite) graphs gets more involved, since the structure of matchings is richer. For example, the flow-based approaches do not work any more. And while Berge's theorem (Theorem 6.7) still holds in this case, König's theorem (Theorem 6.9) is no longer true. Indeed, the 3-cycle C_3 has a maximum matching of size 1, but the smallest vertex cover is of size 2. However, we can still give a min-max relationship, via the Tutte-Berge theorem.

To state it, let us give a definition: for a subset $U \subseteq V$, suppose deleting the nodes of U and their incident edges from G gives connected components $\{K_1, K_2, \dots, K_t\}$. The quantity $\text{odd}(G \setminus U)$ is the number of such pieces with an odd number of vertices.

Theorem 6.12 (The Tutte-Berge Max-Min Theorem)**.** *Given a graph G, the size of the maximum matching is described by the following equation.*

$$MM(G) = \min_{U \subseteq V} \frac{n + |U| - odd(G \setminus U)}{2}.$$

Tutte (1947), Berge (1958)

Tutte showed that the graph has a *perfect matching* precisely if for every $U \subseteq V$, $\text{odd}(G \setminus U) \leq |U|$. Berge gave the generalization to *maximum matchings*.

The expression on the right can seem a bit confusing, so let's consider some cases.

- If $U = \emptyset$, we get that if $|V|$ is even then $MM(G) \leq n/2$, and if $|V|$ is odd, the maximum matching cannot be bigger than $(n-1)/2$. (Or if G is disconnected with k odd-sized components, this gives $n/2 - k/2$.)

- Another special case is when U is any vertex cover with size c. Then the K_i's must be isolated vertices, so $\text{odd}(G \setminus U) = n - c$. This gives us $MM \leq \frac{c+n-(n-c)}{2} = c$, i.e., the size of the maximum matching is at most the size of any vertex cover.

- Give example where G is even, connected, but $MM < VC$.

Trying special cases is a good way to understand the

Proof of the $\leq$ direction of Theorem 6.12. The easy direction is to show that $MM(G)$ is at most the quantity on the right. Indeed, consider a maximum matching M. At most $|U|$ of the edges in M can be hit by nodes in U; the other edges must lie completely within some connected component of $G \setminus U$. The maximum size of a matching within K_i is $\lfloor K_i/2 \rfloor$, and it are these losses from the odd components that gives the expression on the right. Indeed, we get

$$\begin{aligned} |M| &\leq |U| + \sum_{i=1}^{t} \left\lfloor \frac{|K_i|}{2} \right\rfloor \\ &= |U| + \frac{n - |U|}{2} - \frac{\text{odd}(G \setminus U)}{2} \\ &= \frac{|U| + n - \text{odd}(G \setminus U)}{2}. \end{aligned}$$

We can prove the "hard" direction using induction (see the webpage for several such proofs). However, we defer it for now, and derive it later from the proof of the Blossom algorithm. □

6.4 The Blossom Algorithm

The Blossom algorithm for finding the maximum matching in a general graph is by Jack Edmonds. Recall: the algorithm for minimum-weight arborescences in §**??** was also due to him, and you may see some similarities in these two algorithms.

Edmonds (1965)

Theorem 6.13. *Given a graph G, the Blossom algorithm finds a maximum matching M in time $O(mn^2)$.*

The rest of this section defines the algorithm, and proves this theorem. The essential idea of the algorithm is simple, and similar to the one for the bipartite case: if we have a matching M, Berge's characterization from Theorem 6.7 says that if M is not optimal, there exists an M-augmenting path. So the natural idea would be to find such an augmenting path. However, it is not clear how to do this directly. The clever idea in the Blossom algorithm is to either find an M-augmenting path, or else find a structure called a "blossom". The good thing about blossoms is that we can use them to contract the graph in a certain way, and make progress. Let us now give some definitions, and details.

A ***flower*** is a subgraph of G that looks like the the object to the right: it has a open vertex at the base, then a ***stem*** with an even number of edges (alternating between matched and unmatched edges), and then a cycle with an odd number of edges (again alternating, though naturally having two unmatched edges adjacent to the stem). The cycle itself is called the ***blossom***.

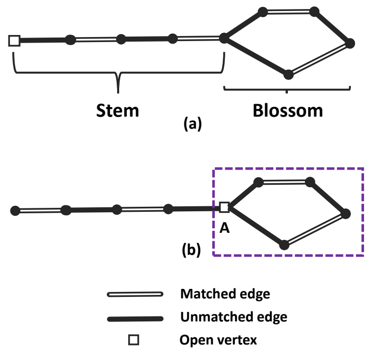


Figure 6.6: An example of blossom and the toggling of the stem.

6.4.1 *The Main Procedure*

The algorithm depends on a subroutine called `FindAugPath`, which has the following guarantee.

Lemma 6.14. *Given graph G and matching M, the subroutine `FindAugPath`, runs in $O(m)$ time. If G has an M-augmenting path, then it returns either (a) a flower F, or (b) an M-augmenting path.*

Note that we have not said what happens if there is no M-augmenting path. Indeed, we cannot find an augmenting path, but we show that the `FindAugPath` returns either a flower, or says "no M-augmenting path, and returns a Tutte-Berge set U achieving equality in Theorem 6.12 with respect to M. We can now use this `FindAugPath` subroutine within our algorithm as follows.

1. *Says "no M-augmenting path" and a set U of nodes.* In this case, M is the maximum matching.

2. *Finds augmenting path P.* We can now augment along P, by setting $M \leftarrow M \triangle P$.

3. *Finds a flower F.* In this case, we don't yet know if M is a maximum matching or not. But we can shrink the blossom down to get a smaller graph G' (and a matching M' in it), and recurse. Either we will find a proof of maximality of M' in G', or an M'-augmenting path. This we can extend to the matching M in G. That's the whole algorithm!

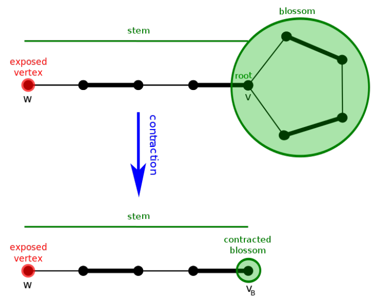


Figure 6.7: The shrinking of a blossom. Image found at `http://en.wikipedia.org/wiki/Blossom_algorithm`.

Let's give some more details for the last step. Suppose we find a flower F, with stem S and blossom B. First, toggle the stem (by setting $M \leftarrow M \triangle S$): this moves the open node to the blossom, without changing the size of the matching M. (It makes the following arguments easier, with one less case to consider.) (Change figure.) Next, contract the blossom down into a single vertex v_B, which is now open. Denote the new graph $G' \leftarrow G/B$, and $M' \leftarrow M/B$. Since all the nodes in blossom B, apart from perhaps the base, were matched by edges within the blossom, M' is also a matching in G'.

Given a graph and a subset $C \subseteq V$, recall that G/C denotes the ***contraction*** of C in G.

Next, we recurse on this smaller graph G' with matching M'. Finally, if we get back an M'-augmenting path, we "lift" it to get an M-augmenting path (as we see soon). Else if we find that M' is a maximum matching in G', we declare that M is maximum in G. To show correctness, it suffices to prove the following theorem.

Lemma 6.15. *Given graph G and matching M, suppose we shrink a blossom to get G' and M'. Then there exists an M-augmenting path in G if and only if there exists an M'-augmenting path in G'.*

Moreover, given an M'-augmenting path in G', we can lift it back to an M-augmenting path P in G in $O(m)$ time.

Proof. Since we toggled the stem, the vertex v at the base of the blossom B is open, and so is the vertex v_B created in G' by contracting B. Moreover, all other nodes in the blossom are matched by edges within itself, so all edges leaving B are non-matching edges. The picture essentially gives the proof, and can be used to follow along.

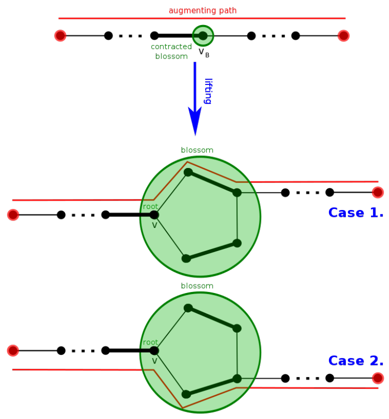


Figure 6.8: The translation of augmenting paths from $G \setminus B$ to G and back.

($\Rightarrow$) Consider an M-augmenting path in G, denoted by P. If P does not go through the blossom B, the path still exists in G'. Else if P goes through the blossom, we can assume that one of its endpoints is the base of the blossom (which is the only open node on the blossom)—indeed, any other M-augmenting path P can be rerouted to the base. (Figure!) So suppose this path P starts at the base and ends at some v' not in B. Because v_B is open in G', the path from v_B to v' is an M'-augmenting path in G'.

($\Leftarrow$) Again, an M'-augmenting path P' in G' that does not go through v_B still exists in G. Else, the M'-augmenting path P' passes through v_B, and because v_B is open in G', the path starts at v_B and ends at some node t. Let the first edge on P' be $e' = v_B y$ for some node y, and let it correspond to edge $e = xy$ in G, where $x \in B$. Now, if v is the open vertex at the base of the blossom, following one of the two paths (either clockwise or counter-clockwise) along the blossom from v to x, using the edge xy and then following the rest of the path P' from y to t gives an M-augmenting path in G. (This is where we use the fact that the cycle is odd, and is alternating except for the two edges incident to v.)

The process to get from P' in G' to the M-augmenting path in G be done algorithmically in $O(m)$ time, completing the proof. $\square$

We can now analyze the runtime, and prove Theorem 6.13:

Proof of Theorem 6.13. We first call `FindAugPath`, which takes $O(m)$ time. We are either done (because M is a maximum matching, or else we have an augmenting path), or else we contract down in another

$O(m)$ time to get a graph G' with at most $n-3$ vertices and at most m edges. Inductively, the time taken in the recursive call on G' is $O(m(n-3))$. Now lifting an augmenting path takes $O(m)$ time more. So the total runtime to find an augmenting path in G (if one exists) is $O(mn)$.

Finally, we start with an empty matching, so its size can be augmented at most $n/2$ times, giving us a total runtime of $O(mn^2)$. □

6.4.2 The `FindAugPath` Subroutine

The subroutine `FindAugPath` is very similar to the analogous procedure in the bipartite case, but since there is no notion of left and right vertices, we start with level X_0 containing *all* vertices that are unmatched in M_0, and try to grow M-alternating paths from them, in the hope of finding an M-augmenting path.

As before, let $X_{\leq j}$ denote $X_0 \cup \ldots \cup X_j$, and let nodes added to some level X_j be called ***marked***.

```
9.1 $X_0 \leftarrow$ all open vertices in $V$
9.2 for $i = 0, 1, 2, \ldots$ do
9.3     $X_{2i+1} \leftarrow \{v \mid$ exists $u \in X_{2i}$ s.t. $uv \notin M$, and $v \notin X_{\leq 2i}\}$
9.4     $X_{2i+2} \leftarrow \{v \mid$ exists $u \in X_{2i+1}$ s.t. $uv \in M$, and $v \notin X_{\leq 2i+1}\}$
9.5 if exists a "cross" edge between nodes of same level then
9.6     return augmenting path or flower
9.7 else
9.8     say "no $M$-augmenting path"
```

To argue correctness, let us look at the steps above in more detail. In line 9.2, for each vertex $u \in X_{2i}$, we consider the possible cases for each non-matching edge uv incident to it:

1. If v is not in $X_{\leq 2i+1}$ already (i.e., not marked already) then we add it to X_{2i+1}. Note that $v \in X_{2i+1}$ now has an M-alternating path to some node in X_0, that hits each layer exactly once.

2. If $v \in X_{2i}$, then uv is an unmatched edge linking two vertices in the same level. This gives an augmenting path or a blossom! Indeed, by construction, there are M-alternating paths P and Q from u and v to open vertices in X_0. If P and Q do not intersect, then concatenating path P, edge uv, and path Q gives an M-augmenting path. If P and Q intersect, they must first intersect some vertex $w \in X_{2j}$ for some $j \leq i$, and the cycle containing u, v, w gives us the blossom, with the stem being a path from w back to an open vertex in X_0.

3. If $v \in X_{2j}$ for $j < i$, then u would have been added to the odd level X_{2j+1}, which is impossible.

4. Finally, v may belong to some previous odd level, which is fine.

Observe that this "backward" non-matching edge uv is also an ***even-to-odd*** edge, like the "forward" edge in the first case.

Now for the edges out of the odd layers considered in line 9.3. Given $u \in X_{2i+1}$ and *matching* edge $uv \in M$, the cases are:

1. If v is not in $X_{\leq 2i+1}$ then add it to X_{2i+2}. Notice that v cannot be in X_{2i+2} already, since nodes in even layers are matched to nodes in the preceding odd layer, and there cannot be two matching edges incident to v.

   Again, observe inductively that v has a path to some vertex in X_0 that hits each intermediate layer once.

2. If v is in X_{2i+1}, there is an matching edge linking two vertices in the same odd level. This gives an augmenting path or a blossom, as in case 2 above. (Success!)

3. The node v cannot be in a previous level, because all those vertices are either open, or are matched using other edges.

Observe that if the algorithm does not succeed, all the matching edges we explored are ***odd-to-even***, whereas all the non-matching edges are ***even-to-odd***. Now we can prove Lemma 6.14.

Proof of Lemma 6.14. Let P be an M-augmenting path in G. For a contradiction, suppose we do not succeed in finding an augmenting path or blossom. Starting from one of the endpoints of P (which is in X_0, an even level), trace the path in the leveled graph created above. The next vertex should be in an odd level, the next in an even level, and so forth. Since the path P is alternating, `FindAugPath` ensures that all its edges will be explored. (Make sure you see this!) Now P has an odd number of edges (i.e., even number of vertices), so the last vertex has an opposite parity from the starting vertex. But the last vertex is open, and hence in X_0, an even level. This is a contradiction. □

6.4.3 *Finding a Tutte-Berge Set⋆*

If `FindAugPath` did not succeed, all the edges we explored form a bipartite graph. This does not mean that the entire graph is bipartite, of course—there can be non-matching edges incident to nodes in odd levels that lead to nodes that remain unmarked. But these components have no open vertices (which are all in X_0 and marked). Now define $U = X_{\text{odd}} := X_1 \cup X_3 \cup \ldots$ be the vertices in odd levels. Since there are no cross edges, each of these nodes has a distinct matching edge leading to the next level. Now $G \setminus U$ has two kinds of components:

(a) the marked vertices in the even levels, X_{even} which are all singletons since there are no cross edges, and

(b) the unmarked components, which have no open vertices, and hence have even size.

Hence

$$\begin{aligned}\frac{n+|U|-\text{odd}(G\setminus U)}{2} &= \frac{n+|X_{\text{odd}}|-|X_{\text{even}}|}{2}\\ &= \frac{2|X_{\text{odd}}|+(n-|X|)}{2}\\ &= |X_{\text{odd}}|+\frac{(n-|X|)}{2} = |M|.\end{aligned}$$

The last equality uses that all nodes in $V \setminus X$ are perfectly matched among themselves, and all nodes in X_{odd} are matched using unique edges.

The last piece is to show that a Tutte-Berge set U' for a contracted graph $G' = G/B$ with respect to $M' = M/B$ can be lifted to one for G with respect to M. We leave it as an exercise to show that adding the entire blossom B to U' gives such an U.

6.5 *Subsequent Work*

The best runtime of combinatorial algorithms for maximum matching in general graphs is $O(m\sqrt{n})$ by an algorithm of Silvio Micali and Vijay Vazirani[3]. The algorithm is based on finding augmenting paths much faster; it is quite involved, though a recent paper of Vijay Vazirani[4] giving a more approachable explanation. In a later chapter, we will see a very different "algebraic" algorithm based on fast matrix multiplication. This algorithm due to Marcin Mucha and Piotr Sankowski gives a runtime of $O(n^{\omega})$, where $\omega \approx 2.376$. Coming up next, however, is a discussion of weighted versions of matching, where edges have weights and the goal is to find the matching of maximum weight.

[3]

[4]

Mucha and Sankowski (2006)

7
Graph Matchings II: Weighted Matchings

In this chapter, we study the matching problem from the perspective of linear programs, and also learn results about linear programming using the matching problem as our running example. In fact, we see how linear programs capture the structure of many problems we have been studying: MSTs, min-weight arborescences, and graph matchings.

7.1 Linear Programming

We start with some basic definitions and results in Linear Programming. We will use these results while designing our linear program solutions for min-cost perfect matchings, min-weight arborescences and MSTs. This will be a sufficient jumping-off point for the contents of this lecture; a more thorough introduction to the subject can be found in the introductory text by Matoušek and Gärtner.

Definition 7.1. Let $\vec{a} \in \mathbb{R}^n$ be a vector and let $b \in \mathbb{R}$ a scalar. Then a *half-space* in $\mathbb{R}^n$ is a region defined by the set $\{\vec{x} \in \mathbb{R}^n \mid \vec{a} \cdot \vec{x} \geq b\}$.

As an example, the half space $S = \{\vec{x} \mid \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \vec{x} \geq 3\}$ in $\mathbb{R}^2$ is shown on the right. (Note that we implicitly restrict ourselves to *closed* half-spaces.)

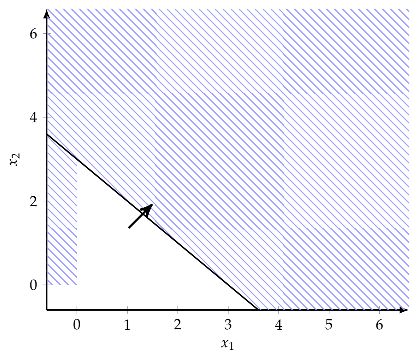


Figure 7.1: The half-space in $\mathbb{R}^2$ given by the set S

7.1.1 Polytopes and Polyhedra

Definition 7.2 (Polyhedron)**.** A *polyhedron* in $\mathbb{R}^n$ is the intersection of a finite number of half spaces.

A polyhedron is a convex region which is defined by finitely many linear constraints. A polyhedron in n dimensions with m constraints is often written compactly as

$$K = \{Ax \leq b\},$$

where A is an $m \times n$ ***constraint matrix***, x is an $n \times 1$ vector of variables, and b is an $m \times 1$ vector of constants.

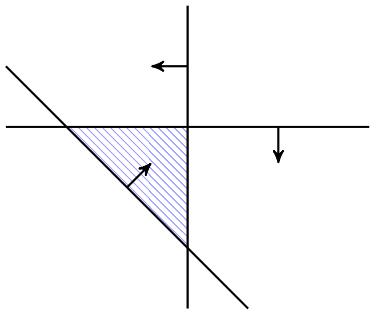


Figure 7.2: The polytope in $\mathbb{R}^2$ given by the constraints $-x_1 - x_2 \leq 1$, $x_1 \leq 0$, and $x_2 \leq 0$.

Definition 7.3 (Polytope)**.** A *polytope* $K \in \mathbb{R}^n$ is a bounded polyhedron.

In other words, a polytope is polyhedron such that there exists some radius $R > 0$ such that $K \subseteq B(\mathbf{0}, R) = \{x \mid \|x\|_2 \leq R\}$. A simple example of a polytope (where the bounded region of the polytope is highlighted by ▨) appears on the right. We can now define a linear program (often abbeviated as LP) in terms of a polyhedron.

Definition 7.4 (Linear Program)**.** For some integer n, a polyhedron $K = \{x \mid Ax \leq b\}$, and an n by 1 vector c, a *linear program* in n dimensions is the linear optimization problem

$$\min\{c \cdot x \mid x \in K\} = \min\{c \cdot x \mid Ax \leq b\}.$$

The set K is called the *feasible region* of the linear program.

Although all linear programs can be put into this canonical form, in practice they may have many different forms. These presentations can be shown to be equivalent to one another by adding new variables and constraints, negating the entries of A and c, etc. For example, the following are all linear programs:

$$\max_x\{c \cdot x : Ax \leq b\} \qquad \min_x\{c \cdot x : Ax = b\}$$
$$\min_x\{c \cdot x : Ax \geq b\} \qquad \min_x\{c \cdot x : Ax \leq b, x \geq 0\}.$$

The polyhedron K need not be bounded for the linear program to have a (finite) optimal solution. For example, the following linear program has a finite optimal solution even though the polyhedron is unbounded:

$$\min_x\{x_1 + x_2 \mid x_1 + x_2 \geq 3\}. \tag{7.1}$$

7.1.2 *Vertices, Extreme Points, and BFSs*

We now introduce three different classifications of some special points associated with polyhedra. (Several of these definitions extend to convex bodies.)

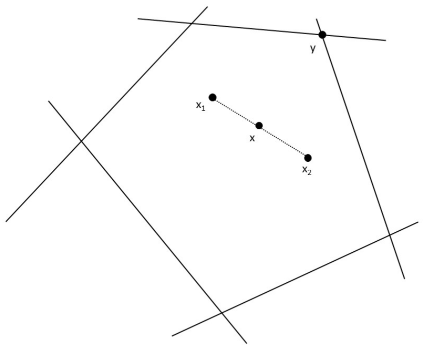


Figure 7.3: Here y is an extreme point, but x is not.

Definition 7.5 (Extreme Point)**.** Given a polyhedron $K \in \mathbb{R}^n$, a point $x \in K$ is an *extreme point* of K if there do not exist distinct $x_1, x_2 \in K$, and $\lambda \in [0, 1]$ such that $x = \lambda x_1 + (1 - \lambda)x_2$.

In other words, x is an extreme point of K if it cannot be written as the convex combination of two other points in K. See Figure 7.3 for an example.

Here's another kind of point in K.

In this course, we will use the notation $c \cdot x$, $c^\intercal x$, and $\langle c, x \rangle$ to denote the inner-product between vectors c and x.

Definition 7.6 (Vertex)**.** Given a polyhedron $K \subseteq \mathbb{R}^n$, a point $x \in K$ is a *vertex* of K if there exists an vector $c \in \mathbb{R}^n$ such that $c \cdot x < c \cdot y$ for all $y \in K$ $y \neq x$.

In other words, a vertex is the unique optimizer of some linear objective function. Equivalently, the hyperplane $\{y \in \mathbb{R}^n \mid c \cdot y = c \cdot x\}$ intersects K at the single point x. Note that there may be a polyhedron that does not have any vertices: e.g., one given by a single constraint, or two parallel constraints.

Finally, here's a third kind of special point in K:

Definition 7.7 (Basic Feasible Solution)**.** Given a polyhedron $K \in \mathbb{R}^n$, a point $x \in K$ is a *basic feasible solution* (bfs) for K if there exist n linearly independent defining constraints for K which x satisfies at equality.

In other words, let $K := \{x \in \mathbb{R}^n \mid Ax \leq b\}$, where the m constraints corresponding to the m rows of A are denoted by $a_i \cdot x \leq b_i$. Then $x^* \in \mathbb{R}^n$ is a basic feasible solution if there exist n linearly independent constraints for which $a_i \cdot x^* = b_i$, and moreover $a_i \cdot x^* \leq b_i$ for all other constraints (because x^* must belong to K, and hence satisfy all other constraints as well). Note there are only $\binom{m}{n}$ basic feasible solutions for K, where m is the total number of constraints and n is the dimension.

As you may have guessed by now, these three definitions are all related. In fact, they are all equivalent.

Fact 7.8. Given a polyhedron K and a point $x \in K$, the following are equivalent:

- x is a basic feasible solution,
- x is an extreme point, and
- x is a vertex.

While we do not prove it here, you could try to prove it yourself, or consult a textbook. For now, we proceed directly to the main fact we need for this section.

Fact 7.9. For a polytope K and a linear program $LP := \min\{c \cdot x \mid x \in K\}$, there exists an optimal solution $x^* \in K$ such that x^* is an extreme point/vertex/bfs of K.

Observe that we claimed Fact 7.9 for LPs whose feasible region is a polytope, since that suffices for today, but it can be proven with weaker conditions. However it is not true for all LPs: e.g., the LP in (7.1) has an infinite number of optimal solutions, none of which are at vertices.

This fact suggests an algorithm for LPs when K is a polytope: simply find all of the (at most $\binom{m}{n}$) basic feasible solutions and pick the one that gives the minimum solution value. Of course, there are more efficient algorithm to solve linear programs; we will talk about them in a later chapter. However, let us state a theorem—a very restricted form of the general result—about LP solving that will suffice for now:

Theorem 7.10. *There exist algorithms that take any LP* $\min\{c \cdot x \mid Ax = b, x \geq 0, x \in \mathbb{R}^n\}$*, where both the constraint matrix* A *and the RHS* b *have entries in* $\{0, 1\}$ *and* $\text{poly}(n)$ *rows, and output a basic feasible solution to this LP in* $\text{poly}(n)$ *time.*

We will see a sketch of the proof in a later chapter. Discuss the dependence on the number of bits to represent c? Or make this an informal theorem?

7.1.3 *Convex Hulls and an Alternate Representation*

The next definition allows us to give another representation of polytopes:

Definition 7.11 (Convex Hull). Given $x_1, x_2, \ldots, x_N \in \mathbb{R}^n$, the *convex hull* of $x_1, \ldots, x_N$ is the set of all convex combinations of these points. In other words, $\text{CH}(x_1, \ldots, x_N)$ is defined as

$$\left\{ x \in \mathbb{R}^n \;\middle|\; \exists \lambda_1, \ldots, \lambda_N \geq 0 \text{ s.t. } \sum_{i=1}^{N} \lambda_i = 1 \text{ and } x = \sum_{i=1}^{N} \lambda_i x_i \right\}. \qquad (7.2)$$

Put yet another way, the convex hull of $x_1, \ldots, x_N$ is the intersection of all convex sets that contain $x_1, \ldots, x_N$. It follows from the definition that the convex hull of finitely many points is a polytope. (Check!) We also know the following fact:

Fact 7.12. Given a polytope K with extreme points $\text{ext}(K)$,

$$K = \text{CH}(\text{ext}(K)).$$

The important insight that polytopes may be represented in terms of their extreme points, or their bounding half-planes. One representation may be easier to work with than the other, depending on the situation. The rest of this chapter will involve moving between these two methods of representing polytopes.

7.2 *Weighted Matchings in Bipartite Graphs*

While the previous chapters focused on finding maximum matchings in graphs, let us now consider the problem of finding a minimum-weight perfect matching in a graph with edge-weights. As before, we start with bipartite graphs, and extend our techniques to general graphs.

We are given a bipartite graph $G = (L, R, E)$ with edge-weights w_e. We want to use linear programs to solve the problem, so it is natural to have a variable x_e for each edge e of the graph. We want our solution to set $x_e = 1$ if the edge is in the minimum-weight perfect matching, and $x_e = 0$ otherwise. Compactly, this collection of variables gives us a $|E|$-dimensional vector $x \in \mathbb{R}^{|E|}$, that happens to contain only zeros and ones.

A bit of notation: for any subset $S \subseteq E$, let $\chi_S \in \{0,1\}^{|E|}$ denote the *characteristic vector* of this subset S, where χ_S has ones in coordinates that correspond to elements in S, and zeros in the other coordinates.

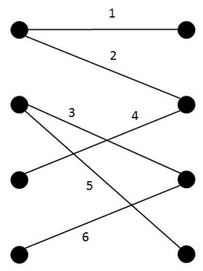


Figure 7.4: This graph has one perfect matching M: it contains edges 1, 4, 5, and 6, represented by the vector $\chi_M = (1,0,0,1,1,1)$.

7.2.1 *Goal: the Bipartite Perfect Matching Polytope*

It is conceptually easy to define an $|E|$-dimensional polytope whose vertices are precisely the perfect matchings of G: we simply define

$$C_{PM(G)} = CH(\{\chi_M \mid M \text{ is a perfect matching in } G\}). \tag{7.3}$$

And now we get a linear program that finds the minimum-weight perfect matching in a bipartite graph.

$$\min\{w \cdot x \mid x \in C_{PM(G)}\}.$$

By Fact 7.9, there is an optimal solution at a vertex of $C_{PM(G)}$, which by construction represents a perfect matching in G.

The good part of this linear program is that its feasible region has (a) only integer extreme points, (b) which are in bijection with the objects we want to optimize over. So optimizing over this LP will immediately solve our problem. (We can assume that there are linear program solvers which always return an optimal vertex solution, if one exists.) Moreover, the LP solver runs in time polynomial in the size of the LP.

The catch, of course, is that we have no control over the size of the LP, as we have written it. Our graph G may have an exponential number of matchings, and hence the definition of $C_{PM(G)}$ given in (7.3) is too unwieldly to work with. Of course, the fact that there are an exponential number of vertices does not mean that there cannot be a smaller representation using half-spaces. Can we find a compact way to describe $C_{PM(G)}$?

The unit cube

$$K = \{x \in \mathbb{R}^n \mid 0 \le x_i \le 1 \ \forall i\}$$

is a polytope with $2n$ constraints but 2^n vertices.

7.2.2 *A Compact Linear Program*

The beauty of the bipartite matching problem is that the "right" linear program is perhaps the very first one you may write. Here is the definition of the polytope using linear constraints:

$$K_{PM(G)} = \left\{ x \in \mathbb{R}^{|E|} \text{ s.t. } \begin{cases} \sum_{r \in N(l)} x_{lr} = 1 & \forall l \in L \\ \sum_{l \in N(r)} x_{lr} = 1 & \forall r \in R \\ x_e \geq 0 & \forall e \in E \end{cases} \right\}$$

The constraints of this polytope merely enforce that each coordinate is non-negative (which gives us $|E|$ constraints), and that the x_e values of the edges leaving each vertex sum to 1 (which gives us $|L| + |R|$ more constraints). All these constraints are satisfied by each χ_M corresponding to a matching M, which is promising. But it still always comes as a surprise to realize that his first attempt is actually successful:

Theorem 7.13. *For any bipartite graph G, $K_{PM(G)} = C_{PM(G)}$.*

Proof. For brevity, let us refer to the polytopes as K and C. The easy direction is to show that $C \subseteq K$. Indeed, the characteristic vector χ_M for each perfect matching M satisfies the constraints for K. Moreover K is convex, so if it contains all the vertices of C, it contains all their convex combinations, and hence all of C.

For the other direction, we show that an arbitrary vertex x^* of K is contained within C. Using Fact 7.8, we use the fact that x^* is also an extreme point for K. (We can also use the fact that x^* is a basic feasible solution, or that it is a vertex of the polytope, to prove this theorem; we will add the former proof soon, the latter proof appears in §7.3.)

Let $supp(x^*) = \{e \mid x^*_e > 0\}$ be the support of this solution. We claim that $supp(x^*)$ is acyclic. Indeed, suppose not, and cycle $C = e_1, e_2, \ldots, e_k$ is contained within the support $supp(x^*)$. Since the graph is bipartite, this is an even-length cycle. Define

$$\varepsilon := \min_{e \in supp(x^*)} x^*_e.$$

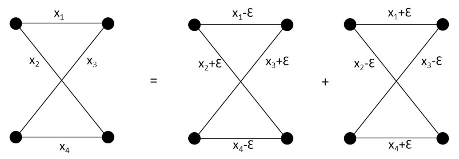


Figure 7.5: There cannot be a cycle in $supp(x^*)$, because this violates the assumption that x^* is an extreme point.

Observe that for all $e_i \in C$, $x^*_{e_i} + x^*_{e_{i+1}} \leq 1$, so $x^*_{e_i} \leq 1 - \varepsilon$. And of course $x^*_{e_i} \geq \varepsilon$, merely by the definition of ε. Now consider two solutions x^+ and x^-, where

$$x^+_{e_i} = x^*_{e_i} + (-1)^i \,\varepsilon$$

and

$$x^+_{e_i} = x^*_{e_i} - (-1)^i \,\varepsilon.$$

I.e., the two solutions add and subtract ε on alternate edges; this ensures that both the solutions stay within K. But then $x^* = \frac{1}{2}x^+ + \frac{1}{2}x^-$, contradicting our that x^* is an extreme point.

Therefore there are no cycles in $supp(x^*)$; this means the support is a forest. Consider a leaf vertex v in the support. Then, by the equality constraint at v, the single edge $e \in supp(x^*)$ leaving v must have $x^*_e = 1$. But this edge $e = uv$ goes to another vertex u; because x^* is in K, this vertex u cannot have other edges in $supp(x^*)$ without violating its equality constraint. So u and v are a matched pair in x^*. Now remove u and v from consideration. We have introduced no cycles into the remainder of $supp(x^*)$, so we may perform the same step inductively to show that x^* is the indicator of a perfect matching, and hence $x^* \in C$. This means all vertices of K are in C, which proves $C \subseteq K$, and completes the proof. $\square$

This completes the proof that the polytope $K_{PM(G)}$ exactly captures precisely the perfect matchings in G, despite having such a simple description. Now, using the fact that the linear program

$$\min\{w \cdot x \mid x \in K_{PM(G)}\}$$

can be solved in polynomial time, we get an efficient algorithm for finding minimum-weight perfect matching in graphs.

7.2.3 *A Proof via Basic Feasible Solutions*

Here is how to prove Theorem 7.13 using the notion of basic feasible solutions (bfs). Suppose $x^* \in \mathbb{R}^{|E|}$ is a bfs: we now show that $x^*_e \in \{0,1\}$ for all edges. By the definition of a bfs, there is a collection of $|E|$ tight linearly independent constraints that define x^*. These constraints are of two kinds: the degree constraints $\sum_{e \in \partial(v)} x_e = 1$ for some subset S of vertices, and the non-negativity constraints $x_e \geq 0$ for some subset $E' \subseteq E$ be edges. (Hence we have $|E'| + |S| = |E|$.)

By reordering columns and rows, we can put the degree constraints at the top, and put all the edges in E' at the end, to get that x^* is defined by:

$$\begin{bmatrix} C & C' \\ 0 & I \end{bmatrix} x^* = \begin{bmatrix} \mathbf{1}_s \\ \mathbf{0}_{m-s} \end{bmatrix}$$

where $C \in \{0,1\}^{s \times s}$, $C' \in \{0,1\}^{(m-s) \times s}$, and $m = |E|$ and $s = |S|$. The edges in E' have $x^*_e = 0$, so consider edges in $E \setminus E'$. By the linear independence of the constraints, we have C being non-singular, so

$$x^*|_{E \setminus E'} = C^{-1}(\mathbf{1} - C' x^*|_{E'}) = C^{-1}\mathbf{1}.$$

By Cramer's rule,

$$x^*_e = \frac{\det(C[\mathbf{1}]_i)}{\det(C)}.$$

The numerator is an integer (since the entries of C are integers), so showing $\det(C) \in \{\pm 1\}$ means that x^*_e is an integer.

Claim 7.14. Any $k \times k$-submatrix of C has determinant in $\{-1, 0, 1\}$.

Proof. The proof is by induction on k; the base case is trivial. If the submatrix D has a column with a single 1, we can expand using that entry, and use the inductive hypothesis. Else each column of D has two non-zeros. Recall that the columns of D correspond to some edges E_D in $E \setminus E'$, and the rows correspond to vertices S_D in S—two non-zeros in each column means each edge in E_D has both endpoints in S_D. Now if we sum rows for vertices in $S_D \cap L$ would give the all ones vector, as will summing up rows for vertices in $S_D \cap R$. (Here is the only place we're using bipartiteness.) In this case $\det(D) = 0$. □

Using the claim and using the fact C is non-singular and hence $\det(C)$ cannot be zero, we get that the entries of x_e^* are integers. By the structure of the LP, the only integers possible in a feasible solution are $\{0, 1\}$ and the vector x^* corresponds to a matching.

7.2.4 *Minimum-Weight Matchings*

How can we we find a minimum-weight (possibly non-perfect) matching in a bipartite graph G? If the edge weights are all non-negative, the empty matching would be the solution—but what if some edge weights are negative? (In fact, that's how we would find a maximum-weight matching–by negating all the weights.) As before, we can define the matching polytope for G as

$$C_{Match(G)} = CH(\{\chi_M \mid M \text{ is a matching in } G\}).$$

To write a compact LP that describes this polytope, we slightly modify our linear constraints as follows:

$$K_{Match} = \left\{ x \in \mathbb{R}^{|E|} \text{ s.t. } \begin{cases} \sum_{j \in R} x_{ij} \leq 1 & \forall i \in L \\ \sum_{j \in L} x_{ji} \leq 1 & \forall i \in R \\ x_{i,j} \geq 0 & \forall i, j \end{cases} \right\}$$

We leave it as an exercise to apply the techniques used in Theorem 7.13 to show that the vertices of K_{Match} are matchings of G, and hence the following theorem:

Theorem 7.15. *For any bipartite graph G, $K_{Match} = CH_{Match}$*

7.3 *Another Perspective: Buyers and sellers*

The results of the previous section show that the bipartite perfect matching polytope is integral, and hence the max-weight perfect matching problem on bipartite graphs can be be solved by "simply" solving the LP and getting a vertex solution. But do we need a generic linear program solver? Can we solve this problem faster? In this section, we develop (a variant of) the Hungarian algorithm that finds an optimal solutions using a "combinatorial" algorithm. This proof also shows that any vertex of the polytope $K_{PM(G)}$ is integral, and hence gives another proof of Theorem 7.13.

7.3.1 The Setting: Buyers and Items

Consider the setting with a set B with n *buyers* and another set I with n *items*, where buyer b has ***value*** v_{bi} for item i. The goal is to find a max-value perfect matching, that matches each buyer to a distinct item and maximizes the sum of the values obtained by this matching.

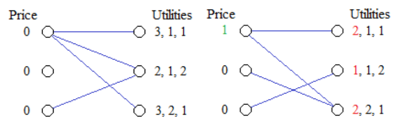
Our algorithm will maintain a set of ***prices*** for items: each item i will have price p_i. Given a price vector $p := (p_1, \dots, p_n)$, define the ***utility*** of item i to buyer b to be

$$u_{bi}(p) := v_{bi} - p_i.$$

Intuitively, the utility measures how favorable it is for buyer b to buy item i, since it factors in both the value and the price of the item. We say that buyer b *prefers* item i if item i gives the highest utility to buyer b, among all items. Formally, buyer $b \in B$ *prefers* item i at prices p if $i \in \arg\max_{i' \in I} u_{bi'}(p)$. The ***utility*** of buyer b at prices p is the utility of this preferred item:

$$u_b(p) := \max_{i \in I} u_{bi}(p) = \max_{i \in I}(v_{bi} - p_i). \tag{7.4}$$

A buyer has at least one preferred item, and can have multiple preferred items, since there can be ties. Given prices p, we build a ***preference graph*** $H = H(p)$, where the vertices are buyers B on the left, items I on the right, and where bi is an edge if buyer b prefers item i at prices p. The two examples show preference graphs, where the second graph results from an increase in price of item 1. Flip the figure.



Theorem 7.16. *For any price vector p^*, if the preference graph $H(p^*)$ contains a perfect matching M, then M is a max-value perfect matching.*

Proof. This proof uses weak linear programming duality. Indeed, recall the linear program we wrote for the bipartite perfect matching problem: we allow fractional matchings by assigning each edge bi a fractional value $x_{bi} \in [0,1]$.

$$\begin{array}{lll} \text{maximize} & \sum_{bi} v_{bi} x_{bi} & \\ \text{subject to} & \sum_{b=1}^{n} x_{bi} = 1 & \forall i \\ & \sum_{i=1}^{n} x_{bi} = 1 & \forall b \\ & x_{bi} \geq 0 & \forall (b,i) \end{array}$$

The perfect matching M is clearly feasible for this LP, so it remains to show that it achieves the optimum. Indeed, we show this by exhibiting a feasible dual solution with value $\sum_{bi \in M} v_{bi}$, the value of the primal solution. Then by weak duality, both these solutions must be optimal.

The dual linear program is the following:

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{n} p_i + \sum_{b=1}^{n} u_b & \\ \text{subject to} & p_i + u_b \geq v_{bi} & \forall bi \end{array}$$

(Observe that u and p are unconstrained variables.) In fact, given any settings of the p_i variables, the u_b variables that minimize the objective function, while still satisfying the linear constraints, are given by $u_b := \max_{i \in I}(v_{bi} - p_i)$, exactly matching (7.4). Hence, the dual program can then be rewritten as the following (non-linear, convex) program with no constraints:

$$\min_{p=(p_1,\ldots,p_n)} \quad \sum_{i \in I} p_i + \sum_{b \in B} u_b(p).$$

Consider the dual solution given by the price vector p^*. Recall that M is a perfect matching in the preference graph $H(p^*)$, and let $M(i)$ be the buyer matched to item i by it. Since $u_{M(i)}(p) = v_{M(i)i} - p_i$, the dual objective is

$$\sum_{i \in I} p_i^* + \sum_{b \in B} u_b(p^*) = \sum_{i \in I} p_i^* + \sum_{i \in I} (v_{M(i)i} - p_i) = \sum_{bi \in M} v_{bi}.$$

Since the primal and dual values are equal, the primal matching M must be optimal. □

Prices $p = (p_1, \ldots, p_n)$ are said to be ***market-clearing*** if each item can be assigned to some person who has maximum utility for it at these prices, subject to the constraints of the problem. In our setting, having such prices are equivalent to having a perfect matching in the preference graph. Hence, Theorem 7.16 shows that market-clearing prices give us an optimal matching, so our goal will be to find such prices.

7.3.2 *The Hungarian algorithm*

The "Hungarian" algorithm uses the buyers-and-sellers viewpoint from the previous section. The idea of the algorithm is to iteratively change item prices as long as they are not market-clearing, and the key is to show that this procedure terminates. To make our proofs easier, we assume for now that all the values v_{bi} are integers.

The algorithm was named the Hungarian algorithm by Harold Kuhn, who based his ideas on the works of Jenö Egervary and Dénes König. Munkres subsequently showed that the algorithm was in fact implementable in $O(n^3)$. Later, the algorithm was found to have been proposed even earlier by Carl Gustav Jacobi, before 1851.

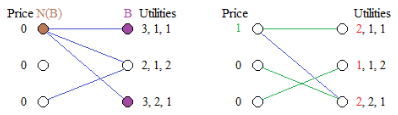
The price-changing algorithm proceeds as follows:

1. Initially, all items have price $p_i = 0$.

2. In each iteration, build the current preference graph $H(p)$. If it contains a perfect matching M, return it. Theorem 7.16 ensures that M is an optimal matching.

3. Otherwise, by Hall's theorem, there exists a set S of buyers such that if
$$N(S) := \{i \in I \mid \exists b \in S, bi \in E(H(p))\}$$
is the set of items preferred by at least one buyer in S, then $|N(S)| < |S|$. ($N(S)$ is the *neighborhood* of S in the preference graph.) Intuitively, we have many buyers trying to buy few items, so logically, the sellers of those items should raise their prices! The algorithm increases the price of every item in $N(S)$ by 1, and starts a new iteration by going back to step 2.

That's it. Running the algorithm on our running example gives the prices on the right.



The only way the algorithm can stop is to produce an optimal matching. So we must show it does stop, for which we use a "semi-invariant" argument. We keep track of the "potential"

$$\Phi(p) := \sum_i p_i + \sum_b u_b(p),$$

where p_i are the current prices and $u_b(p) = \max_i(v_{bi} - p_i)$ as above. This is just the dual value, and hence is is lower-bounded by the *optimal value* of the dual program. (We assume the optimal value of the LP is finite, e.g., if all the input values are finite.) Then, it suffices to prove the following:

Lemma 7.17. *Every time we increase the prices in $N(S)$ by 1, the value of $\sum_i p_i + \sum_b u_b$ decreases by at least 1.*

Proof. The value of $\sum_i p_i$ increases by $|N(S)|$, because we increase the price of each item $i \in N(S)$ by 1. For each buyer $b \in S$, the value u_b must decrease by 1, since all their preferred items had their prices increased by 1, and all other items previously had utilities at least one lower than the original $u_b(p)$. (Here, we used the fact that all values were integral.) Therefore, the value of the potential $\sum_i p_i + \sum_b u_b$ changes by $|N(B)| - |B| \leq -1$. □

Hence the algorithm stops in finite time, and produces a maximum-value perfect matching. By the arguments above **??** we get yet another proof of integrality of the LP **??** for the bipartite pefect matching problem. A few other remarks about the algorithm:

- In fact, one can get rid of the integrality assumption by raising the prices by the maximum amount possible for the above proof to still go through, namely

  $$\min_{b \in S} \left(u_b(p) - \max_{i \notin N(S)} (v_{ib} - p_i) \right).$$

  It can be shown that this update rule makes the algorithm stop in only $O(n^3)$ iterations.

- If all the values are non-negative, and we don't like the utilities to be negative, then we can do one of the following things: (a) when all the prices become non-zero, subtract the same amount from all of them to make the lowest price hit zero, or (b) choose S to be a minimal "consticted" set and raise the prices for $N(S)$. This way, we can ensure that each buyer still has at least one item which gives it nonngegative utility. (Exercise!)

- Suppose there are n buyers and a single item, with all non-negative values. (Imagine there are $n-1$ dummy items, with buyers having zero values for them.) The above algorithm behaves like the usual ascending-price English or Vickery auction, where prices are raised until only one bidder remains. Indeed, the final price for the "real" item will be such that the second-highest bidder is indifferent between it and a dummy item.

  This is a more general phenomenon: indeed, even in the setting with multiple items, the final prices are those produced by the Vickery-Clarke-Groves truthful mechanism, at least if we use the version of the algorithm that raises prices on minimal constricted sets. The truthfulness of the mechanism means there is no incentive for buyers to unilaterally lie about their values for items. See, e.g., [1] for the rich connection of matching algorithms to auction theory and (algorithmic) mechanism design.

[1]

Check about negative values, they don't seem to matter at all, as long as everything is finite. What about max-weight maximum matching: we can always convert the graph, but does the algorithm work out of the box?

This proof shows that for any setting of values, there is an optimal integer solution to the linear program

$$\max\{v \cdot x \mid x \in K_{LP(G)}\}.$$

This implies that every vertex x^* of the polytope $K_{LP(G)}$ is integral—indeed, the definition of vertex means x^* is the unique solution to the linear program for some values v, and our proof just produced an integral matching that is the optimal solution. Hence, we get another proof of Theorem 7.13, this time using the notion of vertices instead of extreme points.

7.4 *A Third Algorithm: Shortest Augmenting Paths*

Let us now see yet another algorithm for solving weighted matching problems in bipartite graphs. For now, we switch from maximum-weight matchings to minimum-weight matchings, because they are conceptually cleaner to explain here. Of course, the two problems are equivalent, since we can always negate edges.

In fact, we solve a min-cost max-flow problem here: given an flow network with terminals s and t, edge capacities u_e, and also edge costs/weights w_e, find an s-t flow with maximum flow value, and whose total cost/weight is the least among all such flows. (Moreover, if the capacities are integers, the flow we find will also have integer flow values on all edges.) Casting the maximum-cardinality bipartite matching problem as a integer max-flow problem, as in §blah gives us a minimum-weight bipartite matching.

This algorithm uses an augmenting path subroutine, much like the algorithm of Ford and Fulkerson. The subroutine, which takes in a matching M and returns one of size $|M| + 1$, is presented below. Then, we can start with the empty matching and call this subroutine until we get a maximum matching.

Let the original bipartite graph be G. Construct the **directed** graph G_M as follows: For each edge $e \in M$, insert that edge directed from right to left, with weight $-w_e$. For each edge $e \in G \backslash M$, insert that edge directed from left to right, with weight w_e. Then, compute the shortest path P that starts from the left and ends on the right, and return $M \triangle P$. It is easy to see that $M \triangle P$ is a matching of size $|M| + 1$, and has total weight equal to the sum of the weights of M and P.

Call a matching M an extreme matching if M has minimum weight among all matchings of size $|M|$. The main idea is to show that the above subroutine preserves extremity, so that the final matching must be extreme and therefore optimal.

Theorem 7.18. *If M is an extreme matching, then so is $M \triangle P$.*

Proof. Suppose that M is extreme. We will show that there exists an augmenting path P such that $M \triangle P$ is extreme. Then, since the algorithm finds the shortest augmenting path, it will find a path that is no longer than P, so the returned matching must also be extreme.

Consider an extreme matching M' of size $|M|+1$. Then, the edges in $M \triangle M'$ are composed of disjoint paths and cycles. Since $M \triangle M'$ has more edges in M' than edges in M, there is some path $P \subset M \triangle M'$ with one more edge in M' than in M. This path necessarily starts and ends on opposite sides, so we can direct it to start from the left and end on the right. We know that $|M' \cap P| = |M \cap P| + 1$, which means that $M \backslash P$ and $M' \backslash P$ must have equal size. The total weight of $M \backslash P$ and $M' \backslash P$ must be the same, since otherwise, we can swap the two matchings and improve one of M and M'. Therefore, $M \triangle P = (M' \cap P) \cup (M \backslash P)$ has the same weight as M' and is extreme. $\square$

Note that the formulation of G_M is exactly the graph constructed if we represent the minimum matching problem as a min-cost flow. Indeed, the previous theorem can be generalized to a very similar statement for the augmenting path algorithm for min-cost flows.

7.5 *Perfect Matchings in General Graphs*

Interestingly, Theorem 7.13 is false for non-bipartite graphs. Indeed, consider graph K_3 which consists of a single 3-cycle: this graph has no perfect matching, but setting $x_e = 1/2$ for each edge satisfies all the constraints. This suggests that the linear constraints defining $K_{PM(G)}$ are not enough, and we need to add more constraints to capture the convex hull of perfect matchings in general graphs.

In situations like this, it is instructive to look at the counter-example, to see what constraints must be satisfied by any integer solution, but are violated by this fractional solution. For a set of vertices $S \subseteq V$, let $\partial(S)$ denote the edges leaving S. Here is one such set of constraints:

$$\left\{ \sum_{e \in \partial(S)} x_e \geq 1 \qquad \forall S \subseteq V \text{ such that } |S| \text{ is odd,} \right\}.$$

These constraints say: the vertices belonging to a set $S \subseteq V$ of odd size cannot be perfectly matched within themselves, and at least one edge from any perfect matching must leave S. Indeed, this constraint would be violated by $S = V(K_3)$ in the example above.

Adding these ***odd-set constraints*** to the previous degree constraints gives us the following polytope, which was originally proposed by Edmonds [2]:

[2] ; and

$$K_{\text{genPM(G)}} = \left\{ x \in \mathbb{R}^{|E|} \text{ s.t. } \begin{cases} \sum_{u \in \partial(v)} x_{vu} = 1 & \forall v \in V \\ \sum_{e \in \partial(S)} x_e \geq 1 & \forall S \text{ s.t. } |S| \text{ odd} \\ x_e \geq 0 & \forall e \in E \end{cases} \right\}$$

Theorem 7.19. *For an undirected graph G, we have*

$$K_{genPM(G)} = C_{genPM(G)},$$

where $C_{genPM(G)}$ the convex hull of all perfect matchings of G.

One proof is a more sophisticated version of the one in §7.2.3, where we may now have tight odd-set constraints; we leave it as a slightly challenging exercise.

This LP potentially contains a exponential number of constraints, in contrast with the linear number of constraints needed for the bipartite case. In fact, a powerful theorem by Thomas Rothvoß (building on work by Mihalis Yannakakis) shows that any polytope whose vertices are the perfect matchings of the complete graph on n vertices must contain an exponential number of constraints.

Rothvoß (2014,2017)

Yannakakis (1991)

Given this negative result, this LP seems useless: we cannot even write it down explicitly in polynomial time! It turns out that despite this large size, it is possible to solve this LP in polynomial time. In a later lecture, we will see the Ellipsoid method to solve linear programs. This method can solve such a large LP, provided we give it a helper procedure called a "separation oracle", which, given a point $x \in \mathbb{R}^{|E|}$, outputs YES if x lies is within the desired polytope, and otherwise it outputs NO and returns a violated constraint of this LP. It is easy to check if x satisfies the degree constraints, so the challenge here is to find an odd set S with $\sum_{e \in \partial(S)} x_e < 1$, if there exists one. Such an algorithm can be indeed obtained using a sequence of min-cut computations in the graph, as was shown by. We will see this in a HW problem later in the course.

Padberg and Rao (1982)

7.6 *Integrality of Polyhedra*

We just saw several proofs that the bipartite perfect matching polytope has a compact linear program. Moreover, we claimed that the pefect matching polytope on general graphs has an explicit linear program that, while exponential-sized, can be solved in polynomial time. Such results allow us to solve the weighted bipartite matching problems using generic linear programming solvers (as long as they return vertex solutions).

Having many different ways to view a problem gives us a deeper insight, and thereby come up with faster and better ways to solve it. Moreover, these different perspectives give us a handle into solving extensions of these problems. E.g., if we have a matching problem with two different kinds weights w_1 and w_2 on the edges: we want to find a matching $x \in K_{PM(G)}$ minimizing $w_1 \cdot x$, now subject to the additional constraint $w_2 \cdot x \leq B$. While the problem is now NP-hard, this linear constraint can easily be added to the linear program to get a fractional optimal solution. Then we can reason about how to "round" this solution to get a near-optimal matching.

We now show how two problems we considered earlier, namely minimum-cost arborescence and spanning trees, can be exactly modeled using linear programs. We then conclude with a pointer to a general theory of integral polyhedra.

7.6.1 *Arborescences*

We already saw a linear program for the min-weight r-arborescence polytope in §2.3.2: since each node that is not the root r must have a path in the arborescence to the root, it is natural to say that for any subset of vertices $S \subseteq V$ that does not contain the root, there must be an edge leaving it. Specifically, given the digraph $G = (V, A)$, the polytope can be written as

$$K_{Arb(G)} = \left\{ x \in \mathbb{R}^{|A|} \text{ s.t. } \begin{cases} \sum\limits_{a \in \partial^+(S)} x_a \geq 1 & \forall S \subset V s.t. r \notin S \\ x_a \geq 0 & \forall a \in A \end{cases} \right\}.$$

Here $\partial^+(S)$ is the set of arcs that leave set S. The proof in §2.3.2 already showed that for each weight vector $w \in \mathbb{R}^{|A|}$, we can find an optimal solution to the linear program $\min\{w \cdot x \mid x \in K_{Arb(G)}\}$.

7.6.2 *Minimum Spanning Trees*

One way to write an LP for minimum spanning trees is to reduce it to minimum-weight r-arborescences: indeed, replace each edge by two arcs in opposite directions, each having the same cost. Pick any node as the root r. Observe the natural bijection between r-arborescence in this digraph and spanning trees in the original graph, having the same weight.

But why go via arborescences? Why not directly model the fact that any tree has at least one undirected edge crossing each cut $(S, V \setminus S)$, perhaps as follows:

$$K_{STtry} = \left\{ x \in \mathbb{R}^{|E|} \text{ s.t. } \begin{cases} \sum\limits_{e \in \partial(S)} x_e \geq 1 & \forall S \subset V, S \neq \emptyset, V \\ x_e \geq 0 & \forall e \in E \end{cases} \right\}.$$

(The first constraint excludes the case where S is either empty or the entire vertex set.) Sadly, this does not precisely capture the spanning tree polytope: e.g., for the familiar cycle graph having three vertices, setting $x_e = 1/2$ for all three edges satisfies all the constraints. If all edge weights are 1, this solution get a value of $\sum_e x_e = 3/2$, whereas any spanning tree on 3 vertices must have 2 edges.

One can indeed write a different linear program that captures the spanning tree polytope, but it is a bit non-trivial:

$$K_{ST(G)} = \left\{ x \in \mathbb{R}^{|E|} \text{ s.t. } \begin{cases} \sum_{ij \in E: i,j \in S} x_{ij} \leq |S| - 1 & \forall S \subseteq V, S \neq \emptyset \\ \sum_{ij \in E} x_{ij} = |V| - 1 \\ x_{ij} \geq 0 & \forall ij \in E \end{cases} \right\}$$

Define the convex hull of all minimum spanning trees of G to be CH_{MST}. Then, somewhat predictably, we will again find that $CH_{MST} = K_{MST}$.

Both the polytopes for arborescences and spanning trees had exponentially many constraints. Again, we can solve these LPs if we are given separation oracles for them, i.e., procedures that take x and check if it is indeed feasible for the polytope. If it is not feasible, the oracle should output a violated linear inequality. We leave it as an exercise to construct separation oracles for the polytopes above.

A different approach is to represent such a polytope K compactly via an ***extended formulation***: i.e., to define a polytope $K' \in \mathbb{R}^{n+n'}$ using a polynomial number of linear contraints (on the original variables $x \in \mathbb{R}^n$ and perhaps some new variables $y \in \mathbb{R}^{n'}$) such that projecting K' down onto the original n-dimensions gives us K. I.e., we want that

$$K = \{x \in \mathbb{R}^n \mid \exists y \in \mathbb{R}^{n'} s.t. \ (x, y) \in K'\}.$$

The homework exercises will ask you to write such a compact extended formulation for the arborescence problem.

7.6.3 *Integrality of Polyhedra*

This section still needs work. We have seen that LPs are a powerful way of formulating problems like min-cost matchings, min-weight r-aborescences, and MSTs. We reasoned about the structure of the polytopes that underly the LPs, and we were able to show that these LPs do indeed solve their combinatorial problems. But notice that simply forming the LP is not sufficient–significant effort was expended to show that these polytopes do indeed have integer solutions at the vertices [3]. Without this guarantee, we could get fractional

[3] i.e. that the solution vector is integer valued

solutions to these LPs that do not actually give us solutions to our problem.

There is a substantial field of study concerned with proving the integrality of various LPs. We will briefly introduce a matrix property that implies the integrality of corresponding LPs. Recall that an LP can be written as

$$[A]_{m\times n} \cdot \vec{x} \leq \vec{b}$$

where A is a $m \times n$ matrix with each row corresponding to a constraint, $\vec{x}$ is a vector of n variables, and $\vec{b} \in \mathbb{R}^m$ is a vector corresponding to the m scalars $b_i \in \mathbb{R}$ in the constraint $A^{(i)} \cdot \vec{x} \leq b_i$.

Definition 7.20. A matrix $[A]_{m\times n}$ is called *totally unimodular* if every square submatrix B of A has the property that $\det(B) \in \{0, \pm 1\}$

We then have the following neat theorem, due to Hoffman and Kruskal:

Theorem 7.21 (Hoffman and Kruskal Theorem). *If the constraint matrix $[A]_{m\times n}$ is totally unimodular and the vector $\vec{b}$ is integral, i.e: $\vec{b} \in \mathbb{Z}^m$, then, the vertices of the polytope induced by the LP are integer valued.*

A.J. Hoffman and J.B. Kruskal (1956)

Thus, to show that the vertices are indeed integer valued, one need not go through producing combinatorial proofs, as we have. Instead, one could just check that the constraint matrix A is totally unimodular.

=

8
Graph Matchings III: Algebraic Algorithms

We now introduce some algebraic methods to find perfect matchings in general graphs. We use so-called the "polynomial method", based on the elementary fact that low-degree polynomials have few zeroes. This is a powerful and versatile idea, using a combination of basic algebra and randomness, that can be used to solve many related problems as well. For instance, we will use it to get parallel (randomized) algorithms for perfect matchings, and also to find *red-Blue perfect matchings*, an algorithm for which we know no deterministic algorithms. But before we digress to these problems, let us discuss some of the algebraic results for perfect matchings.

We focus on perfect matchings here; it is an exercise to reduce finding maximum matchings to perfect matchings. Check!

- The first result along these lines is that of Laci Lovász, who introduced the general idea, and gave a randomized algorithm to detect the presence of perfect matchings in time $O(n^\omega)$, and to find it in time $O(mn^\omega)$. We will present all the details of this elegant idea soon.

Lovász (1979)

- Dick Karp, Eli Upfal, and Avi Wigderson, and then Ketan Mulmuley, Umesh Vazirani, and Vijay Vazirani showed how to find such a matching in parallel. The question of getting a deterministic parallel algorithm remains an outstanding open problem, despite recent progress (which discuss at the end of the chapter).

Karp, Upfal, and Wigderson (1986)

Mulmuley, Vazirani, and Vazirani (1987)

- Michael Rabin and Vijay Vazirani sped up the sequential algorithm to run in $O(n \cdot n^\omega)$. This was substantially improved by the work of Marcin Mucha and Piotr Sankowski to get a runtime of $O(n^\omega)$.

Rabin and Vazirani (1989)

Mucha and Sankowski (2006)

8.1 Preliminaries: roots of low degree polynomials

For the rest of this lecture, we fix a field $\mathbb{F}$, and consider (univariate and multivariate) polynomials over this field. We assume that we can perform basic arithmetic operations in constant time, though sometimes it will be important to look more closely at this assumption.

For finite fields $\mathbb{F}_q$ (where q is a prime power), we can perform arithmetic operations (addition, multiplication, division) in time poly $\log q$.

Given $p(x)$, a ***root/zero*** of this polynomial is some value z such that $p(z)$ evaluates to zero. The critical idea for today's lecture is simple: ***low-degree polynomials have "few" roots***. In this section, we will see this for both univariate and multivariate polynomials, for the right notion of "few". The following theorem well-knwon is for univariate polynomials. (The proof is essentially by induction on the degree; will add a reference.)

Theorem 8.1 (Univariate Few-Roots Theorem)**.** *A univariate polynomial $p(x)$ of degree at most d over any field $\mathbb{F}$ has at most d roots, unless $p(x)$ is zero polynomial.*

Now, for multivariate polynomials, the trivial extension of this theorem is not true. For example, $p(x, y) := xy$ has degree two, and the solutions to $p(x, y) = 0$ over the reals are exactly the points in $\{(x, y) \in \mathbb{R}^2 : \ x = 0 \text{ or } y = 0\}$, which is infinite. However, the roots are still "few", in the sense that the set of roots is very sparse in $\mathbb{R}^2$. To formalize this observation, let us write a trivial corollary of Theorem 8.1:

Corollary 8.2. *Given a non-zero univariate polynomial $p(x)$ over a field $\mathbb{F}$, such that p has degree at most d. Suppose we choose R uniformly at random from a subset $S \subseteq \mathbb{F}$. Then*

$$\Pr\left[p(R) = 0\right] \leq \frac{d}{|S|}.$$

This statement holds for multivariate polynomials as well, as we see next. The result is called the ***Schwartz-Zippel lemma***, and it appears in papers by Richard DeMillo and Richard Lipton [1], by Richard Zippel, and by Jacob Schwartz.

[1]

Zippel (1979)

Schwartz (1980)

Like many powerful ideas, the provenance of this result gets complicated. A version of this for finite fields was apparently already proved in 1922 by Øystein Ore; anyone have a copy of that paper?

A *monomial* is a product a collection of variables. The degree of a monomial is the sum of degrees of the variables in it. The degree of a polynomial is the maximum degree of any monomial in it.

Theorem 8.3. *Let $p(x_1, \ldots, x_n)$ be a non-zero polynomial over a field $\mathbb{F}$, such that p has degree at most d. Suppose we choose values $R_1, \ldots, R_n$ independently and uniformly at random from a subset $S \subseteq \mathbb{F}$. Then*

$$\Pr\left[p(R_1, \ldots, R_n) = 0\right] \leq \frac{d}{|S|}.$$

Hence, the number of roots of p inside S^n is at most $d|S|^{n-1}$.

Proof. We argue by induction on n. The base case of $n = 1$ considers univariate polynomials, so the claim follows from Theorem 8.1. Now for the inductive step for n variables. Let k be the highest power of x_n that appears in p, and let q be the quotient and r be the remainder when dividing p by x_n^k. That is, let $q(x_1, \ldots, x_{n-1})$ and $r(x_1, \ldots, x_n)$ be the (unique) polynomials such that

$$p(x_1, \ldots, x_n) = x_n^k\, q(x_1, \ldots, x_{n-1}) + r(x_1, \ldots, x_n),$$

where the highest power of x_n in r is less than k. Now letting $\mathcal{E}$ be the event that $q(R_1, \ldots, R_{n-1})$ is zero, we find

$$\begin{aligned}\Pr[p(R_1,\ldots,R_n)=0] &= \Pr[p(R_1,\ldots,R_n)=0 \mid E]\Pr[\mathcal{E}] \\ &\quad + \Pr\left[p(R_1,\ldots,R_n)=0 \mid \overline{\mathcal{E}}\right]\Pr\left[\overline{\mathcal{E}}\right] \\ &\leq \Pr[\mathcal{E}] + \Pr\left[p(R_1,\ldots,R_n)=0 \mid \overline{\mathcal{E}}\right]\end{aligned}$$

By the inductive assumption, and noting that q has degree at most $d-k$, we know

$$\Pr[\mathcal{E}] = \Pr[q(R_1,\ldots,R_{n-1})=0] \leq (d-k)/|S|.$$

Similarly, fixing the values of $R_1, \ldots, R_{n-1}$ and viewing p as a polynomial only in variable x_n (with degree k), we know

$$\Pr\left[p(R_1,\ldots,R_n)=0 \mid \overline{\mathcal{E}}\right] \leq k/|S|.$$

Thus we get

$$\Pr[p(R_1,\ldots,R_n)=0] \leq \frac{d-k}{|S|} + \frac{k}{|S|} = \frac{d}{|S|}. \qquad \square$$

Remark 8.4. Finding the set $S \subseteq \mathbb{F}$ such that $|S| \geq dn^2$, guarantees that if p is a non-zero polynomial,

$$\Pr[p(R_1,\ldots,R_n)=0] \leq \frac{1}{n^2}.$$

Naturally, if p is zero polynomial, then the probability equals 1.

8.2 *Detecting Perfect Matchings by Computing a Determinant*

Let us solve the easier problem of detecting a perfect matching in a graph, first for bipartite graphs, and then for general graphs. We define the *Edmonds matrix* of a bipartite graph G.

Definition 8.5. For a bipartite graph $G = (L, R, E)$ with $|L| = |R| = n$, its *Edmonds matrix* $\mathbf{E}(G)$ is the following $n \times n$ matrix of indeterminates/variables.

$$\mathbf{E}_{i,j} = \begin{cases} 0 & \text{if } (i,j) \notin E \text{ and } i \in L, j \in R \\ x_{i,j} & \text{if } (i,j) \in E \text{ and } i \in L, j \in R. \end{cases}$$

Example 8.6. The Edmonds matrix of the graph to the right is

$$\mathbf{E} = \begin{bmatrix} x_{11} & x_{12} \\ 0 & x_{22} \end{bmatrix},$$

which has determinant $x_{11}x_{22}$.

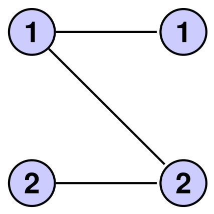


Figure 8.1: Bipartite graph

Recall the Leibniz formula to compute the determinant, and apply it to the Edmonds matrix:

$$\det(\mathbf{E}(G)) = \sum_{\sigma \in S_n} (-1)^{\mathrm{sign}(\sigma)} \prod_{i=1}^{n} \mathbf{E}_{i,\sigma(i)}$$

There is a natural correspondence between potential perfect matchings in G and permutations $\sigma \in S_n$, where we match each vertex $i \in L$ to vertex $\sigma(i) \in R$. Moreover, the term in the above expansion corresponding to a permutation σ gives a non-zero *monomial* (a product of x_{ij} variables) if and only if all the edges coresponding to that permutation exist in G. Moreover, all the monomials are distinct, by construction. This proves the following simple claim.

Proposition 8.7. *Let* $\mathbf{E}(G)$ *denote the Edmonds matrix of a bipartite graph* G*. Then* $\det(\mathbf{E}(G))$ *is a non-zero polynomial (over any field* $\mathbb{F}$*) if and only if* G *contains a perfect matching.*

However, writing out this determinant of indeterminates could take exponential time—it would correspond to a brute-forece check of all possible perfect matchings. Lovász's idea was to use the randomized algorithm implicit in Theorem 8.3 to ***test*** whether G contains a perfect matching.

```
Algorithm 10: PM-tester(bipartite graph $G$, $S \subseteq \mathbb{F}$)
10.1 $\mathbf{E} \leftarrow$ Edmonds matrix for graph $G$
10.2 For each non-zero entry $\mathbf{E}_{ij}$, sample $R_{ij} \in S$ independently and
       uniformly at random
10.3 $\widetilde{\mathbf{E}} \leftarrow \mathbf{E}(\{R_{ij}\}_{i,j})$ be matrix with sampled values substituted
10.4 if $\det(\widetilde{\mathbf{E}}) = 0$ then
10.5     return $G$ does not have a perfect matching (No)
10.6 else
10.7     return $G$ contains a perfect matching (Yes)
```

Lemma 8.8. *For* $|S| \geq n^3$*, Algorithm 10 always returns* `No` *if* G *has no perfect matching, while it says* `Yes` *with probability at least* $1 - \frac{1}{n^2}$ *otherwise. Moreover, the algorithm can be implemented in time* $O(n^\omega)$*, where* ω *is the exponent of matrix multiplication.*

Proof. The success probability follows from Remark 8.4, and the fact that the determinant is a polynomial of degree n. Assuming that arithmetic operations can be done in constant time, we can compute the determinant of $\widetilde{\mathbf{E}}$ in time $O(n^3)$, using Gaussian elimination. [2] Hence Algorithm 10 easuly runs in time $O(n^3)$. In fact, Bunch and Hopcroft [3] proved that both computing matrix inverses and determinants can be done in asymptotically the same time as matrix multiplication. Thus, we can make Algorithm 10 run in time $O(n^\omega)$. □

[2] If we work over finite fields, the size of the numbers is not an issue. However, Gaussian Elimination over the rationals could cause some of the numbers to get unreasonably large. Ensuring that the numbers remain polynomially bounded requires care; see Edmonds' paper, or a book on numerical methods.

[3]

8.2.1 Non-bipartite matching

The extension to the non-bipartite case requires a very small change: instead of using the Edmonds matrix, which is designed for bipartite graphs, we use the analogous object for general graphs. This object was introduced by Bill Tutte in his 1947 paper with the Tutte-Berge theorem.

Tutte (1947)

Definition 8.9. For a general graph $G = (V, E)$ with $|V| = n$, the *Tutte matrix* $\mathbf{T}(G)$ of G is the $n \times n$ skew-symmetric matrix given by

A matrix A is ***skew-symmetric*** if $A^\intercal = -A$.

$$\mathbf{T}_{i,j} = \begin{cases} 0 & \text{if } (i,j) \notin E \text{ or } i = j \\ x_{i,j} & \text{if } (i,j) \in E \text{ and } i < j \\ -x_{j,i} & \text{if } (i,j) \in E \text{ and } i > j. \end{cases}$$

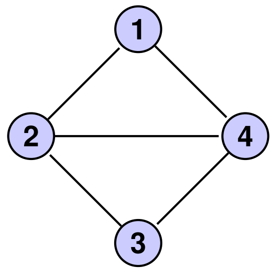


Figure 8.2: Non-bipartite graph

Example 8.10. For the graph to the right, the Tutte matrix is

$$\begin{bmatrix} 0 & x_{1,2} & 0 & x_{1,4} \\ -x_{1,2} & 0 & x_{2,3} & x_{2,4} \\ 0 & -x_{2,3} & 0 & x_{3,4} \\ -x_{1,4} & -x_{2,4} & -x_{3,4} & 0 \end{bmatrix}$$

And its determinant is blah blah.

Observe that now each variable occurs twice, with the variables below the diagonal being the negations of those above. We claim the same property for this matrix as we did for the Edmonds matrix:

Theorem 8.11. *For any graph G, the determinant of the Tutte matrix $\mathbf{T}(G)$ is a non-zero polynomial over any field $\mathbb{F}$ if and only if there exists a perfect matching in G.*

Proof. As before, the determinant is

$$\det(\mathbf{T}(G)) = \sum_{\sigma} (-1)^{\text{sign}(\sigma)} \prod_{i=1}^{n} \mathbf{T}_{i,\sigma(i)}.$$

One direction of the theorem is easy: if G has a perfect matching M, consider the permutation σ mapping each vertex to the other endpoint of the matching edge containing it. The corresponding monomial above is $\pm \prod_{e \in M} x_e^2$, which cannot be cancelled by any other permutation, and makes the determinant non-zero over any field.

To prove the converse, suppose the determinant is non-zero. In the monomial corresponding to permutation σ, either each $(i, \sigma(i))$ in the product corresponds to an edge of G, or else the monomial is zero. This means each non-zero monomial of $\det(\mathbf{T}(G))$ chooses an edge incident to i, for each vertex $i \in [n]$, giving us a *cycle cover* of G.

If the cycle cover for σ has an odd-length cycle, take the permutation σ' obtained by reversing the order of, say, the first odd cycle in it. This does not change the sign of the permutation, which depends on how many odd and even cycles there are, but the skew symmetry and the odd cycle length means the product of matrix entries flips sign. Hence the monomials for σ and σ' cancel each other. Formally, we get a bijection between permutations with odd-length cycles that cancel out.

The remaining monomials corresponding to cycle covers with even cycles. Choosing either the odd edges or even edges on each such even cycle gives a perfect matching. □

Now given Theorem 8.11, the Tutte matrix can simply be substituted instead of the Edmonds matrix to extend the results to general graphs.

8.3 From Detecting to Finding Perfect Matchings

We can convert the above perfect matching tester (which solves the ***decision version*** of the perfect matching problem) into an algorithm for the ***search version***: one that outputs a perfect matching in a graph (if one exists), using the simple but brilliant idea of ***self-reducibility***. Suppose that graph G has a perfect matching. Then we can pick any edge $e = uv$ and check if $G[E-e]$, the subgraph of G obtained by dropping just the edge e, contains a perfect matching. If not, then edge e must be part of every perfect matching in G, and hence we can find a perfect matching on the induced subgraph $G[V \setminus \{u,v\}]$. The following algorithm is based on this observation.

We are reducing the problem to smaller instances of itself, hence the name.

Algorithm 11: Find-PM(bipartite graph G, $S \subseteq \mathbb{F}$)

```
11.1 Assume: $G$ has a perfect matching let $e = uv$ be an edge in $G$ if
     PM-tester($G[E-e]$, $S$) == Yes then
11.2     return Find-PM($G[E-e]$, $S$)
11.3 else
11.4     $M' \leftarrow$ Find-PM($G[V-\{u,v\}]$, $S$)
11.5     return $M' \cup \{e\}$
```

Theorem 8.12. *Let $|S| \geq n^3$. Given a bipartite graph G that contains some perfect matching, Algorithm 11 finds a perfect matching with probability at least $\frac{1}{2}$, and runs in time $O(m \cdot n^\omega)$.*

Proof. At each step, we call the tester once, and then recurse after either deleting an edge or two vertices. Thus, the number of total recursive steps inside Algorithm 11 is at most $\max\{m, n/2\} = m$, if the graph is connected. This gives a runtime of $O(m \cdot n^\omega)$. Moreover, at each step, the probability that the tester returns a wrong answer is at most $\frac{1}{n^2}$, so the PM-tester makes a mistake with probability at most $\frac{m}{n^2} \leq 1/2$, by a union bound. □

Observe that the algorithm assumes that G contains a perfect matching. We could simply run the PM-tester once on G at the beginning to check for a perfect matching, and then proceed as above. Or indeed, we could just run the algorithm regardless; if G has no perfect matching, there is no danger of this algorithm erronenously returning one.

Moreover, there are at least two ways reducing the error probability to $1/n^c$: we could increase the size of S to n^{s+3}, or we could repeat the above algorithm $c \log_2 n$ times. For the latter approach, the probability of not getting a perfect matching in all iterations is at most $(1/2)^{c \log_2 n} = \frac{1}{n^c}$. Hence we get correctness with high probability.

Corollary 8.13. *Given a bipartite graph G containing a perfect matching, there is an $O(m \cdot n^\omega \log n)$-time algorithm which finds a perfect matching with high probability.*

Exercise 8.14. Reduce the time-complexity of the basic version of Algorithm 11 from $O(m \cdot n^\omega)$ to $O(n \log n \cdot n^\omega)$.

8.3.1 The Algorithm of Rabin and Vazirani

Rewrite this section. How can we speed up the algorithm further? The improvement we give here is small, it only removes a logarithmic term from the algorithm you get from Exercise 8.14, but it has a nice idea that we want to emphasize. Again, we only focus on the bipartite case, and leave the general case for the reader. Also, we identify the nodes of both L and R with $[n]$, so that we can index the rows and columns of the Edmonds matrix using vertices in L and R respectively.

We can view Algorithm 11 as searching for a permutation π such that $M := \{i\pi(i) \mid i \in [n]\}$ is a perfect matching in G. Hence it picks a vertex, say $1 \in L$, and searches for a vertex $j \in R$ such that there is an edge $1j$ covering vertex 1, and also the remaining graph has a perfect matching. Interestingly, the remaining graph has an Edmonds matrix which is simple: it is simply the matrix $\mathbf{E}_{-1,-j}$, which is our notation from dropping row 1 and column j from $\mathbf{E}$.

Therefore, our task is simple: find a value j such that $1j$ is an edge in G, and also $\det(\mathbf{E}_{-1,-j})$ is a non-zero polynomial. Doing this naively would require n determinant computations, one for each j, and we'd be back to square one. But the smart observation is to recall Cramer's rule for the inverse for any matrix A:

Some jargon you may care for, or not:

$$A^{-1} = \frac{\text{adjugate}(A)}{\det(A)},$$

where adjugate(A) is the transpose of the *cofactor matrix* of A, given by

$$\text{cofactor}(A)_{p,q} := (-1)^{p+q} \det(A_{-p,-q}),$$

where $\det(A_{-p,-q})$ is also called a *minor* of A.

$$\left(A^{-1}\right)_{i,j} = \frac{(-1)^{i+j}\det(A_{-j,-i})}{\det(A)}. \tag{8.1}$$

Take the matrix $\widetilde{\mathbf{E}}$ obtained by substituting random values into the Edmonds matrix $\mathbf{E}$, and assume the set S and field $\mathbb{F}$ are of size at least n^{10}, say, so that all error probabilities are tiny. Compute its inverse in time $O(n^\omega)$, and use (8.1) to get all the values $\det(\widetilde{\mathbf{E}}_{-1,-j})$:

$$\det(\widetilde{\mathbf{E}}_{-1,-j}) = \left(\widetilde{\mathbf{E}}^{-1}\right)_{j,1} \times (-1)^{j+1} \times \det(\widetilde{\mathbf{E}})$$

using just n scalar multiplications. In fact, it suffices to find a non-zero $\widetilde{\mathbf{E}}^{-1}\big)_{j,1}$, which indicates that $\det(\widetilde{\mathbf{E}}_{-1,-j})$ is non-zero, and hence the corresponding $\det(\mathbf{E}_{-1,-j})$ (without the tilde) is a non-zero polynomial, so that $G[V \setminus \{1, j\}]$ has a perfect matching.

In summary, by computing one matrix inverse and n scalar multiplications, we can figure out one edge of the matching. Hence the runtime can be made $O(n \cdot n^\omega)$. Extending this to general graphs requires a bit more work; we refer to the Rabin and Vazirani paper for details. Also, Marcin Mucha's thesis has a very well-written introduction which discusses these details, and also gives the details of his improvement (with Sankowski) to $O(n^\omega)$ time.

8.3.2 The Polynomial Identity Testing (PIT) Problem

In *Polynomial Identity Testing* we are given a polynomial $P(x_1, x_2, \dots, x_n)$ over some field $\mathbb{F}$, and we want to test if it is identically zero or not. If P were written out explicitly as a list of monomials and their coefficients, this would not be a problem, since we could just check that all the coefficients are zero. But if P is represented implicitly, say as a determinant, then things get more tricky. A big question is whether polynomial identity testing (PIT) can be derandomized.

We don't know deterministic algorithms that given P can decide whether P is identically zero or not, in poly-time. How is P given, for this question? Even if P is given as an arithmetic circuit (a circuit whose gates are addition, subtraction and multiplication, and inputs are the variables and constants), it turns out that derandomizing PIT will result in surprising circuit lower bounds—for example, via a result of Kabanets and Impagliazzo. Derandomizing special cases of PIT can be done. For example, just the PIT instances that come from matchings can, however, be derandomized. This was shown in work by Jim Geelen and Nick Harvey, among others; however, but the runtime seems to get much worse.

8.4 Red-Blue Perfect Matchings

To illustrate the power of the algebraic approach, let us now consider the *red-blue matching* problem. We solve this problem using the algebraic approach and randomization. Interestingly, no determistic polynomial-time algorithm is currently known for this problem! Again, we consider bipartite graphs for simplicity.

Given a graph G where each edge is colored either red or blue, and an integer k, a k-red matching is a perfect matching in G which has exactly k red edges. The goal of the red-blue matching problem is to decide whether G has a k-red matching. (We focus on the decision version of the problem; the self-reducibility ideas used above can solve the search version.

To begin, let's solve the case when G has a *unique* red-blue matching with k red edges. Define the following $n \times n$ matrix:

$$\mathbf{M}_{i,j} = \begin{cases} 0 & \text{if } (i,j) \notin E, \\ 1 & \text{if } (i,j) \in E \text{ and colored blue,} \\ y & \text{if } (i,j) \in E \text{ and colored red.} \end{cases}$$

Claim 8.15. Let G have at most one perfect matching with k red edges. The determinant $\det(\mathbf{M})$ has a term of the form $c_k y^k$ if and only if G has a k-red matching.

Proof. Consider $p(y) := \det(\mathbf{M})$ as a univariate polynomial in the variable y. Again, using the Leibniz formula, the only way to get a non-zero term of the form $c_k y^k$ is if the graph has a k-red matching. And since we assumed that G has at most one such perfect matching, such a term cannot be cancelled out by other such matchings. □

The polynomial $p(y)$ has degree at most n, and hence we can recover it by Lagrangian interpolation. Indeed, we can choose $n+1$ distinct numbers $a_0, \dots, a_n$, and evaluate $p(a_0), \dots, p(a_n)$ by computing the determinant $\det(\mathbf{M})$ at $y = a_i$, for each i. These $n+1$ values are enough to determine the polynomial as follows:

$$p(y) = \sum_{i=1}^{n+1} p(a_i) \prod_{j \neq i} \left(\frac{x - a_j}{a_i - a_j} \right).$$

(E.g., see 451 lecture notes or Ryan's lecture notes.) Note this is a completely deterministic algorithm, so far.

8.4.1 Getting Rid of the Uniqueness Assumption

To extend to the case where G could have many k-red matchings, we can redefine the matrix as the following:

$$\mathbf{M}_{i,j} = \begin{cases} 0 & \text{if } (i,j) \notin E, \\ x_{ij} & \text{if } (i,j) \in E \text{ and colored blue,} \\ yx_{ij} & \text{if } (i,j) \in E \text{ and colored red.} \end{cases}$$

The determinant $\det(\mathbf{M})$ is now a polynomial in $m+1$ variables and degree at most $2n$. Writing

$$P(\mathbf{x}, y) = \sum_{i=0}^{n} y^i Q_i(\mathbf{x}),$$

where Q_i is a multilinear degree-n polynomial that corresponds to all the i-red matchings. If we set the $\mathbf{x}$ variables randomly (say, to values $x_{ij} = a_{ij}$) from a large enough set S, we get a polynomial $R(y) = P(\mathbf{a}, y)$ whose only variable is y. The coefficient of y^k in this polynomial is $Q_k(\mathbf{a})$, which is non-zero with high probability, by the Schwartz-Zippel lemma. Now we can again use interpolation to find out this coefficient, and decide the red-blue matching problem based on whether it is non-zero.

Multilinear just means that the degree of each variable in each monomial is at most one.

8.5 Matchings in Parallel, and the Isolation Lemma

One of the "killer applications" of the algebraic method for finding a perfect matching is that the approach extends to getting a (randomized) parallel algorithm as well. The basic idea is simple when there is a unique perfect matching. Indeed, computing the determinant can be done in parallel with poly-logarithmic depth and polynomial work. Hence, for each edge e we can run the PM-tester algorithm on G, and also on $G[E-e]$ to see if e belongs to this unique perfect matching; we output e if it does.

However, this approach fails when the graph has multiple perfect matchings. A fix for this problem was given by Mulmuley, Vazirani, and Vazirani [4] by adding further randomness! The approach is first extend the approach from §8.2 to find a ***minimum-weight*** perfect matching using the Tutte matrix and Schwartz-Zippel lemma, as long as the weights are polynomially bounded. (Exercise: solve this!) The trickier part is to show that assigning random polynomially-bounded weights to the edges of G causes it to have a ***unique*** minimum-weight perfect matching with high probability. Then this unique matching can also be found in parallel, as we outlined in the previous paragraph.

[4]

The proof showing that random weights result in a unique minimum-weight perfect matching is via a beautiful result called the *Isolation Lemma*. Let us give its simple elegant proof.

Theorem 8.16. *Consider a collection $\mathcal{F} = \{M_1, M_2, \dots,\}$ of sets over a universe E of size m. Assign a random weight to each elements of E, where the weights are drawn independently and uniformly from $\{1, \dots, 2m\}$. Then there exists a unique minimum-weight set with probability at least $\frac{1}{2}$.*

Proof. Call an element $e \in E$ "confused" if the weight of a minimum-weight set containing e is the same as the weight of a minimum-weight set *not* containing e. We claim that any specific element e is confused with probability at most $1/2m$. Observe is that there exists a confused element if and only if there are two minimum-weight sets, so using the claim and taking a union bound over all elements proves the theorem.

To prove the claim, make the random choices for all elements except e. Now the identity (and weight) of the minimum-weight set not containing e is determined; let its weight be W^-. Also, the identity (but *not* the weight) of the minimum-weight set containing e is determined. Its weight is not determined because the random choice for e's weight has not been made, so denote its weight by $W^+ + w_e$, where w_e is the random variable denoting the weight of e. Now e will be confused precisely if $W^- = W^+ + w_e$, i.e., if $w_e = W^- - W^+$. But since w_e is chosen uniformly at random from a set of size $2m$, this probability is at most $1/2m$, as claimed. □

We are using the ***principle of deferred decisions*** again.

It is remarkable the result does not depend on number of sets in $\mathcal{F}$, but only on the size of the universe. We also emphasize that the weights being drawn from a polynomially-sized set is what gives the claim its power: it is trivial to obtain a unique minimum-weight set if the weights are allowed to be in $\{1, \dots, 2^m\}$. (Exercise: how?) Finally, the proof strategy for the Isolation Lemma is versatile and worth remembering.

8.5.1 Towards Deterministic Algorithms

The question of finding perfect matchings deterministically in poly-logarithmic depth and polynomial work still remains open. Some recent work of Fenner, Gurjar, and XXXX, and of Svensson and Tarnawski has shown how to obtain poly-logarithmic depth and quasi-polynomial work. We will see some ideas in a HW.

8.6 A Matrix Scaling Approach

Talk about Matrix Scaling approach?

Part II

The Curse of Dimensionality, and Dimension Reduction

9
Concentration of Measure

Consider the following questions:

1. You distribute n tasks among n machines, by sending each task to a machine uniformly and independently at random: while any machine has unit expected load, what is the maximum load (i.e., the maximum number of tasks assigned to any machine)?

2. You want to estimate the bias p of a coin by repeatedly flipping it and then taking the sample mean. How many samples suffice to be within $\pm\varepsilon$ of the correct answer p with confidence $1-\delta$?

3. How many unit vectors can you choose in $\mathbb{R}^n$ that are almost orthonormal? I.e., they must satisfy $|\langle v_i, v_j\rangle| \leq \varepsilon$ for all $i \neq j$?

4. A n-dimensional hyercube has $N = 2^n$ nodes. Each node $i \in [N]$ contains a packet p_i, which is destined for node π_i, where π is a permutation. The routing happens in rounds. At each round, each packet traverses at most one edge, and each edge can transmit at most one packet. Find a routing policy where each packet reaches its destination in $O(n)$ rounds, regardless of the permutation π.

All these questions can be answered by the same basic tool, which goes by the name of ***Chernoff bounds*** or ***concentration inequalities*** or ***tail inequalities*** or ***concentration of measure***, or tens of other names. The basic question is simple: *if we have a real-valued function $f(X_1, X_2, \ldots, X_m)$ of several independent random variables X_i, such that it is "not too sensitive to each coordinate", how often does it deviate far from its mean?* To make it more concrete, consider this—

Given n independent random variables $X_1, \ldots, X_n$, each bounded in the interval $[0,1]$, let $S_n = \sum_{i=1}^n X_i$. What is

$$\Pr\left[S_n \notin (1 \pm \varepsilon)\mathbb{E}S_n\right]?$$

This question will turn out to have relations to convex geometry, to online learning, to many other areas. But of greatest interest to us, this question will solve many problems in algorithm analysis, including the above four. Let us see some basic results, and then give the answers to the four questions.

9.1 *Asymptotic Analysis*

We will be concerned with *non-asymptotic analysis*, i.e., the qualitative behavior of sums (and other Lipschitz functions) of finite number of (bounded) independent random variables. Before we begin that, a few words about the asymptotic analysis, which concerns the convergence of averages of infinite sequences of random variables.

Given a sequence of random variables $\{X_n\}$ and another random variable Y, the following two notions of convergence can be defined.

Definition 9.1 (Convergence in Probability)**.** $\{X_n\}$ converges in probability to Y if for every $\epsilon > 0$ we have

$$\lim_{n\to\infty} \mathbb{P}(|X_n - Y| > \epsilon) = 0 \tag{9.1}$$

This is denoted by $X_n \xrightarrow{\text{p}} Y$.

Definition 9.2 (Convergence in Distribution)**.** Let $F_X(.)$ denote the CDF of a random variable X. $\{X_n\}$ converges in distribution to Y if

$$\lim_{n\to\infty} F_{X_n}(t) = F_Y(t) \tag{9.2}$$

for all points t where the distribution function F_Y is continuous. This is denoted by $X_n \xrightarrow{\text{d}} Y$.

There are many results known here, and we only mention the two well-known results below. The ***weak law of large numbers*** states that the average of independent and identically distributed (i.i.d.) random variables converges in probability to their mean.

Theorem 9.3 (Weak law of large numbers)**.** *Let S_n denote the sum of n i.i.d. random variables, each with mean μ and variance $\sigma^2 < \infty$, then*

$$S_n/n \xrightarrow{p} \mu. \tag{9.3}$$

The ***central limit theorem*** tells us about the distribution of the sum of a large collection of i.i.d. random variables. Let $N(0,1)$ denote the standard normal variable with mean 0 and variance 1, whose probability density function is $f(x) = \frac{1}{\sqrt{2\pi}} \exp(-\frac{x^2}{2})$.

Theorem 9.4 (Central limit theorem)**.** *Let S_n denote the sum of n i.i.d. random variables, each with mean μ and variance $\sigma^2 < \infty$, then*

$$\frac{S_n - n\mu}{\sqrt{n}\sigma} \xrightarrow{d} N(0,1). \tag{9.4}$$

There are many powerful asymptotic results in the literature; see need to give references here.

9.2 *Non-Asymptotic Convergence Bounds*

Our focus will be on the behavior of finite sequences of random variables. The central question here will be: what is the chance of deviating far from the mean? Given an r.v. X with mean μ, and some deviation $\lambda > 0$, the quantity

$$\Pr[X \geq \mu + \lambda]$$

is called the *upper tail*, and the analogous quantity

$$\Pr[X \leq \mu - \lambda]$$

is the *lower tail*. We are interested in bounding these tails for various values of λ.

9.2.1 *Markov's inequality*

Most of our results will stem from the most basic of all results: ***Markov's inequality***. This inequality qualitatively generalizes that idea that a random variable cannot always be above its mean, and gives a bound on the upper tail.

Theorem 9.5 (Markov's Inequality)**.** *Let X be a non negative random variable and $\lambda > 0$, then*

$$\mathbb{P}(X \geq \lambda) \leq \frac{\mathbb{E}(X)}{\lambda} \tag{9.5}$$

With this in hand, we can start substituting various non-negative functions of random variables X to deduce interesting bounds. For instance, the next inequality looks at both the mean $\mu := \mathbb{E}X$ and the variance $\sigma^2 := \mathbb{E}[(X - \mu)^2]$ of a random variable, and bounds both the upper and lower tails.

9.2.2 *Chebychev's Inequality*

Theorem 9.6 (Chebychev's inequality)**.** *For any random variable X with mean μ and variance σ^2, we have*

$$\Pr[|X - \mu| \geq \lambda] \leq \frac{\sigma^2}{\lambda^2}.$$

Proof. Using Markov's inequality on the non-negative r.v. $Y = (X - \mu)^2$, we get

$$\Pr[Y \geq \lambda^2] \leq \frac{\mathbb{E}[Y]}{\lambda^2}.$$

The proof follows from $\Pr[Y \geq \lambda^2] = \Pr[|X - \mu| \geq \lambda]$. □

9.2.3 *Examples I*

Example 1 (Coin Flips): Let $X_1, X_2, \ldots, X_n$ be i.i.d. Bernoulli random variables with $\Pr[X_i = 0] = 1 - p$ and $\Pr[X_i = 1] = p$. (Im other words, these are the outcomes of independently flipping n coins, each with *bias* p.) Let $S_n := \sum_i^n X_i$ be the *number of heads*. Then S_n is distributed as a binomial random variable $\text{Bin}(n, p)$, with

$$\mathbb{E}[S_n] = np \qquad \text{and} \qquad \text{Var}[S_n] = np(1-p).$$

Recall that linearity of expectations for r.v.s X, Y means $\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$. For *independent* we have $\text{Var}[X+Y] = \text{Var}[X] + \text{Var}[Y]$.

Applying Markov's inequality for the upper tail gives

$$\Pr[S_n - pn \geq \beta n] \leq \frac{pn}{pn + \beta n} = \frac{1}{1 + (\beta/p)}.$$

So, for $p = 1/2$, this is $\frac{1}{1+2\beta} \approx 1 - O(\beta)$ for small values of $\beta > 0$. However, Chebychev's inequality gives a much tighter bound:

$$\Pr[|S_n - pn| \geq \beta n] \leq \frac{np(1-p)}{\beta^2 n^2} < \frac{p}{\beta^2 n}.$$

In particular, this already says that the sample mean S_n/n lies in the interval $p \pm \beta$ with probability at least $1 - \frac{p}{\beta^2 n}$. Equivalently, to get confidence $1 - \delta$, we just need to set $\delta \geq \frac{p}{\beta^2 n}$, i.e., take $n \geq \frac{p}{\beta^2 \delta}$. (We will see a better bound soon.)

Concretely, to get within an additive 1% error of the correct bias p with probability 99.9%, set $\beta = 0.01$ and $\delta = 0.001$, so taking $n \geq 10^7 \cdot p$ samples suffices.

Example 2 (Balls and Bins): Throw n balls uniformly at random and independently into n bins. Then for a fixed bin i, let L_i denote the number of balls in it. Observe that L_i is distributed as a $\text{Bin}(n, 1/n)$ random variable. Markov's inequality gives a bound on the probability that L_i deviates from its mean 1 by $\lambda \gg 1$ as

$$\Pr\left[L_i \geq 1 + \lambda\right] \leq \frac{1}{1+\lambda} \approx \frac{1}{\lambda}.$$

However, Chebychev's inequality gives a much tighter bound as

$$\Pr\Big[|L_i - 1| \geq \lambda\Big] \leq \frac{(1 - 1/n)}{\lambda^2} \approx \frac{1}{\lambda^2}.$$

So setting $\lambda = 2\sqrt{n}$ says that the probability of any fixed bin having more than $2\sqrt{n} + 1$ balls is at most $\frac{(1-1/n)}{4n}$. Now a union bound over all bins i means that, with probability at least $n \cdot \frac{(1-1/n)}{4n} \leq 1/4$, the load on every bin is at most $1 + 2\sqrt{n}$.

Doing this argument with Markov's inequality would give a trivial upper bound of $1 + 2n$ on the load. This is useless, since there are at most n balls, so the load can never be more than n.

Example 3 (Random Walk): Suppose we start at the origin and at each step move a unit distance either left or right uniformly randomly and independently. We can then ask about the behaviour of the final position after n steps. Each step (X_i) can be modelled as a *Rademacher* random variable with the following distribution.

$$X_i = \begin{cases} 1 & \text{w.p. } \frac{1}{2} \\ -1 & \text{w.p. } \frac{1}{2} \end{cases}$$

The position after n steps is given by $S_n = \sum_{i=1}^n X_i$, with mean and variance being $\mu = 0$ and $\sigma^2 = n$ respectively. Applying Chebyshev's inequality on S_n with deviation $\lambda = t\sigma = t\sqrt{n}$, we get

$$\Pr\left[S_n > t\sqrt{n}\right] \leq \frac{1}{t^2}. \tag{9.6}$$

We will soon see how to get a tighter tail bound.

9.2.4 *Higher-Order Moment Inequalities*

All the bounds in the examples above can be improved by using higher-order moments of the random variables. The idea is to use the same recipe as in Chebychev's inequality.

Theorem 9.7 ($2k^{th}$-Order Moment inequalities). *Let $k \in \mathbb{Z}_{\geq 0}$. For any random variable X having mean μ, and finite moments upto order $2k$, we have*

$$\Pr[|X - \mu| \geq \lambda] \leq \frac{\mathbb{E}((X-\mu)^{2k})}{\lambda^{2k}}.$$

Proof. The proof is exactly the same: using Markov's inequality on the non-negative r.v. $Y := (X - \mu)^{2k}$,

$$\Pr[|X - \mu| \geq \lambda] = \Pr[Y \geq \lambda^{2k}] \leq \frac{\mathbb{E}[Y]}{\lambda^{2k}}. \qquad \square$$

We can get stronger tail bounds for large values of k, however it becomes increasingly tedious to compute $E((X - \mu)^{2k})$ for the random variables of interest.

Example 3 (Random Walk, continued): If we consider the fourth moment of S_n:

$$\begin{aligned} \mathbb{E}\left[(S_n)^4\right] &= \mathbb{E}\Big[\sum_{i=1}^n X_i\Big] \\ &= \mathbb{E}\Big[\sum_i X_i^4 + 4\sum_{i<j} X_i^3 X_j + 6\sum_{i<j} X_i^2 X_j^2 + 12\sum_{i<j<k} X_i^2 X_j X_k + 24\sum_{i<j<k<l} X_i X_j X_k X_l\Big] \\ &= n + 6\binom{n}{2}, \end{aligned}$$

where we crucially used that the r.v.s are independent and mean-zero, hence terms like $X_i^3 X_j$, $X_i^2 X_j X_k$, and $X_i X_j X_k X_l$ all have mean zero. Now substituting this expectation in the fourth-order moment inequality, we get a stronger tail bound for $\lambda = t\sigma = t\sqrt{n}$.

$$\Pr\left[|S_n| \geq t\sqrt{n}\right] \leq \frac{\mathbb{E}\left[(S_n)^4\right]}{t^4 n^2} = \frac{n + 6\binom{n}{2}}{t^4 n^2} = \frac{\Theta(1)}{t^4}. \tag{9.7}$$

Compare this with the bound in (9.6).

9.2.5 *Digression: The Right Answer for Random Walks*

We can actually explicitly computing $\Pr(S_n = k)$ for sums of Rademacher random variables. Indeed, we just need to choose the positions for +1 steps, which means

$$\frac{\Pr[S_n = 2\lambda]}{\Pr[S_n = 0]} = \frac{\binom{n}{\frac{n}{2}+\lambda}}{\binom{n}{\frac{n}{2}}}.$$

For large n, we can use Stirling's formula $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$:

$$\frac{\Pr[S_n = 2\lambda]}{\Pr[S_n = 0]} \approx \frac{(\frac{n}{2})^{n/2}(\frac{n}{2})^{n/2}}{(\frac{n}{2}+\lambda)^{(n/2+\lambda)}(\frac{n}{2}-\lambda)^{(n/2-\lambda)}} = \frac{1}{(1+\frac{2\lambda}{n})^{\frac{n}{2}+\lambda}(1-\frac{2\lambda}{n})^{\frac{n}{2}-\lambda}}$$

If $\lambda \ll n$, then we can approximate $1 + k\frac{\lambda}{n}$ by $e^{k\frac{\lambda}{n}}$:

$$\frac{\Pr[S_n = 2\lambda]}{\Pr[S_n = 0]} \approx e^{\frac{-2\lambda}{n}(\frac{n}{2}+\lambda)} e^{\frac{2\lambda}{n}(\frac{n}{2}-\lambda)} = e^{\frac{-4\lambda^2}{n}}.$$

Finally, substituting $\lambda = t\sigma = t\sqrt{n}$, we get

$$\Pr[S_n = 2\lambda] \approx \Pr[S_n = 0] \cdot e^{-4t^2}.$$

This shows that most of the probability mass lies in the region $|S_n| \leq O(\sqrt{n})$, and drops off exponentially as we go further. And indeed, this is the bound we will derive next—we will get slightly weaker constants, but we will avoid these tedious approximations.

9.3 *Chernoff bounds, and Hoeffding's inequality*

The main bound of this section is a bit of a mouthful, but as Ryan O'Donnell says in his notes, you should memorize it "like a poem". I find it lies in a sweet spot: it is not difficult to remember, and still is very broadly applicable:

The provenance of these bounds is again quite complicated. There's Herman Chernoff's paper, which derives the corresponding inequality for i.i.d. Bernoulli random variables. Wassily Hoeffding gives the generalization for independent random variables all taking values in some bounded interval $[a, b]$. Though Chernoff attributes his result to another Herman, namely Herman Rubin. There's Harald Cramér (of the Cramér-Rao fame, not of Cramer's rule). And there's the bound by Sergei Bernstein, many years earlier, which is at least as strong...

Theorem 9.8 (Hoeffding's inequality). *Let $X_1, \ldots, X_n$ be n independent random variables taking values in $[0, 1]$. Let $S_n := \sum_{i=1}^n X_i$, with mean $\mu := \mathbb{E}[S_n] = \sum_i \mathbb{E}[X_i]$. Then for any $\beta \geq 0$ we have*

$$\textit{Upper tail}: \qquad \Pr\left[S_n \geq \mu(1+\beta)\right] \leq \exp\left\{-\frac{\beta^2\mu}{2+\beta}\right\}. \tag{9.8}$$

$$\textit{Lower tail}: \qquad \Pr\left[S_n \leq \mu(1-\beta)\right] \leq \exp\left\{-\frac{\beta^2\mu}{3}\right\}. \tag{9.9}$$

Proof. We only prove (9.8); the proof for (9.9) is similar. The idea is to use Markov's inequality not on the square or the fourth power, but on a function which is fast-growing enough so that we get tighter bounds, and "not too fast" so that we can control the errors. So we consider the *Laplace transform*, i.e., the function

$$x \mapsto e^{tx}$$

for some value $t > 0$ to be chosen carefully. Since this map is monotone,

$$\begin{aligned}\Pr[S_n \geq \mu(1+\beta)] &= \Pr[e^{tS_n} \geq e^{t\mu(1+\beta)}] \\ &\leq \frac{\mathbb{E}[e^{tS_n}]}{e^{t\mu(1+\beta)}} \quad \text{(using Markov's inequality)} \\ &= \frac{\prod_i \mathbb{E}[e^{tX_i}]}{e^{t\mu(1+\beta)}} \quad \text{(using independence)} \qquad (9.10)\end{aligned}$$

Bernoulli random variables: Assume that all the $X_i \in \{0,1\}$; we will remove this assumption later. Let the mean be $\mu_i = \mathbb{E}[X_i]$, so the *moment generating function* can be explicitly computed as

$$\mathbb{E}[e^{tX_i}] = 1 + \mu_i(e^t - 1) \leq \exp(\mu_i(e^t - 1)).$$

Substituting, we get

$$\begin{aligned}\Pr[S_n \geq \mu(1+\beta)] &\leq \frac{\prod_i \mathbb{E}[e^{tX_i}]}{e^{t\mu(1+\beta)}} \qquad (9.11) \\ &\leq \frac{\prod_i \exp(\mu_i(e^t - 1))}{e^{t\mu(1+\beta)}} \qquad (9.12) \\ &\leq \frac{\exp(\mu(e^t - 1))}{e^{t\mu(1+\beta)}} \qquad (\text{since } \mu = \sum_i \mu_i) \\ &= \exp(\mu(e^t - 1) - t\mu(1+\beta)). \qquad (9.13)\end{aligned}$$

Since this calculation holds for all positive t, and we want the tightest upper bound, we should minimize the expression (9.13). Setting the derivative w.r.t. t to zero gives $t = \ln(1+\beta)$ which is non-negative for $\beta \geq 0$.

$$\Pr[S_n \geq \mu(1+\beta)] \leq \left(\frac{e^\beta}{(1+\beta)^{1+\beta}}\right)^\mu. \qquad (9.14)$$

This bound on the upper tail is also one to be kept in mind; it often is useful when we are interested in large deviations where $\beta \gg 1$. One such example will be the load-balancing application with jobs and machines.

We're almost there: a slight simplification is that

$$\frac{\beta}{1+\beta/2} \leq \ln(1+\beta) \qquad (9.15)$$

for all $\beta \geq 0$, so

$$(9.13) = \exp(\mu(\beta - (1+\beta)\ln(1+\beta))) \overset{(9.15)}{\leq} \exp\left\{\frac{-\beta^2\mu}{2+\beta}\right\},$$

with the last inequality following from simple algebra. This proves the upper tail bound (9.8); a similar proof gives us the lower tail as well.

Removing the assumption that $X_i \in \{0,1\}$: If the r.v.s are not Bernoullis, then we define new Bernoulli r.v.s $Y_i \sim \text{Bernoulli}(\mu_i)$, which take value 0 with probability $1-\mu_i$, and value 1 with probability μ_i, so that $\mathbb{E}[X_i] = \mathbb{E}[Y_i]$. Note that $f(x) = e^{tx}$ is convex for every value of $t \geq 0$; hence the function $\ell(x) = (1-x)\cdot f(0) + x \cdot f(1)$ satisfies $f(x) \leq \ell(x)$ for all $x \in [0,1]$. Hence $\mathbb{E}[f(X_i)] \leq \mathbb{E}[\ell(X_i)]$; moreover $\ell(x)$ is a linear function so $\mathbb{E}[\ell(X_i)] = \ell(\mathbb{E}[X_i]) = \mathbb{E}[\ell(Y_i)]$, since X_i and Y_i have the same mean. Finally, $\ell(y) = f(y)$ for $y \in \{0,1\}$. Putting all this together,

$$\mathbb{E}[e^{tX_i}] \leq \mathbb{E}[e^{tY_i}] = 1 + \mu_i(e^t - 1) \leq \exp(\mu_i(e^t-1)),$$

so the step from (9.11) to (9.12) goes through again. This completes the proof of Theorem 9.8. $\square$

Since the proof has a few steps, let's take stock of what we did:

i. Markov's inequality on the function e^{tX},
ii. independence and linearity of expectations to break into e^{tX_i},
iii. reduction to the Bernoulli case $X_i \in \{0,1\}$,
iv. compute the MGF (moment generating function) $\mathbb{E}[e^{tX_i}]$,
v. choose t to minimize the resulting bound, and
vi. use convexity to argue that Bernoullis are the "worst case".

Do make sure you see why the bounds of Theorem 9.8 are impossible in general if we do not assume some kind of boundedness and independence.

You can get tail bounds for other functions of random variables by varying this template around; e.g., we will see an application for sums of independent normal (a.k.a. Gaussian) random variables in the next chapter.

9.3.1 *The Examples Again: New and Improved Bounds*

Example 1 (Coin Flips): Since each r.v. is a Bernoulli(p), the sum $S_n = \sum_i X_i$ has mean $\mu = np$, and hence

$$\Pr\left[|S_n - np| \geq \beta n\right] \leq \exp\Big(-\frac{\beta^2 n}{2p+\beta}\Big) \leq \exp\Big(-\frac{\beta^2 n}{2}\Big).$$

(For the second inequality, we use that the interesting settings have $p + \beta \leq 1$.) Hence, if $n \geq \frac{2\ln(1/\delta)}{\beta^2}$, the empirical average S_n/n is within an additve β of the bias p with probability at least $1-\delta$. This has an exponentially better dependence on $1/\delta$ than the bound we obtained from Chebychev's inequality.

This is asymptotically the correct answer: consider the problem where we have n coins, $n-1$ of them having bias $1/2$, and one having bias $1/2 + 2\beta$. We want to find the higher-bias coin. One way is to estimate the bias of each coin to within β with confidence $1 - \frac{1}{2n}$, using the procedure above—which takes $O(\log n/\varepsilon^2)$ flips per coin—and then take a union bound. It turns out any algorithm needs $\frac{\Omega(n \log n)}{\varepsilon^2}$ flips, so this the bound we have is tight. .

Example 2 (Load Balancing): Since the load L_i on any bin i behaves like $\mathrm{Bin}(n, 1/n)$, the expected load is 1. Now (9.8) says:

$$\Pr[L_i \geq 1 + \beta] \leq \exp\left(- \frac{\beta^2}{2 + \beta} \right).$$

If we set $\beta = \Theta(\log n)$, the probability of the load L_i being larger than $1 + \beta$ is at most $1/n^2$. Now taking a union bound over all bins, the probability that any bin receives at least $1 + \beta$ balls is at most $\frac{1}{n}$. I.e., the maximum load is $O(\log n)$ balls with high probability.

In fact, the correct answer is that the maximum load is $(1 + o(1))\frac{\ln n}{\ln \ln n}$ with high probability. For example, the proofs in cite show this. Getting this precise bound requires a bit more work, but we can get an asymptotically correct bound by using (9.14) instead, with a setting of $\beta = \frac{C \ln n}{\ln \ln n}$ with a large constant C.

Moreover, this shows that the asymmetry in the bounds (9.8) and (9.9) is essential. A first reaction would have been to believe our proof to be weak, and to hope for a better proof to get

The situation where $\beta \ll 1$ is often called the *Gaussian regime*, since the bound on the upper tail behaves like $\exp(-\beta^2\mu)$. In other cases, the upper tail bound behaves like $\exp(-\beta\mu)$, and is said to be the *Poisson regime*.

$$\Pr[S_n \geq (1 + \beta)\mu] \leq \exp(-\beta^2 \mu / c)$$

for some constant $c > 0$, for all values of β. This is not possible, however, because it would imply a max-load of $\Theta(\sqrt{\log n})$ with high probability.

Example 3 (Random Walk): In this case, the variables are $[-1, 1]$ valued, and hence we cannot apply the bounds from Theorem 9.8 directly. But define $Y_i = \frac{1+X_i}{2}$ to get Bernoulli(1/2) variables, and define $T_n = \sum_{i=1}^n Y_i$. Since $T_n = S_n/2 + n/2$,

In general, if X_i takes values in $[a, b]$, we can define $Y_i := \frac{X_i - a}{b - a}$ and then use Theorem 9.8.

$$\begin{aligned} \Pr\left[|S_n| \geq t\sqrt{n}\right] &= \Pr\left[|T_n - n/2| \geq (t/2)\sqrt{n}\right] \\ &\leq 2\exp\left\{ - \frac{(t^2/n) \cdot (n/2)}{2 + \sqrt{t/n}} \right\} && \text{using (9.8)} \\ &\leq 2\exp(-t^2/6). \end{aligned}$$

Recall from §9.2.5 that the tail bound of $\approx \exp(-t^2/O(1))$ is indeed in the right ballpark.

9.4 *Other concentration bounds*

Many of the extensions address the various assumptions of Theorem 9.8: that the variables are bounded, that they are independent, and that the function S_n is the ***sum*** of these r.v.s. Add details and refs to this section.

But before we move on, let us give the bound that Sergei Bernstein gave in the 1920s: it uses knowledge about the variance of the random variable to get a potentially sharper bound than Theorem 9.8

Theorem 9.9 (Bernstein's inequality). *Consider n independent random variables $X_1, \ldots, X_n$ with $|X_i - \mathbb{E}[X_i]| \le 1$ for each i. Let $S_n := \sum_i X_i$ have mean μ and variance σ^2. Then for any $\lambda \ge 0$ we have*

$$\Pr[|S_n - \mu| \ge \lambda] \le 2\exp\left(-\frac{\lambda^2}{2\sigma^2 + 2\lambda/3}\right).$$

9.4.1 Mildly Correlated Variables

The only place we used independence in the proof of Theorem 9.8 was in (9.10). So if we have some set of r.v.s where this inequality holds even without independence, the proof can proceed unchanged. Indeed, one such case is when the r.v.s are ***negatively correlated***. Loosely speaking, this means that if some variables are "high" then it makes more likely for the other variables to be "low". Formally, $X_1, \ldots, X_n$ are ***negatively associated*** if for all disjoint sets A, B and for all monotone increasing functions f, g, we have

$$\mathbb{E}[f(X_i : i \in A) \cdot g(X_j : j \in B)] \le \mathbb{E}[f(X_i : i \in A)] \cdot \mathbb{E}[g(X_j : j \in B)].$$

We can use this in the step (9.10), since the function e^{tx} is monotone increasing for $t > 0$.

Negative association arises in many settings: say we want to choose a subset S of k items out of a universe of size n, and let $X_i = \mathbf{1}_{i \in S}$ be the indicator for whether the i^{th} item is selected. The variables $X_1, \ldots, X_n$ are clearly not independent, but they are negatively associated.

9.4.2 Martingales

A different and powerful set of results can be obtained when we stop considering random variables are not independent, but allow variables X_j to take on values that depend on the past choices $X_1, X_2, \ldots, X_{j-1}$ but in a controlled way. One powerful formalization is the notion of a ***martingale***. A *martingale difference sequence* is a sequence of r.v.s $Y_1, Y_2, \ldots, Y_n$, such that $\mathbb{E}[Y_i \mid Y_1, \ldots, Y_{i-1}] = 0$ for each i. (This is true for mean-zero independent r.v.s, but may be true in other settings too.)

Theorem 9.10 (Hoeffding-Azuma inequality). *Let $Y_1, Y_2, \ldots, Y_n$ be a martingale difference sequence with $|Y_i| \le c_i$ for each i, for constants c_i.*

Then for any $t \geq 0$,

$$\Pr\left[\left| \sum_{i=1}^{n} Y_i \right| \geq \lambda \right] \leq 2 \exp\left(-\frac{\lambda^2}{2\sum_{i=1}^{n} c_i^2} \right).$$

For instance, applying the Azuma-Hoeffding bounds to the random walk in Example 3, where each Y_i is a Rademacher r.v. gives $\Pr[|S_n| \geq t\sqrt{n}] \leq 2e^{-t^2/8}$, which is very similar to the bounds we derived above. But we can also consider, e.g., a "bounded" random walk that starts at the origin, say, and stops whenever it reaches either $-\ell$ or $+r$. In this case, the step size $Y_i = 0$ with unit probability if $\sum_{j=1}^{i-1} Y_j \in \{-\ell, r\}$, else it is $\{\pm 1\}$ independently and uniformly at random.

9.4.3 Going Beyond Sums of Random Variables

The Azuma-Hoeffding inequality can be used to bound functions of $X_1, \ldots, X_n$ other than their sum—and there are many other bounds for more general classes of functions. In all these cases we want any single variable to affect the function only in a limited way—i.e., the function should be Lipschitz. One popular packaging was given by Colin McDiarmid:

Theorem 9.11 (McDiarmid's inequality)**.** *Consider n independent r.v.s $X_1, \ldots, X_n$, with X_i taking values in a set A_i for each i, and a function $f : \prod A_i \to \mathbb{R}$ satisfying $|f(x) - f(x')| \leq c_i$ whenever x and x' differ only in the i^{th} coordinate. Let $\mu := \mathbb{E}[f(X_1, \ldots, X_n)]$ be the expected value of the random variable $f(\overline{X})$. Then for any non-negative β,*

$$\textit{Upper tail}: \qquad \Pr[f(X) \geq \mu(1+\beta)] \leq \exp\left(-\frac{2\mu^2\beta^2}{\sum_i c_i^2} \right)$$

$$\textit{Lower tail}: \qquad \Pr[f(X) \leq \mu(1-\beta)] \leq \exp\left(-\frac{2\mu^2\beta^2}{\sum_i c_i^2} \right)$$

This inequality does not assume very much about the function, except it being c_i-Lipschitz in the i^{th} coordinate; hence we can also use this to the truncated random walk example above, or for many other applications.

9.4.4 Moment Bounds vs. Chernoff-style Bounds

One may ask how moment bounds relate to Chernoff-Hoeffding bounds: Philips and Nelson [1] showed that bounds obtained using this approach of bounding the moment-generating function are never stronger than moment bounds:

[1]

Theorem 9.12. *Consider n independent random variables $X_1, \dots, X_n$, each with mean 0. Let $S_n = \sum X_i$. Then*

$$\Pr[S_n \geq \lambda] \leq \min_{k \geq 0} \frac{\mathbb{E}[X^k]}{\lambda^k} \leq \inf_{t \geq 0} \frac{\mathbb{E}[e^{tX}]}{e^{t\lambda}}$$

9.4.5 *Matrix-Valued Random Variables*

Finally, an important line of research considers concentration for vector-valued and matrix valued functions of independent (and mildly dependent) r.v.s. One object that we will see in a homework, and also in later applications, is the matrix-valued case: here the notation $A \succeq 0$ means the matrix is positive-semidefinite (i.e., all its eigenvalues are non-negative), and $A \succeq B$ means $A - B \succeq 0$. See, e.g., the lecture notes by Joel Tropp!

Theorem 9.13 (Matrix Chernoff bounds)**.** *Consider n independent symmetric matrices $X_1, \dots, X_n$ of dimension d. Moreover, $I \succeq X_i \succeq 0$ for each i, i.e., the eigenvalues of each matrix are between 0 and 1. If $\mu_{max} := \lambda_{max}(\sum \mathbb{E}[X_i])$ is the largest eigenvalue of their expected sum, then*

$$\Pr\left[\lambda_{max}\left(\sum X_i\right) \geq \mu_{max} + \gamma\right] \leq d \, \exp\left(-\frac{\gamma^2}{2\mu_{max} + \gamma}\right).$$

As an example, if we are throwing n balls into n bins, then we can let matrix X_i have a single 1 at position (j, j) if the i^{th} ball falls into bin j, and zeros elsewhere. Now the sum of these matrices has the loads of the bins on the diagonal, and the maximum eigenvalue is precisely the highest load. This bound therefore gives that the probability of a bin with load $1 + \gamma$ is at most $n \cdot e^{\gamma^2/(2+\gamma)}$—again implying a maximum load of $O(\log n)$ with high probability.

But we can use this for a lot more than just diagonal matrices (which can be reasoned about using the scalar-valued Chernoff bounds, plus the naïve union bound). Indeed, we can sample edges of a graph at random, and then talk about the eigenvalues of the resulting adjacency matrix (or more interestingly, of the resulting Laplacian matrix) using these bounds. We will discuss this in a later chapter.

9.5 *Application: Oblivious Routing on the Hypercube*

Now we return to fourth application mentioned at the beginning of the chapter. (The first two applications have already been considered above, the third will be covered as a homework problem.)

The setting is the following: we are given the d-dimensional hypercube Q_d, with $n = 2^d$ vertices. We have $n = 2^d$ vertices, each labeled with a d-bit vector. Each vertex i has a single packet (which we also call packet i), destined for vertex $\pi(i)$, where π is a permutation on the nodes $[n]$.

Packets move in synchronous rounds. Each edge is bi-directed, and at most one packet can cross each directed edge in each round. Moreover, each packet can cross at most one edge per round. So if $uv \in E(Q_d)$, one packet can cross from u to v, and one from v to u, in a round. Each edge e has an associated queue; if several packets want to cross e in the same round, only one can crosse, and the rest wait in the queue, and try again the next round. (So each node has d queues, one for each edge leaving it.) We assume the queues are allowed to grow to arbitrary size (though one can also show queue length bounds in the algorithm below). The goal is to get a simple routing scheme that delivers the packets in $O(d)$ rounds.

One natural proposal is the ***bit-fixing routing*** scheme: each packet i looks at its current position u, finds the first bit position where u differs from $\pi(i)$, and flips the bit (which corresponds to traversing an edge out of u). For example:

$$0001010 \to 1001010 \to 1101010 \to 1100010 \to 1100011.$$

However, this proposal can create "congestion hotspots" in the network, and therefore delay some packets by $2^{\Omega(d)}$: see example on Piazza. In fact, it turns out any deterministic *oblivious* strategy (that does not depend on the actual sources and destinations) must have a delay of $\Omega(\sqrt{2^d/d})$ rounds.

9.5.1 *A Randomized Algorithm...*

Here's a great randomized strategy, due to Les Valiant, and to Valiant and Brebner. It requires no centralized control, and is optimal in the sense of requiring $O(d)$ rounds (with high probability) on any permutation.

Valiant (1982)

> Each node i picks a randomized midpoint R_i independently and uniformly from $[n]$: it sends its packet to R_i. Then after $5d$ rounds have elapsed, the packets proceed to their final destinations $\pi(i)$. All routing is done using bit-fixing.

9.5.2 *... and its Analysis*

Theorem 9.14. *The random midpoint algorithm above succeeds in delivering the packets in at most $10d$ rounds, with probability at least $1 - \frac{2}{n}$.*

Proof. We only prove that all packets reach their midpoints by time $5d$, with high probability. The argument for the second phase is then identical. Let P_i be the bit-fixing path from i to the midpoint R_i. The following claim is left as an exercise:

Claim 9.15. Any two paths P_i and P_j intersect in one contiguous segment.

Since R_i is chosen uniformly at random from $\{0,1\}^d$, the labels of i and R_i differ in $d/2$ bits in expectation. Hence P_i has expected length $d/2$. There are $d2^d = dn$ (directed) edges, and all $n = 2^d$ paths behave symmetrically, so the expected number of paths P_j using any edge e is $\frac{n \cdot d/2}{dn} = 1/2$. Now define

$$S(i) = \{j \mid \text{ path } P_j \text{ shares an edge with } P_i\}.$$

Claim 9.16. Packet i reaches the midpoint by time at most $d + |S(i)|$.

Proof. This is a clever, cute argument. Let $P_i = \langle e_1, e_2, \dots, e_\ell \rangle$. Say that a packet in $\{i\} \cup S(i)$ that wants to cross edge e_k at the start of round t has *lag* $t - k$. Hence packet i reaches R_i at time equal to the length of P_i, plus its lag just before it crosses the last edge e_ℓ. We now show that if i's lag increases from L to $L+1$ at some point, then some packet leaves the path P_i (forever, because of Claim 9.15) with final lag L at some future point in time. Indeed, if i's lag increased from L to $L+1$ at edge e_k, then some packet crossed e_k instead and its lag was L. Now either this packet leaves path P_i with lag L, or else it is delayed at some subsequent edge on P_i (and the edge traversing that edge has lag L).

Hence each increase in i's lag $L \to L+1$ can be charged to some packet in $S(i)$ that eventually leaves P_i with lag L; this bounds the maximum delay by $|P_i| + |S(i)| \le d + |S(i)|$. □

Claim 9.17. $\Pr[|S(i)| \ge 4d] \le e^{-2d}$.

Proof. If X_{ij} is the indicator of the event that P_i and P_j intersect, then $|S(i)| = \sum_{j \ne i} X_{ij}$, i.e., it is a sum of a collection of independent $\{0,1\}$-valued random variables. Now conditioned on any choice of P_i (which is of length at most d), the expected number of paths using each edge in it is at most $1/2$, so the conditional expectation of $S(i)$ is at most $d/2$. Since this holds for any choice of P_i, the unconditional expectation $\mu = \mathbb{E}[S(i)]$ is also at most $d/2$. Now apply the Chernoff bound to $S(i)$ with $\beta\mu = 4d - \mu$ and $\mu \le d/2$ to get

$$\Pr[|S(i)| \ge 4d] \le \exp\left\{ -\frac{(4d-\mu)^2}{2\mu + (4d-\mu)} \right\} \le e^{-2d}.$$

Note that we could apply the bound even though the variables X_{ij} were not i.i.d., and moreover we did not need estimates for $\mathbb{E}[X_{ij}]$, just an upper bound for their expected sum. □

Now applying a union bound over all $n = 2^d$ packets i means that all n packets reach their midpoints within $d + 4d$ steps with probability $1 - 2^d \cdot e^{-2d} \geq 1 - e^{-d} \geq 1 - 1/n$. Similarly, the second phase has a probability at most $1/n$ of failing to complete in $5d$ steps, completing the proof. $\square$

9.5.3 *Graph Sparsification*

10 Dimension Reduction and the JL Lemma

For a set of n points $\{x_1, x_2, \ldots, x_n\}$ in $\mathbb{R}^D$, can we map them into some lower dimensional space $\mathbb{R}^k$ and still maintain the Euclidean distances between them? We can always take $k \leq n-1$, since any set of n points lies on a $n-1$-dimensional subspace. And this is (existentially) tight, e.g., if $x_2 - x_1, x_3 - x_1, \ldots, x_n - x_1$ are all orthogonal vectors.

But what if we were fine with distances being approximately preserved? There can only be k orthogonal unit vectors in $\mathbb{R}^k$, but there are as many as $\exp(c\varepsilon^2 k)$ unit vectors which are ε-orthogonal—i.e., whose mutual inner products all lie in $[-\varepsilon, \varepsilon]$. Near-orthogonality allows us to pack exponentially more vectors! (Indeed, we will see this in a homework exercise.)

This near-orthogonality of the unit vectors means that distances are also approximately preserved. Indeed, for any two $a, b \in \mathbb{R}^k$,

$$\|a-b\|_2^2 = \langle a-b, a-b\rangle = \langle a,a\rangle + \langle b,b\rangle - 2\langle a,b\rangle = \|a\|_2^2 + \|b\|_2^2 - 2\langle a,b\rangle,$$

so the squared Euclidean distance between any pair of the points defined by these ε-orthogonal vectors falls in the range $2(1 \pm \varepsilon)$. So, if we wanted n points at exactly the same (Euclidean) distance from each other, we would need $n-1$ dimensions. (Think of a triangle in 2-dims.) But if we wanted to pack in n points which were at distance $(1 \pm \varepsilon)$ from each other, we could pack them into

Having $n \geq \exp(c\varepsilon^2 k)$ vectors in d dimensions means the dimension is $k = O(\log n/\varepsilon^2)$.

$$k = O\left(\frac{\log n}{\varepsilon^2}\right)$$

dimensions.

10.1 The Johnson Lindenstrauss lemma

The Johnson Lindenstrauss "flattening" lemma says that such a claim is true not just for equidistant points, but for any set of n points in Euclidean space:

Lemma 10.1. *Let $\varepsilon \in (0, 1/2)$. Given any set of points $X = \{x_1, x_2, \dots, x_n\}$ in $\mathbb{R}^D$, there exists a map $A : \mathbb{R}^D \to \mathbb{R}^k$ with $k = O\left(\frac{\log n}{\varepsilon^2}\right)$ such that*

$$1 - \varepsilon \leq \frac{\|A(x_i) - A(x_j)\|_2^2}{\|x_i - x_j\|_2^2} \leq 1 + \varepsilon.$$

Moreover, such a map can be computed in expected $\text{poly}(n, D, 1/\varepsilon)$ *time.*

Note that the target dimension k is independent of the original dimension D, and depends only on the number of points n and the accuracy parameter ε.

It is not difficult to show that we need at least $\Omega(\log n)$ dimensions in such a result, using a packing argument. Noga Alon showed a lower bound of $\Omega\left(\frac{\log n}{\varepsilon^2 \log 1/\varepsilon}\right)$, and then Kasper Green Larson and Jelani Nelson showed a tight and matching lower bound of $\Omega\left(\frac{\log n}{\varepsilon^2}\right)$ dimensions for any dimensionality reduction scheme from n dimensions that preserves pairwise distances.

Given n points with Euclidean distances in $(1 \pm \varepsilon)$, the balls of radius $\frac{1-\varepsilon}{2}$ around these points must be mutually disjoint, by the minimum distance, and they are contained within a ball of radius $(1 + \varepsilon) + \frac{1-\varepsilon}{2}$ around x_0. Since volumes of balls in $\mathbb{R}^k$ of radius r behave like $c_k r^k$, we have

$$n \cdot c_k \left(\frac{1-\varepsilon}{2}\right)^k \leq c_k \left(\frac{3+\varepsilon}{2}\right)^k$$

or $k \geq \Omega(\log n)$ for $\varepsilon \leq 1/2$.

Alon (2003)

Larson and Nelson (2017)

The JL Lemma was first considered in the area of metric embeddings, for applications like fast near-neighbor searching; today we use it to speed up algorithms for problems like spectral sparsification of graphs, and solving linear programs fast.

10.2 The Construction

The JL lemma is pretty surprising, but the construction of the map is perhaps even more surprising: it is a super-simple randomized construction. Let M be a $k \times D$ matrix, such that every entry of M is filled with an i.i.d. draw from a standard normal $N(0, 1)$ distribution (a.k.a. the "Gaussian" distribution). For $x \in \mathbb{R}^D$, define

$$A(x) = \frac{1}{\sqrt{k}} Mx.$$

That's it. You hit the vector x with a Gaussian matrix M, and scale it down by $\sqrt{k}$. That's the map A.

Since $A(x)$ is a linear map and satisfies $\alpha A(x) + \beta A(y) = A(\alpha x + \beta y)$, it is enough to show the following lemma:

Lemma 10.2. *[Distributional Johnson-Lindenstrauss] Let $\varepsilon \in (0, 1/2)$. If A is constructed as above with $k = c\varepsilon^{-2} \log \delta^{-1}$, and $x \in \mathbb{R}^D$ is a unit vector, then*

$$\Pr[\ \|A(x)\|_2^2 \in 1 \pm \varepsilon] \geq 1 - \delta.$$

To prove Lemma 10.1, set $\delta = 1/n^2$, and hence $k = O(\varepsilon^{-2} \log n)$. Now for each $x_i, x_j \in X$, use linearity of $A(\cdot)$ to infer

$$\frac{\left\|A(x_i) - A(x_j)\right\|^2}{\left\|x_i - x_j\right\|^2} = \frac{\left\|A(x_i - x_j)\right\|^2}{\left\|x_i - x_j\right\|^2} = \left\|A(v_{ij})\right\|^2 \in (1 \pm \varepsilon)$$

with probability at least $1 - 1/n^2$, where v_{ij} is the unit vector in the direction of $x_i - x_j$. By a union bound, all $\binom{n}{2}$ pairs of distances in $\binom{X}{2}$ are maintained with probability at least $1 - \binom{n}{2}\frac{1}{n^2} \geq 1/2$. A few comments about this construction:

- The above proof shows not only the existence of a good map, we also get that a random map as above works with constant probability! In other words, a Monte-Carlo randomized algorithm for dimension reduction. (Since we can efficiently check that the distances are preserved to within the prescribed bounds, we can convert this into a Las Vegas algorithm.) Or we can also get deterministic algorithms: see here.
- The algorithm (at least the Monte Carlo version) is ***data-oblivious***: it does not even look at the set of points X: it works for any set X with high probability. Hence, we can pick this map A before the points in X arrive.

10.3 Intuition for the Distributional JL Lemma

Let us recall some basic facts about Gaussian distributions. The probability density function for the Gaussian $N(\mu, \sigma^2)$ is

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{(x-\mu)^2}{2\sigma^2}}.$$

We also use the following; the proof just needs some elbow grease.

The fact that the means and the variances take on the claimed values should not be surprising; this is true for all r.v.s. The surprising part is that the resulting variables are also Gaussians.

Proposition 10.3. *If $G_1 \sim N(\mu_1, \sigma_1^2)$ and $G_2 \sim N(\mu_2, \sigma_2^2)$ are independent, then for $c \in \mathbb{R}$,*

$$c\, G_1 \sim N(c\mu_1, c^2\, \sigma_1^2) \tag{10.1}$$

$$G_1 + G_2 \sim N(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2). \tag{10.2}$$

Now, here's the main idea in the proof of Lemma 10.2. Imagine that the vector x is the elementary unit vector $e_1 = (1, 0, \ldots, 0)$. Then $M\, e_1$ is just the first column of M, which is a vector with independent and identical Gaussian values.

$$M\, e_1 = \begin{bmatrix} G_{1,1} & G_{1,2} & \cdots & G_{1,D} \\ G_{2,1} & G_{2,2} & \cdots & G_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ G_{k,1} & G_{k,2} & \cdots & G_{k,D} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} G_{1,1} \\ G_{2,1} \\ \vdots \\ G_{k,1} \end{bmatrix}.$$

$A(x)$ is a scaling-down of this vector by $\sqrt{k}$: every entry in this random vector $A(x) = A(e_1)$ is distributed as

$$1/\sqrt{k} \cdot N(0, 1) = N(0, 1/k) \qquad \text{(by (10.1))}.$$

Thus, the expected squared length of $A(x) = A(e_1)$ is

If G has mean μ and variance σ^2, then $\mathbb{E}[G^2] = \mathrm{Var}[G] + \mathbb{E}[G]^2 = \sigma^2 + \mu^2$.

$$\mathbb{E}\left[\|A(x)\|^2\right] = \mathbb{E}\left[\sum_{i=1}^{k} A(x)_i^2\right] = \sum_{i=1}^{k} \mathbb{E}\left[A(x)_i^2\right] = \sum_{i=1}^{k} \frac{1}{k} = 1.$$

So the expectation of $\|A(x)\|^2$ is 1; the heart is in the right place! Now to show that $\|A(x)\|^2$ does not deviate too much from the mean—i.e., to show a concentration result. Indeed, $\|A(x)\|^2$ is a sum of independent $N(0, 1/k)^2$ random variables, so if these $N(0, 1/k)^2$ variables were bounded, we would be done by the Chernoff bounds of the previous chapter. Sadly, they are not. However, their tails are fairly "thin", so if we squint hard enough, these random variables can be viewed as "pretty much bounded", and the Chernoff bounds can be used.

Of course this is very vague and imprecise. Indeed, the Laplace distribution with density function $f(x) \propto e^{-\lambda|x|}$ for $x \in \mathbb{R}$ also has pretty thin tails—"exponential tails". But using a matrix with Laplace entries does not work the same, no matter how hard we squint. It turns out you need the entries of M, the matrix used to define $A(x)$, to have "sub-Gaussian tails". The Gaussian entries have precisely this property.

We now make all this precise, and also remove the assumption that the vector $x = e_1$. In fact, we do this in two ways. First we give a direct proof: it has several steps, but each step is elementary, and you are mostly following your nose. The second proof formally defines the notion of sub-Gaussian random variables, and builds some general machinery for concentration bounds.

10.4 The Direct Proof of Lemma 10.2

Recall that we want to argue about the squared length of $A(x) \in \mathbb{R}^k$, where $A(x) = \frac{1}{\sqrt{k}} Mx$, and x is a unit vector. To start off, observe that the i^{th} coordinate of the vector Mx is the inner product of a row of M with the vector x. This is distributed as

$$Y_i \sim \langle G_1, G_2, \ldots, G_D \rangle \cdot x = \sum_j x_j G_j$$

where the G_j's are the i.i.d. $N(0, 1)$ r.v.s on the i^{th} row of M. Now Proposition 10.3 tells us that $Y_i \sim N(0, x_1^2 + x_2^2 + \ldots + x_D^2)$. Since x is a unit length vector, we get

$$Y_i \sim N(0, 1).$$

So, each of the k coordinates of Mx behaves just like an independent Gaussian!

10.4.1 *The Expectation*

Given the observation above, the squared length of $A(x) = \frac{1}{\sqrt{k}} Mx$ is

$$Z := \|A(z)\|^2 = \sum_{i=1}^{k} \frac{1}{k} \cdot Y_i^2$$

where each $Y_i \sim N(0,1)$, independent of the others. And since $\mathbb{E}[Y_i^2] = \mathrm{Var}(Y_i) + \mathbb{E}[Y_i]^2 = 1$, we get $\mathbb{E}[Z] = 1$.

10.4.2 *Concentration about the Mean*

Now to show that Z does not deviate too much from 1. And Z is the sum of a bunch of independent and identical random variables. Let's start down the usual path for a Chernoff bound, for the upper tail, say:

The easy way out is to observe that the squares of Gaussians are chi-squared r.v.s, the sum of k of them is χ^2 ***with k degrees of freedom***, and the internet conveniently has tail bounds for these things. But if you don't recall these facts, and don't have internet connectivity and cannot check Wikipedia, things are not that difficult.

$$\Pr[Z \geq 1+\varepsilon] \leq \Pr[e^{tkZ} \geq e^{tk(1+\varepsilon)}] \leq \mathbb{E}[e^{tkZ}]/e^{tk(1+\varepsilon)} \tag{10.3}$$

$$= \prod_i \left(\mathbb{E}[e^{tY_i^2}]/e^{t(1+\varepsilon)} \right) \tag{10.4}$$

for every $t > 0$. Now $\mathbb{E}[e^{tG^2}]$, the moment-generating function for G^2, where $G \sim N(0,1)$ is easy to calculate for $t < 1/2$:

$$\frac{1}{\sqrt{2\pi}} \int_{g \in \mathbb{R}} e^{tg^2} e^{-g^2/2} dg = \frac{1}{\sqrt{2\pi}} \int_{z \in \mathbb{R}} e^{-z^2/2} \frac{dz}{\sqrt{1-2t}} = \frac{1}{\sqrt{1-2t}}. \tag{10.5}$$

So our current bound on the upper tail is that for all $t \in (0, 1/2)$ we have

$$\Pr[Z \geq (1+\varepsilon)] \leq \left(\frac{1}{e^{t(1+\varepsilon)}\sqrt{1-2t}} \right)^k.$$

Let's just focus on part of this expression:

$$\left(\frac{1}{e^t \sqrt{1-2t}} \right) = \exp\left(-t - \frac{1}{2}\log(1-2t) \right) \tag{10.6}$$

$$= \exp\left((2t)^2/4 + (2t)^3/6 + \cdots \right) \tag{10.7}$$

$$\leq \exp\left(t^2(1 + 2t + 2t^2 + \cdots) \right) \tag{10.8}$$

$$= \exp(t^2/(1-2t)).$$

Plugging this back, we get

$$\Pr[Z \geq (1+\varepsilon)] \leq \left(\frac{1}{e^{t(1+\varepsilon)}\sqrt{1-2t}} \right)^k$$

$$\leq \exp(kt^2/(1-2t) - kt\varepsilon) \leq e^{-k\varepsilon^2/8},$$

if we set $t = \varepsilon/4$ and use the fact that $1 - 2t \geq 1/2$ for $\varepsilon \leq 1/2$. (Note: this setting of t also satisfies $t \in (0, 1/2)$, which we needed from our previous calculations.)

Almost done: let's take stock of the situation. We observed that $\|A(x)\|_2^2$ was distributed like an average of squares of Gaussians, and by a Chernoff-like calculation we proved that

$$\Pr[\|A(x)\|_2^2 > 1 + \varepsilon] \leq \exp(-k\varepsilon^2/8) \leq \delta/2$$

for $k = \frac{8}{\varepsilon^2} \ln \frac{2}{\delta}$. A similar calculation bounds the lower tail, and finishes the proof of Lemma 10.2.

The JL Lemma was first proved by Bill Johnson and Joram Lindenstrauss. There have been several proofs after theirs, usually trying to tighten their results, or simplify the algorithm/proof (see citations in some of the newer papers): the proof above is some combinations of those by Piotr Indyk and Rajeev Motwani, and Sanjoy Dasgupta and myself.

Johnson and Lindenstrauss (1982)

Indyk and Motwani (1998)

Dasgupta and Gupta (2004)

10.5 Subgaussian Random Variables

While Gaussians have all kinds of nice properties, they are real-valued and hence require more randomness to generate. What other classes of r.v.s could give us bounds that are comparable? E.g., what about setting each $M_{ij} \in_R \{-1, +1\}$?

A random sign is also called a ***Rademacher random variable***, the name Bernoulli being already taken for a random bit in $\{0, 1\}$.

It turns out that Rademacher r.v.s also suffice, and we can prove this with some effort. But instead of giving a proof from first principles, let us abstract out the process of proving Chernoff-like bounds, and give a proof using this abstraction.

Recall the basic principle of a Chernoff bound: to bound the upper tail of an r.v. V with mean μ, we can choose any $t \geq 0$ to get

$$\Pr[V - \mu \geq \lambda] = \Pr[e^{t(V-\mu)} \geq e^{t\lambda}] \leq \mathbb{E}[e^{t(V-\mu)}] \cdot e^{-t\lambda}.$$

Now if we define the (centered) log-MGF of V as

$$\psi(t) := \ln \mathbb{E}[e^{t(V-\mu)}],$$

we get that for any $t \geq 0$,

$$\Pr[V - \mu \geq \lambda] \leq e^{-(t\lambda - \psi(t))}.$$

The best upper bound is obtained when the expression $t\lambda - \psi(t)$ is the largest. The ***Legendre dual*** of the function $\psi(t)$ is defined as

Exercise: if $\psi_1(t) \geq \psi_2(t)$ for all $t \geq 0$, then $\psi_1^*(\lambda) \leq \psi_2^*(\lambda)$ for all λ.

$$\psi^*(\lambda) := \sup t \geq 0 \; \{t\lambda - \psi(t)\},$$

so we get the concise statement for a generic Chernoff bound:

Bounds for the lower tail follow from the arguments applied to the r.v. $-X$.

$$\Pr[V - \mu \geq \lambda] \leq \exp(-\psi^*(\lambda)). \tag{10.9}$$

This abstraction allows us to just focus on bounds on the dual log-MGF function $\psi^*(\lambda)$, making the arguments cleaner.

10.5.1 *A Couple of Examples*

Let's do an example: suppose $V \sim N(\mu, \sigma^2)$, then

$$\begin{aligned}\mathbb{E}[e^{t(V-\mu)}] &= \frac{1}{\sqrt{2\pi}\sigma} \int_{x \in \mathbb{R}} e^{tx} e^{-\frac{x^2}{2\sigma^2}} dx \\ &= \frac{1}{\sqrt{2\pi}\sigma} e^{t^2\sigma^2/2} \int_{x \in \mathbb{R}} e^{-\frac{(x-t\sigma^2)^2}{2\sigma^2}} dx = e^{t^2\sigma^2/2}. \end{aligned} \tag{10.10}$$

Hence, for $N(\mu, \sigma^2)$ r.v.s, we have

$$\psi(t) = \frac{t^2\sigma^2}{2} \qquad \text{and} \qquad \psi^*(\lambda) = \frac{\lambda^2}{2\sigma^2},$$

the latter by basic calculus. Now the generic Chernoff bound for says that for normal $N(\mu, \sigma^2)$ variables,

$$\Pr[V - \mu \geq \lambda] \leq e^{-\frac{\lambda^2}{2\sigma^2}}. \tag{10.11}$$

How about a Rademacher $\{-1, +1\}$-valued r.v. V? The MGF is

$$\mathbb{E}[e^{t(V-\mu)}] = \frac{e^t + e^{-t}}{2} = \cosh t = 1 + \frac{t^2}{2!} + \frac{t^4}{4!} + \cdots \leq e^{t^2/2},$$

so

$$\psi(t) = \frac{t^2}{2} \qquad \text{and} \qquad \psi^*(\lambda) = \frac{\lambda^2}{2}.$$

Note that

$$\psi_{\text{Rademacher}}(t) \leq \psi_{N(0,1)}(t) \implies \psi^*_{\text{Rademacher}}(\lambda) \geq \psi^*_{N(0,1)}(\lambda).$$

This means the upper tail bound for a single Rademacher is at least as strong as that for the standard normal.

10.5.2 *Defining Subgaussian Random Variables*

Definition 10.4. A random variable V with mean μ is ***subgaussian with parameter*** σ if $\psi(t) \leq \frac{\sigma^2 t^2}{2}$.

By the generic Chernoff bound (10.9), such an r.v. has tails that are smaller than those of a normal r.v. with variance σ^2. The following fact is an analog of Proposition 10.3.

Lemma 10.5. *If $V_1, V_2, \ldots$ are independent and σ_i-subgaussian, and $x_1, x_2, \ldots$ are reals, then $V = \sum_i x_i V_i$ is $\sqrt{\sum_i x_i^2 \sigma_i^2}$-subgaussian.*

Proof.

$$\mathbb{E}[e^{t(V-\mu)}] = \mathbb{E}[e^{t \sum_i x_i (V_i - \mu_i)}] = \prod_i \mathbb{E}[e^{t x_i (V_i - \mu_i)}] = \prod_i e^{t x_i (V_i - \mu_i)}.$$

Now taking logarithms, $\psi_V(t) = \sum_i \psi_{V_i}(t x_i) \leq \sum_i \frac{t^2 x_i^2 \sigma_i^2}{2}$. □

10.6 JL Matrices using Rademachers and Subgaussian-ness

Suppose we choose each $M_{ij} \in_R \{-1, +1\}$ and let $A(x) = \frac{1}{\sqrt{k}} Mx$ again? We want to show that

$$Z := \|A(x)\|^2 = \frac{1}{k} \sum_{i=1}^{k} \left(\sum_{j=1}^{D} M_{ij} \cdot x_j \right)^2. \tag{10.12}$$

has mean $\|x\|^2$, and is concentrated sharply around that value.

10.6.1 The Expectation

To keep subscripts to a minimum, consider the inner sum for index i in (10.12), which looks like

$$Y_i := \Big(\sum_j M_j \cdot x_j \Big). \tag{10.13}$$

where M_js are *pairwise independent*, with *mean zero* and *unit variance*.

$$\begin{aligned} \mathbb{E}\Big[Y_i^2\Big] &= \mathbb{E}\Big[(\sum_j M_j x_j)(\sum_l M_l x_l)\Big] \\ &= \mathbb{E}\Big[\sum_j M_j^2 x_j^2 + \sum_{j \neq l} M_j M_l x_j x_l\Big] \\ &= \sum_j \mathbb{E}\Big[M_j^2\Big] x_j^2 + \sum_{j \neq l} \mathbb{E}\Big[M_j M_l\Big] x_j x_l = \sum_j x_j^2. \end{aligned}$$

Here $\mathbb{E}[M_j^2] = \text{Var}(M_j) + \mathbb{E}[M_j]^2 = 1$, and moreover $\mathbb{E}[M_j M_l] = \mathbb{E}[M_j]\mathbb{E}[M_l] = 0$ by pairwise independence. Plugging this into (10.12),

$$\mathbb{E}[Z] = \frac{1}{k} \sum_{i=1}^{k} \mathbb{E}[Y_i^2] = \frac{1}{k} \sum_{i=1}^{k} \sum_j x_j^2 = \|x\|_2^2. \tag{10.14}$$

So the expectation is what we want!

10.6.2 Concentration

Now to show concentration: the direct proof from §10.4 showed the Y_is were themselves Gaussian with variance $\|x\|^2$. Since the Rademachers are 1-subgaussian, Lemma 10.5 shows that Y_i is subgaussian with parameter $\|x\|^2$. Next, we need to consider Z, which is the average of ***squares of*** k independent Y_is. The following lemma shows that the MGF of squares of *symmetric* σ-subgaussians are bounded above by the corresponding Gaussians with variance σ^2.

An r.v. X is ***symmetric*** if it is distributed the same as $R|X|$, where R is an independent Rademacher.

Lemma 10.6. *If V is symmetric mean-zero σ-subgaussian r.v., and $W \sim N(0, \sigma^2)$, then $\mathbb{E}[e^{tV^2}] \leq \mathbb{E}[e^{tW^2}]$ for $t > 0$.*

Proof. Using the calculation in (10.10) in the "backwards" direction

$$\mathbb{E}_V[e^{tV^2}] = \mathbb{E}_{V,W}[e^{\sqrt{2t}(|V|/\sigma)\,W}].$$

(Note that we've just introduced W into the mix, without any provocation!) Since W is also symmetric, we get $|V|W = V|W|$. Hence, rewriting

$$\mathbb{E}_{V,W}[e^{\sqrt{2t}(|V|/\sigma)\,W}] = \mathbb{E}_W[\mathbb{E}_V[e^{(\sqrt{2t}|W|/\sigma)V}]],$$

we can use the σ-subgaussian behavior of V in the inner expectation to get an upper bound of

$$\mathbb{E}_W[e^{\sigma^2(\sqrt{2t}|W|/2)^2/2}] = E_W[e^{tW^2}]. \qquad \square$$

Excellent. Now the tail bound for sums of squares of symmetric mean-zero σ-subgaussians follows from that of Gaussians. Hence we get the same tail bounds as in §10.4.2, and hence that the Rademacher matrix also has the distributional JL property, while using far fewer random bits!

In general one can use other σ-subgaussian distributions to fill the matrix M—using σ different than 1 may require us to rework the proof from §10.4.2 since the linear terms in (10.6) don't cancel any more, see works by Indyk and Naor or Matousek for details.

Indyk and Naor (2008)
Matoušek (2008)

10.6.3 The Fast JL Transform

A different direction to consider is getting fast algorithms for the JL Lemma: Do we really need to plug in non-zero values into every entry of the matrix A? What if most of A is filled with zeroes? The first problem is that if x is a very sparse vector, then Ax might be zero with high probability? Achlioptas showed that having a random two-thirds of the entries of A being zero still works fine: Nir Ailon and Bernard Chazelle showed that if you first hit x with a suitable matrix P which caused Px to be "well-spread-out" whp, and then $\|APx\| \approx \|x\|$ would still hold for a much sparser A. Moreover, this P requires much less randomness, and furthermore, the computations can be done faster too! There has been much work on fast and sparse versions of JL: see, e.g., this paper from SOSA 2018 by Michael Cohen, T.S. Jayram, and Jelani Nelson. Jelani Nelson also has some notes on the Fast JL Transform.

Ailon and Chazelle

Cohen, Jayram, and Nelson (2018)

10.7 Optional: Compressive Sensing

To rewrite. In an attempt to build a better machine to take MRI scans, we decrease the number of sensors. Then, instead of the signal x we intended to obtain from the machine, we only have a small number of measurements of this signal. Can we hope to recover x from the measurements we made if we make sparsity assumptions on x? We use the term r-sparse signal for a vector with at most r nonzero entries.

Formally, x is a n-dimensional vector, and a measurement of x with respect to a vector a is a real number given by $\langle x, a \rangle$. The question we want to answer is how to reconstruct x with r nonzero entries satisfying $Ax = b$ if we are given $k \times n$ matrix A and n dimensional vector b. This is often written as

$$\min \left\{ \|x\|_0 \mid Ax = b. \right\}$$

Here the ℓ_0 "norm" is the total number of non-zeros in the vector x.

Unfortunately, it turns out that the problem as formulated is NP-hard: but this is only assuming A and b are contrived by an adversary. Our setting is a bit different. x is some r-sparse signal out there that we want to determine. We have a handle over A and can choose it to be any matrix we like, and we are provided with appropriate $b = Ax$, from which we attempt to reconstruct x.

Consider the following similar looking problem called the ***basis pursuit*** (BP) problem:

$$\min \left\{ \|x\|_1 \mid Ax = b. \right\}$$

This problem can be formulated as a linear program as follows, and hence can be efficiently solved. Introduce n new variables $y_1, y_2, \ldots, y_n$ under the constraints

$$\min \left\{ \sum_i y_i \mid Ax = b, -y_i \leq x_i \leq y_i \right\}.$$

Definition 10.7. We call a matrix A as *BP-exact* if for all $b = Ax$ such that x^* is an r-sparse solution, x^* is also the unique solution to basis pursuit.

Call a distribution $\mathcal{D}$ over $k \times n$ matrices a **distributional JL family** if Lemma 10.2 is true when A is drawn from $\mathcal{D}$.

Theorem 10.8 (Donoho, Candes-Tao). *If we pick $A \in \mathbb{R}^{k \times D}$ from a distributional JL family with $k \geq \Omega\left(r \log\left(\frac{D}{r}\right)\right)$, then with high probability A is BP-exact.*

We note that the $r \log \frac{D}{r}$ comes from $\log \binom{D}{r} \approx \log \left(\frac{D}{r}\right)^r = r \log \left(\frac{D}{r}\right)$. The last ingredient that one would use to show Theorem 10.8 is the *Restricted Isometry Property* (RIP) of such a matrix A.

Definition 10.9. A matrix A is (t, ε)-RIP if for all unit vectors x with $\|x\|_0 \leq t$, we have $\|Ax\|_2^2 \in [1 \pm \varepsilon]$.

See Chapter 4 of Ankur Moitra's book for more on compressed sensing, sparse recovery and basis pursuit. 10.8 comes from this paper by Emmanuel Candes and Terry Tao.

10.8 *Some Facts about Balls in High-Dimensional Spaces*

Consider the unit ball $\mathbb{B}_d := \{x \in \mathbb{R}^d \mid \|x\|_2 \leq 1\}$. Here are two facts, whose proofs we sketch. These sketches can be made formal (since the approximations are almost the truth), but perhaps the style of arguments are more illuminating.

Theorem 10.10 (Heavy Shells). *At least $1-\varepsilon$ of the mass of the unit ball in $\mathbb{R}^d$ lies within a $\Theta(\frac{\log 1/\varepsilon}{d})$-width shell next to the surface.*

Proof. (Sketch) The volume of a radius-r ball in $\mathbb{R}^d$ goes as r^d, so the fraction of the volume *not* in the shell of width w is $(1-w)^d \approx e^{-wd}$, which is ε when $w \approx \frac{\log 1/\varepsilon}{d}$. □

Given any hyperplane $H = \{x \in \mathbb{R}^d \mid a \cdot x = b\}$ where $\|a\| = 1$, the width-w slab around it is $K = \{x \in \mathbb{R}^d \mid b - w \leq a \cdot x \leq b + w\}$.

Theorem 10.11 (Heavy Slabs). *At least $(1-\varepsilon)$ of the mass of the unit ball in $\mathbb{R}^d$ lies within $\Theta(1/\sqrt{d})$ slab around any hyperplane that passes through the origin.*

Proof. (Sketch) By spherical symmetry we can consider the hyperplane $\{x_1 = 0\}$. The volume of the ball within $\{-w \leq x_1 \leq w\}$ is at

$$\int_{y=0}^{w} (\sqrt{1-y^2})^{d-1} dy \approx \int_{y=0}^{w} e^{-y^2 \cdot \frac{d-1}{2}} dy.$$

If we define $\sigma^2 = \frac{4}{d-1}$, this is

$$\int_{y=0}^{w} e^{-\frac{y^2}{2\sigma^2}} dy \approx \Pr[G \leq w],$$

where $G \sim N(0, \sigma^2)$. But we know that $\Pr[G \geq w] \leq e^{-w^2/2\sigma^2}$ by our generic Chernoff bound for Gaussians (10.11). So setting that tail probability to be ε gives

$$w \approx \sqrt{2\sigma^2 \log(1/\varepsilon)} = O\Big(\sqrt{\frac{\log(1/\varepsilon)}{d}}\Big).$$

□

This may seem quite counter-intuitive: that 99% of the volume of the sphere is within $O(1/d)$ of the surface, yet 99% is within $O(1/\sqrt{d})$ of ***any*** central slab! This challenges our notion of the ball "looking like" the smooth circular object, and more like a very spiky sea-urchin. Finally, a last observation:

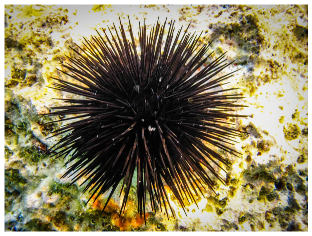


Figure 10.1: Sea Urchin (from uncommoncaribbean.com)

Corollary 10.12. *If we pick two random vectors from the surface of the unit ball in $\mathbb{R}^d$ (i.e., from the sphere), then they are nearly orthogonal with high probability. In particular, their dot-product is smaller than $O(\sqrt{\frac{\log(1/\varepsilon)}{d}})$ with probability $1 - \varepsilon$.*

Proof. Fix u. Then the dot-product $|u \cdot v| \leq w$ if v lies in the slab of width w around the hyperplane $\{x \cdot u = 0\}$. Now using Theorem 10.11 completes the argument. □

This means that if we pick n random vectors in $\mathbb{R}^d$, and set $\varepsilon = 1/n^2$, a union bound gives that all have dot-product $O(\sqrt{\frac{\log n}{d}})$. Setting this dot-product to ε gives us $n = \exp(\varepsilon^2 d)$ unit vectors with mutual dot-products at most ε, exactly as in the calculation at the beginning of the chapter.

11
Streaming Algorithms

We now consider a slightly different computational model called the ***data streaming*** model. In this model we see elements going past in a "stream", and we have very little space to store things. For example, we might be running a program on an Internet router with limited space, and the elements might be IP Addresses. We certainly don't have space to store all the elements in the stream. The question is: which functions of the input stream can we compute with what amount of time and space? While we focus on space, similar questions can be asked for update times.

We denote the stream elements by

$$a_1, a_2, a_3, \ldots, a_t, \ldots$$

We assume each stream element is from alphabet U, and takes $b = |\log_2 U|$ bits to represent. For example, the elements might be 32-bit integers IP addresses. We imagine we are given some function, and we want to compute it continually, on every prefix of the stream. Let us denote $a_{[1:t]} = \langle a_1, a_2, \ldots, a_t \rangle$. For example, if we have seen the integers:

$$3, 1, 17, 4, -9, 32, 101, 3, -722, 3, 900, 4, 32, \ldots \tag{11.1}$$

1. Can we compute the sum of all the integers seen so far? I.e., $F(a_{[1:t]}) = \sum_{i=1}^{t} a_i$. We want the outputs to be

   $$3, 4, 21, 25, 16, 48, 149, 152, -570, -567, 333, 337, 369, \ldots$$

   If we have seen T numbers so far, the sum is at most $T2^b$ and hence needs at most $O(b + \log T)$ space. So we can just keep a counter, and when a new element comes in, we add it to the counter.

2. How about the maximum of the elements so far? $F(a_{[1:t]}) = \max_{i=1}^{t} a_i$. Even easier. The outputs are:

   $$3, 1, 17, 17, 17, 32, 101, 101, 101, 101, 900, 900, 900$$

We just need to store b bits.

3. The median? The outputs on the various prefixes of (11.1) now are

$$3, 1, 3, 3, 3, 3, 4, 3, \ldots$$

And doing this will small space is a lot more tricky.

4. ("distinct elements") Or the number of distinct numbers seen so far? We'd want to output:

$$1, 2, 3, 4, 5, 6, 7, 7, 8, 8, 9, 9, 9 \ldots$$

5. ("heavy hitters") Or the elements that have appeared most often so far? Hmm...

We can imagine the applications of the data-stream model. An Internet router might see a lot of packets whiz by, and may want to figure out which data connections are using the most space? Or how many different connections have been initiated since midnight? Or the median (or the 90^{th} percentile) of the file sizes that have been transferred. Which IP connections are "elephants" (say the ones that have used more than 0.01% of our bandwidth)? Even if we are not working at "line speed", but just looking over the server logs, we may not want to spend too much time to find out the answers, we may just want to read over the file in one quick pass and come up with an answer. Such an algorithm might also be cache-friendly. But how to do this?

Such a router might see tens of millions of packets per second.

Two of the recurring themes will be:

1. Approximate solutions: in several cases, it will be impossible to compute the function exactly using small space. Hence we'll explore the trade-offs between approximation and space.

2. Hashing: this will be a very powerful technique.

11.1 Streams as Vectors, and Additions/Deletions

An important abstraction will be to view the stream as a vector (in high dimensional space). Since each element in the stream is an element of the universe U, we can imagine the stream at time t as a vector $\mathbf{x}^t \in \mathbb{Z}^{|U|}$. Here

$$\mathbf{x}^t = (x_1^t, x_2^t, \ldots, x_{|U|}^t)$$

and x_i^t is the number of times the i^{th} element in U has been seen until time t. (Hence, $x_i^0 = 0$ for all $i \in U$.) When the next element comes in and it is element j, we increment x_j by 1.

This brings us a extension of the model: we could have another model where each element of the stream is either a new element, or an old element departing. Formally, each time we get an *update* a_t, it looks like $(\texttt{add}, e)$ or $(\texttt{del}, e)$. We usually assume that for each element, the number of deletes we see for it is at most the number of adds we see — the running counts of each element is non-negative. As an example, suppose the stream looks like:

In data stream jargon, the addition-only model is called the ***cash-register*** model, whereas the model with both additions and deletions is called the ***turnstile model***. I will not use this jargon.

$$(\texttt{add}, A), (\texttt{add}, B), (\texttt{add}, A), (\texttt{del}, B), (\texttt{del}, A), (\texttt{add}, C), \ldots$$

and if A is the first element of U, then the first coordinate x_1 of the vector s would be $1, 1, 2, 2, 1, 1, \ldots$. This vector notation allows us to formulate some of the problems more easily:

1. The total number of elements currently in the system is just $\|\mathbf{x}\|_1 := \sum_{i=1}^{|U|} x_i$. (This is easy.)

2. We might want to estimate the norms $\|\mathbf{x}\|_2, \|\mathbf{x}\|_p$ of the vector $\mathbf{x}$.

3. The number of distinct elements is the number of non-zero entries in $\mathbf{x}$ is denoted by $\|\mathbf{x}\|_0$.

Let's consider the (non-trivial) problems one by one.

11.2 *Computing Moments*

Recall that $\mathbf{x}^t$ was the vector of frequencies of elements seen so far. Several interesting problems can be posed as computing various norms of $\mathbf{x}^t$: in particular the Euclidean or 2-norm

$$\|\mathbf{x}^t\|_2 = \sqrt{\sum_{i=1}^{|U|} (x_i^t)^2},$$

and the 0-norm (which is not really a norm)

$$\|\mathbf{x}^t\|_0 := \text{ number of non-zeroes in } \mathbf{x}^t.$$

Henceforth, we use the notation that $F_0 := \|\mathbf{x}^t\|_0$ is the number of non-zero entries in $\mathbf{x}$. For $p \geq 1$, we consider the *p-moment*, that is, the p^{th}-power of the p-norm:

$$F_p := \sum_{i=1}^{|U|} (x_i^t)^p. \tag{11.2}$$

We'll develop an algorithm to compute F_2, and to compute F_0; we may see extensions from F_2 to F_p in the homeworks.

11.2.1 *Computing the Second Moment F_2*

The "second moment" F_2 of the stream is often called the "surprise number" (since it captures how uneven the data is). This is also the *size of the self-join*. Clearly we can store the entire vector $\mathbf{x}$ and compute F_2, but that requires storing $|U|$ counts. Here's an algorithm that uses much less space:

This estimator is often called the "tug-of-war" estimator: the hash function randomly partitions the elements into two parties (those mapping to 1, and those to -1), and the counter keeps the difference between the sizes of the two parties.

```
Pick a random hash function $h : U \to \{-1, +1\}$ from family $H$.
Maintain counter $C$, which starts off at zero.
    On update $(add, i) \in U$, increment the counter $C \to C + h(i)$.
    On update $(delete, i) \in U$, decrement the counter $C \to C - h(i)$.
    On query about the value of $F_2$, reply with $C^2$.
```

This estimator was given by Noga Alon, Yossi Matias, and Mario Szegedy, in their Godel-award winning paper on streaming computation.

Alon, Matias, Szegedy (2000)

11.2.2 *Properties of the Hash Family*

The choice of the hash family will be crucial: we want a small family so that we require only a small amount of space to store the hash function, but we want it to be rich enough for the subsequent analysis to go through.

Definition 11.1 (k-universal hash family)**.** H is ***k-universal*** (also called ***uniform*** and ***k-wise independent***) mapping universe U to some range R if all *distinct* elements $i_1, \dots, i_k \in U$ and for values $\alpha_1, \dots, \alpha_k \in R$,

$$\Pr_{h \leftarrow H}\left[\bigwedge_{j=1..k} (h(i_j) = \alpha_j)\right] = \frac{1}{|R|^k}. \tag{11.3}$$

In our application, we want the hash family to be 4-universal from U to the two-element range $R = \{-1, 1\}$. This means that for any element i,

$$\Pr_{h \leftarrow H}[h(i) = 1] = \Pr_{h \leftarrow H}[h(i) = -1] = \frac{1}{2}.$$

Moreover, for four distinct elements i, j, k, l, their maps behave independently of each other, and hence

$$\mathbb{E}[h(i) \cdot h(j) \cdot h(k) \cdot h(l)] = \mathbb{E}[h(i)] \cdot \mathbb{E}[h(j)] \cdot \mathbb{E}[h(k)] \cdot \mathbb{E}[h(l)].$$
$$\mathbb{E}[h(i) \cdot h(j)] = \mathbb{E}[h(i)] \cdot \mathbb{E}[h(j)].$$

We will discuss constructions of such hash families soon, but let us use them to analyze the tug-of-war estimator.

11.2.3 *A Direct Analysis*

Hence, having seen the stream that results in the frequency vector $\mathbf{x} \in \mathbb{Z}_{\geq 0}^{|U|}$, the counter will have the value

$$C := \sum_{i \in U} x_i \, h(i).$$

Remember, the resulting estimate is C^2: so we need to show that $\mathbb{E}[C^2] = F_2$, and variance that is small enough that Chebyshev's inequality ensures we are correct with reasonable probability.

$$\begin{aligned}\mathbb{E}[C^2] &= \mathbb{E}[\sum_{i,j} \big(h(i)x_i \cdot h(j)x_j\big)] = \sum_{i,j} x_i x_j \mathbb{E}[(h(i) \cdot h(j))] \\ &= \sum_i x_i^2 \mathbb{E}[h(i) \cdot h(i)] + \sum_{i \neq j} \sum_{i,j} x_i x_j \mathbb{E}[h(i)] \cdot \mathbb{E}[h(j)] \\ &= \sum_i x_i^2 = F_2.\end{aligned}$$

So in expectation we are correct! Next, recall that the variance is defined as $\mathrm{Var}(C^2) = \mathbb{E}[(C^2)^2] - \mathbb{E}[C^2]^2$:

$$\begin{aligned}\mathbb{E}[(C^2)^2] &= \mathbb{E}[\sum_{p,q,r,s} h(p)h(q)h(r)h(s)x_p x_q x_r x_s] = \\ &= \sum_p x_p^4 \mathbb{E}[h(p)^4] + 6 \sum_{p<q} x_p^2 x_q^2 \mathbb{E}[h(p)^2 h(q)^2] + \text{ other terms} \\ &= \sum_p x_p^4 + 6 \sum_{p<q} x_p^2 x_q^2.\end{aligned}$$

This is because all the other terms have expectation zero. Why? The terms like $\mathbb{E}[h(p)h(q)h(r)h(s)]$ where p, q, r, s are all distinct, all become zero because of 4-universality. Terms like $\mathbb{E}[h(p)^2 h(r)h(s)]$ become zero for the same reason. It is only terms like $\mathbb{E}[h(p)^2 h(q)^2]$ and $\mathbb{E}[h(p)^4]$ that survive, and since $h(p) \in \{-1, 1\}$, they have expectation 1. So

$$\mathrm{Var}(C^2) = \sum_p x_p^4 + 6 \sum_{p<q} x_p^2 x_q^2 - (\sum_p x_p^2)^2 = 4 \sum_{p<q} x_p^2 x_q^2 \leq 2\mathbb{E}[C^2]^2.$$

What does Chebyshev say then?

$$\Pr[|C^2 - \mathbb{E}[C^2]| > \varepsilon \mathbb{E}[C^2]] \leq \frac{\mathrm{Var}(C^2)}{(\varepsilon \mathbb{E}[C^2])^2} \leq \frac{2}{\varepsilon^2}.$$

This is pretty pathetic: since ε is usually less than 1, the RHS usually more than 1.

11.2.4 *Reduce the Variance by Repetition*

The idea is the simplest one: if we have an estimator with mean μ and variance σ^2, then taking the average of k independent copies of this estimator has mean μ and variance σ^2/k. (Why? Summing the independent copies sums the variances and so increases it by k, but dividing by k reduces it by k^2.)

So if we k such independent counters $C_1, C_2, \ldots, C_k$, and return their average $\overline{C} = \frac{1}{k}\sum_i C_i$, we get

$$\Pr[|\overline{C}^2 - \mathbb{E}[\overline{C}^2]| > \varepsilon\mathbb{E}[\overline{C}^2]] \leq \frac{\mathrm{Var}(\overline{C}^2)}{(\varepsilon\mathbb{E}[\overline{C}^2])^2} \leq \frac{2}{k\varepsilon^2}.$$

Taking $k = \frac{2}{\varepsilon^2\delta}$ independent counters gives a probability δ of error on any query. Each counter uses a 4-universal hash function, which requires $O(\log U)$ random bits to store.

11.2.5 *Estimating the p-Moments*

To fix, please skip. A bunch of students (Jason, Anshu, Aram) proposed that for the p^{th}-moment calculation we should use $2p$-wise independent hash functions from U to R, where $R = \{1, \omega, \omega^2, \ldots, \omega^{p-1}\}$, the p primitive roots of unity. Again, we set $C := \sum_{i \in U} x_i h(i)$, and return the real part of C^p as our estimate. This approach has been explored by Ganguly in this paper. Some calculations (and elbow-grease) show that $\mathbb{E}[C^p] = F_p$, but it seems that naively $\mathrm{Var}(C^p)$ tends to grow like F_2^p instead of F_k^p; this leads to pretty bad bounds. Ganguly's paper gives some ways of controlling the variance.

BTW, there is a lower bound saying that any algorithm that outputs a 2-approximation for F_k requires at least $|U|^{1-2/k}$ bits of storage. Hence, while we just saw that for $k = 2$, we can get away with just $O(\log |U|)$ bits to get a $O(1)$-estimate, for $k > 2$ things are much worse.

11.3 *A Matrix View of our Estimator*

Here's a equivalent way of looking at this estimator, that also relates it to the previous chapter and the JL Theorem. Recall that the stream can be viewed as representing a vector $\mathbf{x}$ of size $|U|$, and $F_2 = \|\mathbf{x}\|^2$.

Take a matrix M of dimensions $k \times D$, where $D = |U|$: again, M is a "fat and short" matrix, since $k = O(\varepsilon^{-2}\delta^{-1})$ is small and $D = |U|$ is huge. Pick k independent hash functions $h_1, h_2, \ldots, h_k$ from the 4-universal hash family, and use each one to fill a row of M:

$$M_{ij} := h_i(j).$$

The k counters $C_1, C_2, \ldots, C_k$ are now nothing other than the entries of the matrix-vector product

$$M\,\mathbf{x}.$$

The estimate $\overline{C}^2 = \left(\frac{1}{k}\sum_{i=1}^k C_i\right)^2$ is nothing but

$$\frac{1}{k}\|M\mathbf{x}\|_2^2.$$

This is completely analogous to the construction for JL: we've got a slightly taller matrix with $k = O(\varepsilon^{-2}\delta^{-1})$ rows instead of $k = O(\varepsilon^{-2}\log\delta^{-1})$ rows. However, the matrix entries are not fully independent (as in JL), just 4-wise independent. I.e., we need to store only $O(k \log D)$ bits and can generate any entry of M quickly, whereas the construction for JL stored all kD bits.

Henceforth, we use $S = \frac{1}{\sqrt{k}}M$ to denote the "sketch" matrix.

Let us record two properties of this construction:

Theorem 11.2 (Tug-of-War Sketch)**.** *Take a $k \times D$ matrix S whose columns are 4-wise independent $\{\frac{1}{\sqrt{k}}, \frac{-1}{\sqrt{k}}\}^k$-valued r.v.s. Then for $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$,*

1. $\mathbb{E}[\,\langle S\mathbf{x}, S\mathbf{y}\rangle\,] = \langle \mathbf{x}, \mathbf{y}\rangle$.
2. $\mathrm{Var}(\,\langle S\mathbf{x}, S\mathbf{y}\rangle\,) = \frac{2}{k}\cdot\|\mathbf{x}\|_2^2\|\mathbf{y}\|_2^2$.

The proofs is similar to that in §11.2.3; using $\mathbf{y} = \mathbf{x}$ gives us exactly the results from that section. Moreover, an analogous theorem can also be given in the JL construction, with fewer rows but with completely independent entries.

11.4 *Application: Approximate Matrix Multiplication*

Suppose we want to multiply square matrices $A, B \in \mathbb{R}^{n\times n}$, but want to solve the problem faster, at the expense of getting only an approximate solution $C \approx AB$. How should we measure the error? Requiring that the answer be close entry-wise to the actual answer is a hard problem. Let's aim for something weaker: we want the "aggregate error" to be small.

It's as though we think of the matrix as just a vector and look at its Euclidean length.

Formally, the ***Frobenius norm*** of matrix M is

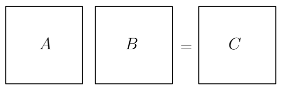
$$\|M\|_F := \sqrt{\sum_{i,j} M_{ij}^2}.$$

Our guarantee for approximate matrix multiplication will be

$$\|C - AB\|_F^2 \leq \text{ small.}$$

Here's the idea: we want to do the matrix multiplication:

$$C = AB$$



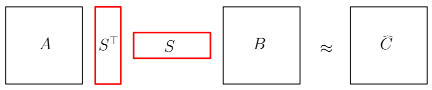
This usually takes $O(n^3)$ time. Indeed, the ij^{th} entry of the product C is the dot-product of the i^{th} row $A_{i\star}$ of A with the j^{th} column $B_{\star j}$ of B, and the dot-product takes $O(n)$ time.

Suppose instead we use a "fat and short" $k \times n$ matrix S (for $k \ll n$), and calculate

$$\widetilde{C} = AS^\intercal SB.$$

By associativity of matrix multiplication, we could first compute $(AS^\intercal)$ and (SB) in times $O(n^2k)$, and then multiply the results in time $O(nk^2)$. Moreover, the matrix S from the previous section works pretty well, where we set $D = n$.

The intuition is that $S^\intercal S$ is an almost-identity matrix, it has 1 on the diagonals and at most ε everywhere else. And hence it gives only a small error. Of course, we don't multiply out $S^\intercal S$, but instead compute $AS^\intercal$ and SB, and then multiply the smaller matrices.



Indeed, entries of the error matrix $Y = C - \widetilde{C}$ satisfy

$$\mathbb{E}[Y_{ij}] = 0$$

and

$$\mathbb{E}[Y_{ij}^2] = \mathrm{Var}(Y_{ij}) + \mathbb{E}[Y_{ij}]^2 = \mathrm{Var}(Y_{ij}) \leq \tfrac{2}{k}\|A_{i\star}\|_2^2\|B_{\star j}\|_2^2.$$

So

$$\begin{aligned}\mathbb{E}[\|AB - AS^\intercal SB\|_F^2] = \mathbb{E}[\sum_{ij} Y_{ij}^2] = \sum_{ij}\mathbb{E}[Y_{ij}^2] &= \tfrac{2}{k}\textstyle\sum_{ij}\|A_{i\star}\|_2^2\|B_{\star j}\|_2^2 \\ &= \tfrac{2}{k}\|A\|_F^2\|B\|_F^2.\end{aligned}$$

The squared Frobenius norm of a matrix is the sum of squared Euclidean lengths of the columns, or of the rows.

Finally, setting $k = \frac{2}{\varepsilon^2\delta}$ and using Markov's inequality, we can say that for any fixed $\varepsilon > 0$, we can compute an approximate matrix product $C := AS^\intercal SB$ such that

$$\Pr\left[\|AB - C\|_F \leq \varepsilon \cdot \|A\|_F\|B\|_F\right] \geq 1 - \delta,$$

in time $O(\frac{n^2}{\varepsilon^2\delta})$. (If we want to make δ very small, at the expense of picking more independent random bits in the sketching matrix S, we can use the JL matrices instead. Details will appear in a homework.) Finally, if the matrices A, B are sparse and contains only $\ll n^2$ entries, the time can be made to depend on $nnz(A, B)$.

The approximate matrix product question has been considered often, e.g., by Edith Cohen and David Lewis using a random-walks approach. The algorithm we present is due to Tamás Sarlós; his paper gives better results, as well as extensions to computing SVDs faster. Better bounds have subsequently been given by Clarkson and Woodruff. More recent refs too.

Cohen and Lewis (1999)

11.5 *Optional: Computing the Number of Distinct Elements*

Our last example today will be to compute F_0, the number of distinct elements seen in the data stream, but in the addition-only model, with no deletions. (We'll see another approach in a HW.)

11.5.1 *A Simple Lower Bound*

Of course, if we store $\mathbf{x}$ explicitly (using $|U|$ space), we can trivially solve this problem exactly. Or we could store the (at most) t elements seen so far, again we could give an exact answer. And indeed, we cannot do much better if we want no errors. Here's a proof sketch for deterministic algorithms (one can extend this to randomized algorithms with some more work).

Lemma 11.3 (A Lower Bound)**.** *Suppose a deterministic algorithm correctly reports the number of distinct elements for each sequence of length at most N. Suppose $N \leq 2|U|$. Then it must use at least $\Omega(N)$ bits of space.*

Proof. Consider the situation where first we send in some subset S of $N-1$ elements distinct elements of U. Look at the information stored by the algorithm. We claim that we should be able to use this information to identify exactly which of the $\binom{|U|}{N-1}$ subsets of U we have seen so far. This would require

$$\log_2 \binom{|U|}{N-1} \geq (N-1)\big(\log_2 |U| - \log_2(N-1)\big) = \Omega(N)$$

bits of memory.[1]

[1] We used the approximation that $\binom{m}{k} \geq \left(\frac{m}{k}\right)^k$, and hence $\log_2 \binom{m}{k} \geq k(\log_2 m - \log_2 k)$.

OK, so why should we be able to uniquely identify the set of elements until time $N-1$? For a contradiction, suppose we could not tell whether we'd seen S_1 or S_2 after $N-1$ elements had come in. Pick any element $e \in S_1 \setminus S_2$. Now if we gave the algorithm e as the N^{th} element, the number of distinct elements seen would be N if we'd already seen S_2, and $N-1$ if we'd seen S_1. But the algorithm could not distinguish between the two cases, and would return the same answer. It would be incorrect in one of the two cases. This contradicts the claim that the algorithm always correctly reports the number of distinct elements on streams of length N. □

OK, so we need an approximation if we want to use little space. Let's use some hashing magic.

11.5.2 *The Intuition*

Suppose there are $d = \|\mathbf{x}\|_0$ distinct elements. If we randomly map d distinct elements onto the line $[0,1]$, we expect to see the smallest mapped value at location $\approx \frac{1}{d}$. (I am assuming that we map these elements ***consistently***, so that multiple copies of an element go to the same place.) So if the smallest value is δ, one estimator for the number of elements is $1/\delta$.

This is the essential idea. To make this work (and analyze it), we change it slightly: The variance of the above estimator is large. By the same argument, for any integer s we expect the s^{th} smallest mapped value at $\frac{s}{d}$. We use a larger value of s to reduce the variance.

11.5.3 The Algorithm

Assume we have a hash family H with hash functions $h : U \to [M]$. (We'll soon figure out the precise properties we'll want from this hash family.) We will later fix the value of the parameter s to be some large constant. Here's the algorithm:

```
Pick a hash function $h$ randomly from $H$.
If query comes in at time $t$
   Consider the hash values $h(a_1), h(a_2), \ldots, h(a_t)$ seen so far.
     Let $L_t$ be the $s^{th}$ smallest distinct hash value $h(a_i)$ in this
set.
  Output the estimate $D_t = \frac{M \cdot s}{L_t}$.
```

The crucial observation is: it does not matter if we see an element e once or multiple times — the algorithm will behave the same, since the output depends on what ***distinct*** elements we've seen so far. Also, maintaining the s^{th} smallest element can be done by remembering at most s elements. (So we want to make s small.)

How does this help? As a thought experiment, if we had d distinct darts and threw them in the continuous interval $[0, M]$, we would expect the location of the s^{th} smallest dart to be about $\frac{s \cdot M}{d}$. So if the s^{th} smallest dart was at location ℓ in the interval $[0, M]$, we would be tempted to equate $\ell = \frac{s \cdot M}{d}$ and hence guessing $d = \frac{s \cdot M}{\ell}$ would be a good move. Which is precisely why we used the estimate

$$D_t = \frac{M \cdot s}{L_t}.$$

Of course, all this is in expectation—the following theorem argues that this estimate is good with reasonable probability.

Theorem 11.4. *Consider some time t. If H is a uniform 2-universal hash family mapping $U \to [M]$, and M is large enough, then both the following guarantees hold:*

$$\Pr[D_t > 2\,\|\mathbf{x}^t\|_0] \le \frac{3}{s}, \text{ and} \tag{11.4}$$

$$\Pr[D_t < \frac{\|\mathbf{x}^t\|_0}{2}] \le \frac{3}{s}. \tag{11.5}$$

We will prove this in the next section. First, some observations. Firstly, we now use the stronger assumption that that the hash family ***2-universal***; recall the definition from Section 11.2.2. Next, setting $s = 8$ means that the estimate D_t lies within $[\frac{\|\mathbf{x}^t\|_0}{2}, 2\|\mathbf{x}^t\|_0]$ with probability at least $1 - (1/4 + 1/4) = 1/2$. (And we can boost the success probability by repetitions.) Secondly, we will see that the estimation error of a factor of 2 can be made $(1+\varepsilon)$ by changing the parameters s and k.

11.5.4 *Proof of Theorem 11.4*

Now for the proof of the theorem. We'll prove bound (11.5), the other bound (11.4) is proved identically. Some shorter notation may help. Let $d := \|\mathbf{x}^t\|_0$. Let these d distinct elements be $T = \{e_1, e_2, \dots, e_d\} \subseteq U$.

The random variable L_t is the s^{th} smallest distinct hash value seen until time t. Our estimate is $\frac{sM}{L_t}$, and we want this to be at least $d/2$. So we want L_t to be ***at most*** $\frac{2sM}{d}$. In other words,

$$\Pr[\text{ estimate too low }] = \Pr[D_t < d/2] = \Pr[L_t > \frac{2sM}{d}].$$

Recall T is the set of all d $(= \|\mathbf{x}^t\|_0)$ distinct elements in U that have appeared so far. How many of these elements in T hashed to values greater than $2sM/d$? The event that $L_t > 2sM/d$ (which is what we want to bound the probability of) is the same as saying that fewer than s of the elements in T hashed to values smaller than $2sM/d$. For each $i = 1, 2, \dots, d$, define the indicator

$$X_i = \begin{cases} 1 & \text{if } h(e_i) \le 2sM/d \\ 0 & \text{otherwise} \end{cases} \tag{11.6}$$

Then $X = \sum_{i=1}^{d} X_i$ is the number of elements seen that hash to values below $2sM/d$. By the discussion above, we get that

$$\Pr\left[L_t < \frac{2sM}{d}\right] \le \Pr[X < s].$$

We will now estimate the RHS.

Next, what is the chance that $X_i = 1$? The hash $h(e_i)$ takes on each of the M integer values with equal probability, so

$$\Pr[X_i = 1] = \frac{\lfloor sM/2d \rfloor}{M} \ge \frac{s}{2d} - \frac{1}{M}. \tag{11.7}$$

By linearity of expectations,

$$\mathbb{E}[X] = E\left[\sum_{i=1}^{d} X_i\right] = \sum_{i=1}^{d} E\left[X_i\right] = \sum_{i=1}^{d} \Pr\left[X_i = 1\right] \ge d \cdot \left(\frac{s}{2d} - \frac{1}{M}\right) = \left(\frac{s}{2} - \frac{d}{M}\right).$$

Let's imagine we set M large enough so that d/M is, say, at most $\frac{s}{100}$. Which means

$$\mathbb{E}[X] \ge \left(\frac{s}{2} - \frac{s}{100}\right) = \frac{49\,s}{100}.$$

So by Markov's inequality,

$$\Pr\left[X > s\right] = \Pr\left[X > \frac{100}{49}\mathbb{E}[X]\right] \leq \frac{49}{100}.$$

Good? Well, not so good. We wanted a probability of failure to be smaller than $2/s$, we got it to be slightly less than $1/2$. Good try, but no cigar.

11.5.5 *Enter Chebyshev*

Recall that $\mathrm{Var}(\sum_i Z_i) = \sum_i \mathrm{Var}(Z_i)$ for pairwise-independent random variables Z_i. (Why?) Also, if Z_i is a $\{0,1\}$ random variable, $\mathrm{Var}(Z_i) \leq \mathbb{E}[Z_i]$. (Why?) Applying these to our random variables $X = \sum_i X_i$, we get

$$\mathrm{Var}(X) = \sum_i \mathrm{Var}(X_i) \leq \sum_i \mathbb{E}[X_i] = E(X).$$

(The first inequality used that the X_i were pairwise independent, since the hash function was 2-universal.) Is this variance "low" enough? Plugging into Chebyshev's inequality, we get:

$$\Pr[X > s] = \Pr[X > \frac{100}{49}\mu_X] \leq \Pr[|X - \mu_X| > \frac{50}{49}\mu_X] \leq \frac{\sigma_X^2}{(50/49)^2\mu_X^2} \leq \frac{1}{(50/49)^2\mu_X} \leq \frac{3}{s}.$$

Which is precisely what we want for the bound (11.4). The proof for the bound (11.5) is similar and left as an exercise.

If we want the estimate to be at most $\frac{\|\mathbf{x}^t\|_0}{(1+\varepsilon)}$, then we would want to bound $\Pr[X < \frac{\mathbb{E}[X]}{(1+\varepsilon)}]$. Similar calculations should give this to be at most $\frac{3}{\varepsilon^2 s}$, as long as M was large enough. In that case we would set $s = O(1/\varepsilon^2)$ to get some non-trivial guarantees.

11.5.6 *Final Bookkeeping*

Excellent. We have a hashing-based data structure that answers "number of distinct elements seen so far" queries, such that each answer is within a multiplicative factor of 2 of the actual value $\|\mathbf{x}^t\|_0$, with small error probability.

Let's see how much space we actually used. Recall that for failure probability $1/2$, we could set $s = 12$, say. And the space to store the s smallest hash values seen so far is $O(s \lg M)$ bits. For the hash functions themselves, the standard constructions use $O((\lg M) + (\lg U))$ bits per hash function. So the total space used for the entire data structure is

$$O(\log M) + (\lg U) \text{ bits.}$$

What is M? Recall we needed to M large enough so that $d/M \leq s/100$. Since $d \leq |U|$, the total number of elements in the universe, set $M = \Theta(U)$. Now the total number of bits stored is

$$O(\log U).$$

And the probability of our estimate D_t being within a factor of 2 of the correct answer $\|\mathbf{x}^t\|_0$ is at least $1/2$.

12
Dimension Reduction: Singular Value Decompositions

12.1 Introduction

In this lecture, we see a very popular und useful dimension reduction technique that is based on the ***singular value decomposition*** (SVD) of a matrix. In contrast to the dimension reduction obtained by the Johnson-Lindenstrauss Lemma, SVD based dimension reductions are not distance preserving. That means that we allow that the distances between pairs of points in our input change. Instead, we want to keep the shape of the point set by fitting it to a subspace according to a least squares error. This preserves most of the 'energy' of the points.

More precisely, the problem that we want to solve is the following. We are given a matrix $A \in \mathbb{R}^{n\times d}$. The points are the rows of A, which we also name $a_1, \ldots, a_n \in \mathbb{R}^d$. Let the rank of A be r, so $r \leq \min\{n, d\}$. Given an integer k, we want to find a subspace V of dimension k that minimizes the sum of the squared distances of all points in A to V. Thus, for each point in A, we square the distance between the point and its projection to V and add these squared errors, and this term should be minimized by our choice of V.

This task can be solved by computing the SVD of A, a decomposition of A into matrices with nice properties. We will see that we can write A as

$$A = \begin{pmatrix} \text{——} & a_1 & \text{——} \\ & \vdots & \\ \text{——} & a_n & \text{——} \end{pmatrix} = \begin{pmatrix} | & & | \\ u_1 & \cdots & u_n \\ | & & | \end{pmatrix} \begin{pmatrix} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_r \end{pmatrix} \begin{pmatrix} \text{——} & v_1 & \text{——} \\ & \vdots & \\ \text{——} & v_d & \text{——} \end{pmatrix} = UDV^\intercal$$

where $U \in \mathbb{R}^{n\times r}$ and $V \in \mathbb{R}^{r\times d}$ are matrices with orthonormal columns and $D \in \mathbb{R}^{r\times r}$ is a diagonal matrix. Notice that the columns of V are the d-dimensional points $v_1, \ldots, v_d$ which appear in the rows of the above matrix since it is $V^\intercal$.

Notice that the SVD can give us an intuition of how A acts as a mapping. We have that

$$AV = UDV^{\mathsf{T}}V = UD$$

because V consists of orthonormal columns. Imagine the r-dimensional sphere that is spanned by $v_1, \ldots, v_r$. The linear mapping defined by A maps this sphere to an ellipsoid with $\sigma_1 u_1, \ldots, \sigma_r u_r$ as the axes, like shown in Figure 12.1.

The singular value decomposition was developed by different mathematicians around the beginning of the 19th century. The survey by Stewart [1] gives an historical overview on its origins. In the following, we see how to obtain the SVD and why it solves our best fit problem. The lecture is partly based on [2].

[1]

[2]

12.2 Best fit subspaces of dimension k and the SVD

We start with the case that $k = 1$. Thus, we look for the line through the origin that minimizes the sum of the squared errors. See Figure 12.2. It depicts a one-dimensional subspace V in blue. We look at a point a_i, its distance β_i to V, and the length of its projection to V which is named α_i in the picture. Notice that the length of a_i is $\alpha_i^2 + \beta_i^2$. Thus, for our fixed a_i, minimizing β_i is equivalent to maximizing α_i. If we represent V by a unit vector v that spans V (depicted in orange in the picture), then we can compute the projection of a_i to V by the dot product $\langle a_i, v\rangle$. We have just argued that we can find the best fit subspace of dimension one by solving

$$\max_{v\in\mathbb{R}^d, \|v\|=1} \sum_{i=1}^{n} \langle a_i, v\rangle^2 = \min_{v\in\mathbb{R}^d, \|v\|=1} \sum_{i=1}^{n} \mathrm{dist}(a_i, \mathrm{span}(v))^2$$

where we denote the distance between a point a_i and the line spanned by v by $\mathrm{dist}(a_i, \mathrm{span}(v))^2$. Now because $Av = (\langle a_1, v\rangle, \langle a_2, v\rangle, \ldots, \langle a_n v\rangle)^{\mathsf{T}}$, we can rewrite $\sum_{i=1}^{d} \langle a_i, v\rangle^2$ as $\|Av\|^2$. We define the first right singular vector to be a unit vector that maximizes $\|Av\|$.[3] We thus know that the subspaces spanned by it is the best fit subspace of dimension one.

[3] There may be many vectors that achieve the maximum: indeed, for every v that achieves the maximum, $-v$ also has the same maximum. Let us break ties arbitrarily.

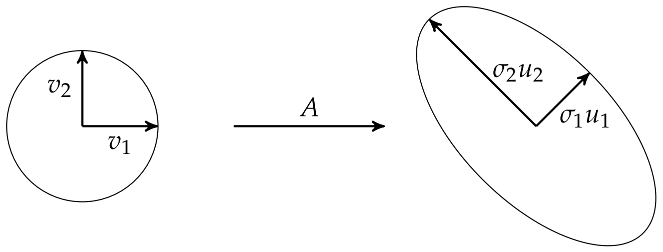


Figure 12.1: A visualization of $AV = UD$ for $r = 2$.

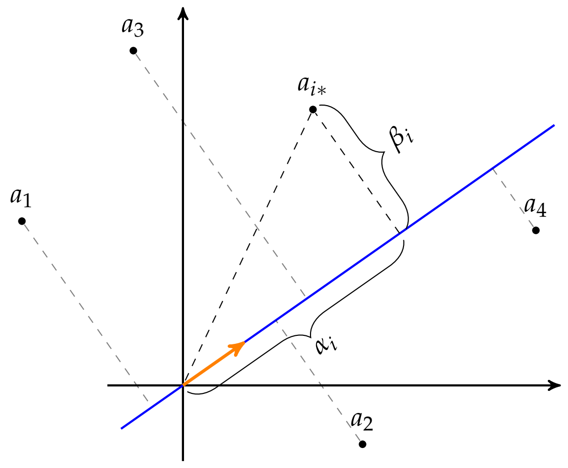


Figure 12.2: Finding the best fit subspace of dimension one.

Now we want to generalize this concept to more than one dimension. It turns out that to do so, we can iteratively pick orthogonal unit vectors that span more and more dimensions. Among all unit vectors that are orthogonal to those chosen so far, we pick a vector that maximizes $\|Av\|$. This is formalized in the following definition.

Definition 12.1. Let $A \in \mathbb{R}^{n\times d}$ be a matrix. We define

$$\begin{aligned}
v_1 &= \arg\max_{\|v\|=1} \|Av\|, & \sigma_1(A) &:= \|Av_1\| \\
v_2 &= \arg\max_{\|v\|=1, \langle v, v_1\rangle = 0} \|Av\|, & \sigma_2(A) &:= \|Av_2\| \\
&\vdots \\
v_r &= \arg\max_{\|v\|=1, \langle v, v_i\rangle = 0\ \forall i=1,\ldots,r-1} \|Av\|, & \sigma_r(A) &:= \|Av_r\|
\end{aligned}$$

and say that $v_1, \ldots, v_r$ are ***right singular vectors*** of A and that $\sigma_1 := \sigma_1(A), \ldots, \sigma_r := \sigma_r(A)$ are the ***singular values*** of A. Then we define the ***left singular vectors*** by setting

$$u_i := \frac{Av_i}{\|Av_i\|} \quad \text{for all } i = 1, \ldots, r.$$

One worry is that this greedy process picked v_2 after fixing v_1, and hence the span of v_1, v_2 may not be the best two-dimensional subspace. The following claim says that Definition 12.1 indeed gives us the the best fit subspaces.

Claim 12.2. For any k, the subspace V_k, which is the span of $v_1, \ldots, v_k$, minimizes the sum of the squared distances of all points among all subspaces of dimension k.

Proof. Let V_2 be the subspace spanned by v_1 and v_2. Let W be any other 2-dimensional subspace and let w_1, w_2 be an orthonormal basis of W. Recall that the squared length of the projection of a point a_i to V decomposes into the squared lengths of the projections to the lines spanned by v_1 and v_2 and the same is true for W, w_1 and w_2.

Since we chose v_1 to maximize $\|Av\|$, we know that $\|Aw_1\| \leq \|Av_1\|$. Similarly, it holds that $\|Aw_2\| \leq \|Av_2\|$, which means that

$$\|Aw_1\|^2 + \|Aw_2\|^2 \leq \|Av_1\|^2 + \|Av_2\|^2.$$

We can extend this argument by induction to show that the space spanned by $v_1, \ldots, v_k$ is the best fit subspace of dimension k. □

We review some properties of the singular values and vectors. Notice that as long as $i < r$, there is always a vector in the row space of A that is linearly independent to $v_1, \ldots, v_i$, which ensures that $\max \|Av\|$ is nonzero. For $i = r$, the vectors $v_1, \ldots, v_r$ span the row space of A. Thus, any vector that is orthogonal to them lies in the kernel of A, meaning that $\arg\max_{\|v\|=1, \langle v, v_i \rangle = 0\ \forall i=1,\ldots,i-1} \|Av\| = 0$, so we end the process at this point. By construction, we know that the singular values are not increasing. We also see that the right singular vectors form a orthonormal basis of the row space of A. This is true for the left singular vectors and the column space as well (homework). The following fact summarizes the important properties.

Fact 12.3. The sets $\{u_1, \ldots, u_r\}$ and $\{v_1, \ldots, v_r\}$ as defined in 12.1 are both orthonormal sets and span the column and row space, respectively. The singular values satisfy $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$.

So far, we defined the v_i purely based on the goal to find the best fit subspace. Now we claim that in doing so, we have actually found the decomposition we wanted, i.e. that

$$UDV^\intercal := \begin{pmatrix} | & & | \\ u_1 & \cdots & u_n \\ | & & | \end{pmatrix} \begin{pmatrix} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_r \end{pmatrix} \begin{pmatrix} \text{---} & v_1 & \text{---} \\ & \vdots & \\ \text{---} & v_d & \text{---} \end{pmatrix} = A. \tag{12.1}$$

Claim 12.4. For any matrix $A \in \mathbb{R}^{n \times d}$ and U, V, D as in (12.1), it holds that

$$A = UDV^\intercal.$$

Proof. We prove the claim by using the fact that two matrices $A, B \in \mathbb{R}^{n \times d}$ are identical iff for all vectors v, the images are equal, i.e. $Av = Bv$. Notice that it is sufficient to check this for a basis, so it is true if the following subclaim holds (which we do not prove):

Subclaim: Two matrices $A, B \in \mathbb{R}^{n \times d}$ are identical iff $Av = Bv$ for all v in a basis of $\mathbb{R}^d$.

We use the subclaim for $B = UDV^\intercal$. Notice that we can extend $v_1, \ldots, v_r$ to a basis of $\mathbb{R}^d$ by adding orthonormal vectors from the kernel of A. These additional vectors are orthogonal to all vectors in the rows of $V^\intercal$, so $V^\intercal v$ is the zero vector for all of them. Since they are in the kernel of A, it holds $\overrightarrow{0} = Av = Bv = UD\overrightarrow{0} = \overrightarrow{0}$ for the additional basis vectors. For $i = 1, \ldots, r$, we notice that

$$(UDV^\intercal)v_i = UDe_i = u_i\sigma_i = \frac{Av_i}{\|Av_i\|} \cdot \|Av_i\| = Av_i$$

which completes the proof. □

12.3 Useful facts, and rank-k-approximation

Singular values are a generalization of the concept of eigenvalues for square matrices. Recall that a square symmetric matrix M can be written as $M = \sum_{i=1}^{r} \lambda_i v_i v_i^\intercal$ where λ_i and v_i are eigenvalues and eigenvectors, respectively. This decomposition can be used to define the singular vectors in a different way. In fact, the right singular vectors of A correspond to the eigenvectors of $A^\intercal A$ (notice that this matrix is square and symmetric), and the left singular vectors correspond to the eigenvectors of $AA^\intercal$.

This fact can also be used to compute the SVD. Computing the SVD or eigenvalues and -vectors in a numerically stable way is the topic of a large research area, and there are different ways to obtain algorithms that converge under the assumption of a finite precision.

Fact 12.5. The SVD can be found (up to arbitrary precision) in time $\mathcal{O}(\min(nd^2, n^2d))$ or even in time $\mathcal{O}(\min(nd^{\omega-1}, dn^{\omega-1}))$ where ω is the matrix multiplication constant. (Here the big-O term hides the dependence on the precision.)

The SVD is unique in the sense that for any $i \in [r]$, the subspace spanned by unit vectors v that maximize $\|Av\|$ is unique. Aside from the different choices of an orthonormal basis of these subspaces, the singular vectors are uniquely defined. For example, if all singular values are distinct, then the subspace of unit vectors that maximize $\|Av\|$ is one-dimensional and the singular vector is unique (up to sign changes, i.e., up to multiplication by -1).

Sometimes, it is helpful to observe that the matrix product $UDV^\intercal$ can also be written as the sum of outer products of the singular vectors. This formulation has the advantage that we can write the projection of A to the best fit subspaces of dimension k as the sum of the first k terms.

Remark 12.6. The SVD can equivalently be written as

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^\intercal$$

where $u_i v_i^\intercal$ is the outer product. For $k \leq r$, the projection of A to V_k is

$$A_k := \sum_{i=1}^{k} \sigma_i u_i v_i^\intercal .$$

Recall that the Frobenius norm of a matrix A is the square root of the sum of its squared entries, i.e. it is defined by $\|A\|_F := \sqrt{\sum_{i,j} a_{ij}^2}$. This means that $\|A - B\|_F^2$ is equal to the sum of the squared distances between each row in A and the corresponding row in B for matrices of equal dimensions. Imagine that B is a rank k matrix. Then its points lie within a k-dimensional subspace, and $\|A - B\|_F^2$ cannot be smaller than the distance between A and this subspace. Since A_k is the projection to the best fit subspace of dimension k, A_k minimizes $\|A - B\|_F$ (notice that A_k has rank at most k). It is therefore also called the best rank k-approximation of A.

Theorem 12.7. *Let $A \in \mathbb{R}^{n \times d}$ be a matrix of rank r and let $k \leq r$ be given. It holds that*

$$\|A - A_k\|_F \leq \|A - B\|_F$$

for any matrix $B \in \mathbb{R}^{n \times d}$ of rank at most k.

The theorem is also true if the Frobenius norm is replaced by the ***spectral norm***.[4] For a matrix A, the spectral norm is equal to the maximum singular value, i.e. $\|A\|_2 := \max_{v \in \mathbb{R}^d, \|v\|=1} \|Av\| = \sigma_1$.

[4] In fact, this theorem holds for any ***unitarily invariant matrix norm***; a matrix norm $\|\cdot\|$ is unitarily invariant if $\|A\| = \|UAV\|$ for any unitary matrices U, V. Other examples of unitarily invariant norms are the Schatten norms, and the Ky Fan norms. J. von Neumann characterized all unitarily invariant matrix norms as those obtained by taking a "symmetric" (vector) norm of the vector of singular values — here symmetric means $\|x\| = \|y\|$ when y is obtained by flipping the signs of some entries of x and then permuting them around. See Theorem 7.4.24 in the text by Horn and Johnson.

12.4 *Applications*

Topic modeling. Replacing A by A_k is a great compression idea. For example, for ***topic modeling***, we imagine A to be a matrix that stores the number of times that any of d words appears in any of n documents. Then we assume that the rank $r$of A corresponds to r ***topics***. Recall that

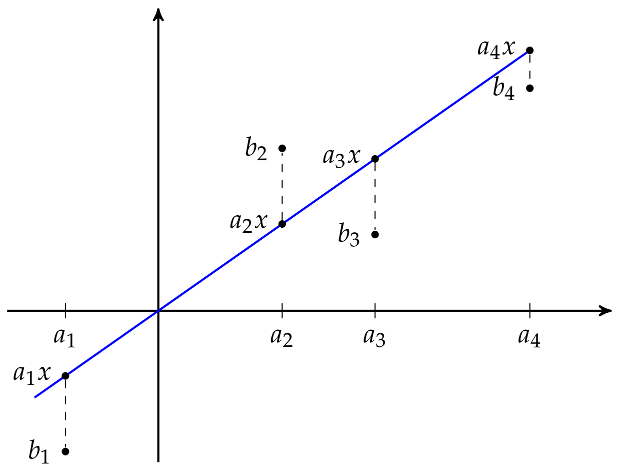
$$A = \begin{pmatrix} \text{---} & a_1 & \text{---} \\ & \vdots & \\ \text{---} & a_n & \text{---} \end{pmatrix} = \begin{pmatrix} | & & | \\ u_1 & \cdots & u_n \\ | & & | \end{pmatrix} \begin{pmatrix} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_r \end{pmatrix} \begin{pmatrix} \text{---} & v_1 & \text{---} \\ & \vdots & \\ \text{---} & v_d & \text{---} \end{pmatrix}.$$

Assume that the entries in U and V are positive. Since the column vectors are unit vectors, they define a convex combination of the r topics. We can thus imagine U to contain information on how much each of the documents consists of each topic. Then, D assigns a weight to each of the topics. Finally, we $V^\intercal$ gives information on how much each topic consists of each of the words. The combination of the three matrices generates the actual documents. By using the

SVD, we can represent a set of documents based on fewer topics, thus obtaining an easier model of how they are generated.

Notice that this interpretation of the SVD needs that the entries are non negative, and that obtaining such a decomposition is an NP-hard problem.

12.4.1 *Pseudoinverse and least squares regression*



For any diagonal matrix $M = \mathrm{diag}(d_1, \ldots, d_\ell)$, define $M^+ := \mathrm{diag}(1/d_1, \ldots, 1/d_\ell)$. We notice that for the matrices from the SVD, it holds that

$$VD^+U^\intercal UDV = \mathrm{diag}(\underbrace{1, \ldots, 1}_{r \text{ times}}, 0, \ldots, 0).$$

If A is an $n \times n$-matrix of rank n, then $r = n$ and the result of this product is I. Thus, $A^+ := VD^+U^\intercal$ is then the inverse of A. In general, A^+ is the (Moore Penrose) ***pseudoinverse*** of A. It satisfies that

$$A(A^+b) = b \quad \forall b \text{ in the image of } A$$

The pseudoinverse helps to find the solution to another popular minimization problem, ***least squares regression***. Given an overconstrained system of equations $Ax = b$, least squares regression asks for a point x that minimizes the squared error $\|Ax - b\|_2^2$. I.e., we want

$$x^* := \arg\min \|Ax - b\|_2^2.$$

Notice that if there is an x' with $Ax' = b$, then it also minimizes $\|Ax' - b\|_2^2$, and if A had full rank this x' would be obtained by computing $A^{-1}b$. If A does not have full rank, an optimal solution is obtained by using the pseudoinverse:

$$x^* = A^+b$$

(This is often used as another definition for the pseudoinverse.)

Here's a proof: for any choice of $x \in \mathbb{R}^d$, Ax is some point in the column span of A. So x^*, the minimizer, must be the projection of b onto colspan(A). One orthonormal basis for colspan(A) is the columns of U. Hence the projection Πb of b onto colspan(A) is given by $UU^\intercal b$. (Why? Extend U to a basis for all of $\mathbb{R}^d$, write b in this basis, and consider what it's projection must be.) Hence we want $Ax^* = UU^\intercal b$. For this, it suffices to set $x^* = VD^+U^\intercal b = A^+b$.

12.5 Symmetric Matrices

For a (square) symmetric matrix A, the (normalized) eigenvectors v_i and the eigenvalues λ_i satisfy the following properties: the v_is form an orthonormal basis, and $A = V\Lambda V^\intercal$, where the columns of V are the v_i vectors, and Λ is a diagonal matrix with λ_is on the diagonal. It is no longer the case that the eigenvalues are all non-negative. (In fact, we can match up the eigenvalues and singular values such that they differ only in sign.)

Given a function $f : \mathbb{R} \to \mathbb{R}$, we can extend this to a function on symmetric matrices as follows:

$$f(A) = V \text{ diag}(f(\lambda_1), \ldots, f(\lambda_n))\, V^\intercal.$$

For instance, you can check that A^k or e^A defined this way indeed correspond to what you think they might mean. (The other way to define e^A would be $\sum_{k\geq 0} \frac{A^k}{k!}$.)

Part III

From Discrete to Continuous Algorithms

13
Online Learning: Experts and Bandits

In this set of chapters, we consider a basic problem in online algorithms and online learning: how to dynamically choose from among a set of "experts" in a way that compares favorably to any fixed expert. Both this abstract problem, and the techniques behind the solution, are important parts of the algorithm designer's toolkit.

13.1 The Mistake-Bound Model

Suppose there are N experts who make predictions about a certain event every day—for example, whether it rains today or not, or whether the stock market goes up or not. Let U be the set of possible choices. The process in the experts setting goes as follows:

The term *expert* just refers to a person who has an opinion, and does not reflect whether they are good or bad at the prediction task at hand.

1. At the beginning of each time step t, each expert makes a prediction. Let $\mathcal{E}^t \in U^N$ be the vector of predictions.

2. The algorithm makes a prediction a^t, and simultaneously, the actual outcome o^t is revealed.

Note the order of events: the experts predictions come first, then the algorithm chooses an expert at the same time as the reality being revealed.

The goal is to minimize the number of mistakes, i.e., the number of times our prediction a^t differs from the outcome o^t.

Suppose we have 8 experts, and $\mathcal{E}^t = (0,1,0,0,0,1,1,0)$. If we follow the third expert and predict $a^t = 0$, but the actual outcome is $o^t = 1$, we make a mistake; if we would have picked the second expert, we would have been correct.

Fact 13.1. There exists an algorithm that makes at most $\lceil \log_2 N \rceil$ mistakes, if there is a perfect expert.

Proof. The algorithm just considers all the experts who have made no mistakes so far, and predicts what the majority of them predict. Note that every time we make a mistake, the number of experts who have not been wrong yet reduces by a factor of 2 or more. (And when we do not make a mistake, this number does not increase.) Since there is at least one perfect expert, we can make at most $\lceil \log_2 N \rceil$ mistakes. □

Show that any algorithm must make at least $\lceil \log_2 N \rceil$ mistakes in the worst case.

Fact 13.2. There is an algorithm that, on any sequence, makes at most $M \leq m^*(\lceil \log_2 N \rceil + 1) + \lceil \log_2 N \rceil$ mistakes, where m^* is the number of mistakes made by the best of these experts on this sequence.

Proof. Think of time as being divided into "epochs". In each epoch, we proceed as in the perfect expert scenario as in Fact 13.1: we keep track of all experts who have not yet made a mistake in that epoch, and predict the majority opinion. The set of experts halves (at least) with every mistake the algorithm makes. When the set becomes empty, we end the epoch, and start a new epoch with all the N experts.

Note that in each epoch, every expert makes at least one mistake. Therefore the number of completed epochs is at most m^*. Moreover, we make at most $\lceil \log_2 N \rceil + 1$ mistakes in each completed epoch, and at most $\lceil \log_2 N \rceil$ mistakes the last epoch, giving the result. □

However, this algorithm is very harsh and very myopic. Firstly, it penalizes even a single mistake by immediately discarding the expert. But then, at the end of an epoch, it wipes the slate clean and forgets the past performance of the experts. Maybe we should be gentler, but have a better memory?

13.2 The Weighted Majority Algorithm

This algorithm, due to Littlestone and Warmuth, is remarkable for its simplicity. We assign a weight w_i to each expert $i \in [N]$. Let $w_i^{(t)}$ denote the weight of expert i at the beginning of round t. Initially, all weights are 1, i.e., $w_i^{(1)} = 1$.

1. In round t, predict according to the weighted majority of experts. In other words, choose the outcome that maximizes the sum of weights of experts that predicted it. I.e.,

$$a^t \leftarrow \arg\max_{u \in U} \sum_{i:\text{expert } i \text{ predicts } u} w_i^{(t)}.$$

We break ties arbitrarily, say, by picking the first of the options that achieve the maximum.

2. Upon seeing the outcome, set

$$w_i^{(t+1)} = w_i^{(t)} \cdot \begin{cases} 1 & \text{if } i \text{ was correct} \\ \frac{1}{2} & \text{if } i \text{ was incorrect} \end{cases}.$$

Theorem 13.3. *For any sequence of predictions, the number of mistakes made by the weighted majority algorithm (WM) is at most*

$$2.41(m_i + \log_2 N),$$

where m_i is the number of mistakes made by expert i.

Proof. The proof uses a potential-function argument. Let

$$\Phi^t := \sum_{i \in [N]} w_i^{(t)}.$$

Note that

1. $\Phi_1 = N$, since the weights start off at 1,
2. $\Phi_{t+1} \le \Phi_t$ for all t, and
3. if Algorithm WM makes a mistake in round t, the sum of weights of the wrong experts is higher than the sum of the weights of the correct experts, so

$$\begin{aligned}\Phi^{t+1} &= \sum_{i \text{ wrong}} w_i^{(t+1)} + \sum_{i \text{ correct}} w_i^{(t+1)} \\ &= \frac{1}{2} \sum_{i \text{ wrong}} w_i^{(t)} + \sum_{i \text{ correct}} w_i^{(t)} \\ &= \Phi^t - \frac{1}{2} \sum_{i \text{ wrong}} w_i^{(t)} \\ &\le \frac{3}{4} \Phi^t\end{aligned}$$

If after T rounds, expert i has made m_i mistakes and WM has made M mistakes, then

$$\left(\frac{1}{2}\right)^{m_i} = w_i^{(T+1)} \le \Phi^{T+1} \le \Phi^1 \left(\frac{3}{4}\right)^M = N \left(\frac{3}{4}\right)^M.$$

Now taking logs, and rearranging,

$$M \le \frac{m_i + \log_2 N}{\log_2 \frac{4}{3}} \le 2.41(m_i + \log_2 N). \qquad \square$$

In other words, if the best of the N experts on this sequence was wrong m^* times, we would be wrong at most $2.41(m^* + \log_2 n)$ times. Note that we are much better on the multiplier in front of the m^* term than Fact 13.2 was, at the expense of being slightly worse on the multiplier in front of the $\log_2 N$ term.

We cannot hope to compare ourselves to the best way of dynamically choosing experts to follow. This result says that at least we do not much worse to the best static policy of choosing an expert—in fact, choosing the best expert in hindsight—and sticking with them. We'll improve our performance soon, but all our results will still compare to the best static policy for now.

13.2.1 *A Gentler Penalization*

Instead of penalizing each wrong expert by a factor of 1/2, we could penalize the experts by a factor of $(1 - \varepsilon)$. This allows us to trade off the multipliers on the m^* term and the logarithmic term.

Theorem 13.4. *For $\varepsilon \in (0, 1/2)$, penalizing each incorrect expert by a factor of $(1 - \varepsilon)$ guarantees that the number of mistakes made by MW is at most*

$$2(1+\varepsilon)m_i + O\left(\frac{\log N}{\varepsilon}\right).$$

Proof. Using an analysis identical to Theorem 13.3, we get that $\Phi^{t+1} \leq (1 - \frac{\varepsilon}{2})\Phi^t$ and therefore

$$(1-\varepsilon)^{m_i} \leq \Phi^{T+1} \leq \Phi^1 \Big(1 - \frac{\varepsilon}{2}\Big)^M = N \Big(1 - \frac{\varepsilon}{2}\Big)^M \leq N \exp\big(-\varepsilon M/2\big).$$

Now taking logs, and simplifying,

$$\begin{aligned} M &\leq \frac{-m_i \log(1-\varepsilon) + \ln N}{\varepsilon/2} \\ &\leq 2\,\frac{m_i(\varepsilon+\varepsilon^2)}{\varepsilon} + O\left(\frac{\log N}{\varepsilon}\right), \end{aligned}$$

because $-\ln(1-\varepsilon) = \varepsilon + \frac{\varepsilon^2}{2} + \frac{\varepsilon^3}{3} + \cdots \leq \varepsilon + \varepsilon^2$ for $\varepsilon \in [0,1]$. □

This shows that we can make our mistakes bound as close to $2m^*$ as we want, but this approach seems to have this inherent loss of a factor of 2. In fact, no deterministic strategy can do better than a factor of 2, as we show next.

Proposition 13.5. *No deterministic algorithm $\mathcal{A}$ can do better than a factor of 2, compared to the best expert.*

Proof. Note that if the algorithm is deterministic, its predictions are completely determined by the sequence seen thus far (and hence can also be computed by the adversary). Consider a scenario with two experts A,B, the first always predicts 1 and the second always predicts 0. Since $\mathcal{A}$ is deterministic, an adversary can fix the outcomes such that $\mathcal{A}$'s predictions are always wrong. Hence at least one of A and B will have an error rate of $\leq 1/2$, while $\mathcal{A}$'s error rate will be 1. □

13.3 Randomized Weighted Majority

Consider the proof of Proposition 13.5, but applied to the WM algorithm: the algorithm alternates between predicting 0 and 1, whereas the actual outcome is the opposite. The weights of the two experts remain approximately the same, but because we are deterministic, we choose the wrong one. What if we interpret the weights being equal as a signal that we should choose one of the two options with equal probability?

This is the idea behind the *Randomized Weighted Majority* algorithm (RMW) of Littlestone and Warmuth: the weights evolve in exactly the same way as in Theorem 13.4, but now the prediction at each time is drawn randomly proportional to the current weights of the experts. I.e., instead of Step 1 in that algorithm, we do the following:

$$\Pr[\text{action } u \text{ is picked}] = \frac{\sum_{i:\text{expert } i \text{ predicts } u} w_i^{(t)}}{\sum_i w_i^{(t)}}.$$

Note that the update of the weights proceeds exactly the same as previously.

Theorem 13.6. *Fix $\varepsilon \leq 1/2$. For any fixed sequence of predictions, the expected number of mistakes made by randomized weighted majority (RWM) is at most*

$$\mathbb{E}[M] \leq (1+\varepsilon)m_i + O\left(\frac{\log N}{\varepsilon}\right)$$

The quantity $\varepsilon m_i + O(\frac{\log N}{\varepsilon})$ gap between the algorithm's performance and that of the best expert is called the ***regret*** with respect to expert i.

Proof. The proof is an analysis of the weight evolution that is more careful than in Theorem 13.4. Again, the potential is $\Phi^t = \sum_i w_i^{(t)}$. Define

$$F^t := \frac{\sum_{i \text{ incorrect}} w_i^{(t)}}{\sum_i w_i^{(t)}}$$

to be the fraction of weight on incorrect experts at time t. Note that

$$\mathbb{E}[M] = \sum_{t \in [T]} F_t.$$

Indeed, we make a mistake at step t precisely with the probability F_t, since the adversary does not see our random choice when deciding on the actual outcome a^t. By our re-weighting rules,

$$\Phi^{t+1} = \Phi^t\left((1-F_t) + F_t(1-\varepsilon)\right) = \Phi^t(1-\varepsilon F_t)$$

Bounding the size of the potential after T steps,

$$(1-\varepsilon)^{m_i} \leq \Phi^{T+1} = \Phi^1 \prod_{t=1}^{T}(1-\varepsilon F_t) \leq N e^{-\varepsilon \sum F_t} = N e^{-\varepsilon \mathbb{E}[M]}$$

Now taking logs, we get $m_i \ln(1-\varepsilon) \leq \ln N - \varepsilon\mathbb{E}[M]$, using the approximation $-\log(1-\varepsilon) \leq \varepsilon + \varepsilon^2$ gives us

$$\mathbb{E}[M] \leq m_i(1+\varepsilon) + \frac{\ln N}{\varepsilon}. \qquad \square$$

13.3.1 *Classifying Adversaries for Randomized Algorithms*

In the above analysis, it was important that the actual random outcome was independent of the prediction of the algorithm. Let us formalize the power of the adversary:

Oblivious Adversary. Constructs entire sequence $\mathcal{E}^1, o^1, \mathcal{E}^2, o^2, \cdots$ upfront.

Adaptive Adversary. Sees the previous choices of the algorithm, but must choose o^t independently of our actual prediction a^t in round t. Hence, o^t can be a function of $\mathcal{E}^1, o^1, \ldots, \mathcal{E}^{t-1}, o^{t-1}, \mathcal{E}^t$, as well as of $a^1, \ldots, a^{t-1}$, but not of a^t.

The adversaries are equivalent on deterministic algorithms, because such an algorithm always outputs the same prediction and the oblivious adversary could have calculated a^t in advance when creating $\mathcal{E}^{t+1}$. They may be different for randomized algorithms. However, it turns out that RWM works in both models, because our predictions do not affect the weight updates and hence the future.

13.4 The Hedge Algorithm, and a Change in Perspective

Let's broaden the setting slightly, and consider the following *dot-product* game. In each round,

Define the *probability simplex* as
$$\Delta_N := \{x \in [0,1]^N \mid \sum_i x_i = 1\}.$$

1. The algorithm produces a vector of probabilities
$$p^t = (p_1^t, p_2^t, \cdots, p_N^t) \in \Delta_N.$$
2. The adversary produces
$$\ell^t = (\ell_1^t, \ell_2^t, \cdots, \ell_N^t) \in [-1,1]^N.$$
3. The loss of the algorithm in this round is $\langle \ell^t, p^t \rangle$.

This equivalence between randomized and fractional algorithms is a common theme in algorithm design, especially in approximation and online algorithms.

We can move between this "fractional" model where we play a point in the probability simplex Δ_N, and the randomized model of the previous section (with an adaptive adversary), where we must play a single expert (which is a vertex of the simplex Δ_N. Indeed, setting ℓ^t to be a vector of 0s and 1s can capture whether an expert is correct or not, and we can set

$$p_i^t = \Pr[\text{algorithm plays expert } i \text{ at time } t]$$

to deduce that
$$\Pr[\text{mistake at time } t] = \left\langle \ell^t, p^t \right\rangle.$$

13.4.1 The Hedge Algorithm

The Hedge algorithm starts with weights $w_i^1 = 1$ for all experts i. In each round t, it defines $p^t \in \Delta_N$ using:

$$p_i^t \leftarrow \frac{w_i^t}{\sum_j w_j^t}, \tag{13.1}$$

and updates weights as follows:

$$w_i^{t+1} \leftarrow w_i^t \cdot \exp(-\varepsilon \ell_i^t). \tag{13.2}$$

Theorem 13.7. *Consider a fixed $\varepsilon \leq 1/2$. For any sequences of loss vectors in $[-1,1]^N$ and for all indices $i \in [N]$, the Hedge algorithm guarantees:*

$$\sum_{i=1}^T \langle p^t, \ell^t \rangle \leq \sum_{t=1}^T \ell_i^t + \varepsilon T + \frac{\ln N}{\varepsilon}$$

Proof. As in previous proofs, let $\Phi^t = \sum_j w_j^t$, so that $\Phi^1 = N$, and

$$\begin{aligned}
\Phi^{t+1} &= \sum w_i^{t+1} = \sum_i w_i^t e^{-\varepsilon \ell_i^t} \\
&\le \sum_i w_i^t \big(1 - \varepsilon \ell_i^t + \varepsilon^2 (\ell_i^t)^2\big) && \text{(using } e^x \le 1 + x + x^2 \; \forall x \in [-1,1]) \\
&\le \sum w_i^t (1+\varepsilon^2) - \varepsilon \sum w_i^t \ell_i^t && \text{(because } |\ell_i^t| \le 1) \\
&= (1+\varepsilon^2)\Phi^t - \varepsilon \Phi^t \, \langle p^t, \ell^t \rangle && \text{(because } w_i^t = p_i^t \cdot \Phi^t) \\
&= \Phi^t \, \big(1 + \varepsilon^2 - \varepsilon \langle p^t, \ell^t \rangle\big) \\
&\le \Phi^t \, e^{\varepsilon^2 - \varepsilon \langle p^t, \ell^t \rangle} && \text{(using } 1 + x \le e^x)
\end{aligned}$$

Again, comparing to the final weight of the i^{th} coordinate,

$$e^{-\varepsilon \sum \ell_i^t} = w_i^{(t+1)} \le \Phi^{T+1} \le \Phi^1 \, e^{\varepsilon^2 T - \varepsilon \sum \langle p^t, \ell_i^t \rangle};$$

now using $\Phi^1 = N$ and taking logs proves the claim. □

Moreover, choosing $\varepsilon = \sqrt{\frac{\ln N}{T}}$ gives $\varepsilon T + \frac{\ln N}{\varepsilon} = 2\sqrt{T \ln N}$, and the regret term is *concave and sublinear in time* T. This suggests that the further we run the algorithm, the quicker the average regret goes to zero, which suggests the algorithm is in some sense "learning".

13.4.2 *Two Useful Corollaries*

The following corollary will be useful in many contexts: it just flips Theorem 13.7 on its head, and shows that the average regret is small after sufficiently many steps.

Corollary 13.8. *For $T \ge \frac{4 \log N}{\varepsilon^2}$, the average loss of the Hedge algorithm is*

$$\begin{aligned}
\frac{1}{T} \sum_t \langle p^t, \ell^t \rangle &\le \min_i \; \frac{1}{T} \sum_t \ell_i^t + \varepsilon \\
&= \min_{p^* \in \Delta_N} \; \frac{1}{T} \sum_t \langle \ell^t, p^* \rangle + \varepsilon.
\end{aligned}$$

The viewpoint of the last expression is useful, since it indicates that the dynamic strategy given by Hedge for the dot-product game is comparable (in the sense of having tiny regret) against any fixed strategy p^* in the probability simplex.

Finally, we state a further corollary that is useful in future lectures. It can be proved by running Corollary 13.8 with losses $\ell^t = -g^t/\rho$.

Corollary 13.9 (Average Gain). *Let $\rho \ge 1$ and $\varepsilon \in (0, 1/2)$. For any sequence of gain vectors $g^1, \ldots, g^T \in [-\rho, \rho]^N$ with $T \ge \frac{4\rho^2 \ln N}{\varepsilon^2}$, the gains version of the Hedge algorithm produces probability vectors $p^t \in \Delta_N$ such that*

$$\frac{1}{T} \sum_{t=1}^{T} \langle g^t, p^t \rangle \ge \max_{i \in [N]} \; \frac{1}{T} \sum_{t=1}^{T} \langle g^t, e_i \rangle - \varepsilon.$$

In passing we mention that if the gains or losses lie in the range $[-\gamma, \rho]$, then we can get an asymmetric guarantee of $T \geq \frac{4\gamma\rho \ln N}{\varepsilon^2}$.

13.5 *Optional: The Bandit Setting*

The model of experts or the dot-product problem is often called the ***full-information*** model, because the algorithm gets to see the entire loss vector ℓ^t at each step. (Recall that we view the entries of the probability vector p^t played by the algorithm as the probability of playing each of the actions, and hence $\langle \ell^t, p^t \rangle$ is just the expected loss incurred by the algorithm. Now we consider a different model, where the algorithm only gets to see the loss of the action it plays. Specifically, in each round,

1. The algorithm again produces a vector of probabilities

$$p^t = (p^t_1, p^t_2, \cdots, p^t_N) \in \Delta_N.$$

   It then chooses an action $a^t \in [N]$ with these marginal probabilities.

2. *In parallel*, the adversary produces

$$\ell^t = (\ell^t_1, \ell^t_2, \cdots, \ell^t_N) \in [-1, 1]^N.$$

   However, now the algorithm only gets to see the loss $\ell^t_{a^t}$ corresponding to the action chosen by the algorithm, and not the entire loss vector.

This limited-information setting is called the ***bandit*** setting.

The name comes from the analysis of slot machines, which are affectionately known as “one-armed bandits”.

13.5.1 *The Exp3 Algorithm*

Surprisingly, we can obtain algorithms for the bandit setting from algorithms for the experts setting, by simply “hallucinating” the cost vector, using an idea called ***importance sampling***. This causes the parameters to degrade, however.

Indeed, consider the following algorithm: we run an instance $\mathcal{A}$ of the RWM algorithm, which is in the full information model. So at each timestep,

1. $\mathcal{A}$ produces a probability vector $p^t \in \Delta_N$.

2. We choose an expert $I^t \in [N]$, where

$$\Pr[I^t = i] = q^t_i := \gamma \cdot \frac{1}{N} + (1 - \gamma) \cdot p^t_i.$$

   I.e., with probability γ we pick a uniformly random expert, else we follow the suggestion given by p^t.

3. We get back the loss value $\ell^t_{I^t}$ for this chosen expert.
4. We construct an "estimated loss" $\hat{\ell}^t \in [0,1]^N$ by setting
$$\tilde{\ell}^t_j = \begin{cases} \frac{\ell^t_j}{q^t_j} & \text{if } j = I^t \\ 0 & \text{if } j \neq I^t \end{cases}.$$
We now feed $\tilde{\ell}^t$ to the RWM instance $\mathcal{A}$, and go back to Step 1.

We now show this algorithm achieves low regret. The first observation is that the estimated loss vector is an unbiased estimate of the actual loss, just because of the way we reweighted the answer by the inverse of the probability of picking it. Indeed,
$$\mathbb{E}[\tilde{\ell}^t_i] = \frac{\ell^t_i}{q^t_i} \cdot q^t_i + 0 \cdot (1 - q^t_i) = \ell^t_i. \tag{13.3}$$
Since each true loss value lies in $[-1,1]$, and each probability value is at least γ/N, the absolute value of each entry in the $\tilde{\ell}$ vectors is at most N/γ. Now, since we run RWM on these estimated loss vectors belonging to $[0, N/\gamma]^N$, we know that
$$\sum_t \langle p^t, \tilde{\ell}^t \rangle \leq \sum_t \tilde{\ell}^t_i + \frac{N}{\gamma}\left(\varepsilon T + \frac{\log N}{\varepsilon}\right).$$
Taking expectations over both sides, and using (13.3),
$$\sum_t \langle p^t, \ell^t \rangle \leq \sum_t \ell^t_i + \frac{N}{\gamma}\left(\varepsilon T + \frac{\log N}{\varepsilon}\right).$$
However, the LHS is not our real loss, since we chose I^t according to q^t and not p^t. This means our expected total loss is really
$$\begin{aligned}\sum_t \langle q^t, \ell^t \rangle &= (1-\gamma) \sum_t \langle p^t, \ell^t \rangle + \frac{\gamma}{N} \sum_t \langle \mathbb{1}, \ell^t \rangle \\ &\leq \sum_t \ell^t_i + \frac{N}{\gamma}\left(\varepsilon T + \frac{\log N}{\varepsilon}\right) + \gamma T.\end{aligned}$$
Now choosing $\varepsilon = \sqrt{\frac{\log N}{T}}$ and $\gamma = \sqrt{N}\left(\frac{\log N}{T}\right)^{1/4}$ gives us a regret of $\approx N^{1/2}T^{3/4}$. The interesting fact here is that the regret is again sub-linear in T, the number of timesteps: this means that as $T \to \infty$, the per-step regret tends to zero.

The dependence on N, the number of experts/options, is now polynomial, instead of being logarithmic as in the full-information case. This is necessary: there is a lower bound of $\Omega(\sqrt{NT})$ in the bandit setting. And indeed, the Exp3 algorithm itself achieves a near-optimal regret bound of $O(\sqrt{NT \log N})$; we can show this by using a finer analysis of Hedge that makes more careful approximations. We defer these improvements for now, and instead give an application of this bandit setting to a problem in item pricing.

13.5.2 Item Pricing via Bandits

To be added in.

14
Solving Linear Programs using Experts

We can now use the low-regret algorithms for the experts problem to show how to approximately solve linear programs (LPs). As a warm-up, we use it to solve two-player zero-sum games, which are a special case of LPs.

In fact, zero-sum games are equivalent to linear programming, see this work of Ilan Adler. Is there an earlier reference?

14.1 (Two-Player) Zero-Sum Games

There are two players in such a game, traditionally called the "row player" and the "column player". Each of them has some set of actions: the row player with m actions (associated with the set $[m]$), and the column player with the n actions in $[n]$. Finally, we have a payoff matrix $M \in \mathbb{R}^{m \times n}$. In a play of the game, the row player chooses a row $i \in [m]$, and simultaneously, the column player chooses a column $j \in [n]$. If this happens, the row player gets $M_{i,j}$, and the column player loses $M_{i,j}$. The winnings of the two players sum to zero, and so we imagine that the payoff is *from* the row player *to* the column player.

Henceforth, when we talk about payoffs, these will always refer to payoffs to the row player from the column player. This payoff may be negative, which would capture situations where the column player does better.

14.1.1 Strategies, and Best-Response

Each player is allowed to have a randomized strategy. Given strategies $p \in \Delta_m$ for the row player, and $q \in \Delta_n$ for the column player, the expected payoff (to the row player) is

$$\mathbb{E}[\text{payoff to row}] = p^\intercal M q = \sum_{i,j} p_i q_j M_{i,j}.$$

The row player wants to maximize this value, while the column player wants to minimize it.

Suppose the row player fixes a strategy $p \in \Delta_m$. Knowing p, the column player can choose an action to minimize the expected payoff:

$$C(p) := \min_{q \in \Delta_n} p^\intercal M q = \min_{j \in [n]} p^\intercal M e_j.$$

The equality holds because the expected payoff is linear, and hence the column player's best strategy is to choose a column that minimizes the expected payoff. The column player is said to be playing their ***best response***. Analogously, if the column player fixes a strategy $q \in \Delta_n$, the row player can maximize the expected payoff by playing their own best response:

$$R(q) := \max_{p \in \Delta_m} p^\intercal M q = \max_{i \in [m]} e_i^\intercal M q.$$

Now, the row player would love to play the strategy p such that even if the column player plays best-response, the payoff is as large as possible: i.e., it wants to achieve

$$\max_{p \in \Delta_m} C(p).$$

Similarly, the column player wants to choose q to minimize the payoff against a best-response row player, i.e., to achieve

$$\min_{q \in \Delta_n} R(q).$$

Lemma 14.1. *For any $p \in \Delta_m, q \in \Delta_n$, we have*

$$C(p) \leq R(q) \tag{14.1}$$

Proof. Intuitively, since the column player commits to a strategy q, it hence gives more power to the row player. Formally, the row player could always play strategy p in response to q, and hence could always get value $C(p)$. But $R(q)$ is the best response, which could be even higher. □

Interestingly, there always exist strategies $p \in \Delta_m, q \in \Delta_n$ which achieve equality. This is formalized by the following theorem:

Theorem 14.2 (Von Neumann's Minimax Theorem)**.** *For any finite zero-sum game $M \in \mathbb{R}^{m \times n}$,*

$$\max_{p \in \Delta_m} C(p) = \min_{q \in \Delta_n} R(q).$$

This common value V is called the value of the game M.

Proof. We assume for the sake of contradiction that $\exists M \in [-1,1]^{m \times n}$ such that $\max_{p \in \Delta_m} C(p) \leq \min_{q \in \Delta_n} R(q) - \delta$ for some $\delta > 0$. (The assumption that $M_{ij} \in [-1,1]$ follows by scaling. Now we use the fact that the average regret of the Hedge algorithm tends to zero to construct strategies $\widehat{p}$ and $\widehat{q}$ that have $R(\widehat{q}) - C(\widehat{p}) < \delta$, thereby giving us a contradiction.

We consider an instance of the experts problem, with m experts, one for each row of M. At each time step t, the row player produces

$p^t \in \Delta_m$. Initially $p^1 = \left(\frac{1}{m}, \dots, \frac{1}{m}\right)$, which represents that the row player chooses each row with equal probability, when they have no information to work with.

At each time t, the column player plays the best-response to p^t, i.e.,

$$j_t := \arg\max_{j \in [n]} (p^t)^\intercal M e_j.$$

This defines a gain vector for the row player:

$$g_t := M e_{j_t},$$

which is the j^{th} column of M. The row player uses this to update the weights and get p_{t+1}, etc. Define

$$\widehat{p} := \frac{1}{T} \sum_{t=1}^{T} p^t \quad \text{and} \quad \widehat{q} := \frac{1}{T} \sum_{t=1}^{T} e_{j_t}$$

to be the average long-term plays of the row player, and of the best responses of the column player to those plays. We know that

$$C(\widehat{p}) \le R(\widehat{q})$$

by (14.1). But by Corollary 13.9, after $T \ge \frac{4 \ln m}{\varepsilon^2}$ steps,

$$\begin{aligned}
\frac{1}{T}\sum_t \langle p^t, g^t \rangle &\ge \max_i \frac{1}{T} \sum_t \langle e_i, g^t \rangle - \varepsilon && \text{(by Hedge)} \\
&= \max_i \left\langle e_i, \frac{1}{T} \sum_t g^t \right\rangle - \varepsilon \\
&= \max_i \left\langle e_i, M\Big(\frac{1}{T}\sum_t e_{j_t}\Big) \right\rangle - \varepsilon && \text{(by definition of } g_t) \\
&= \max_i \langle e_i, M\widehat{y} \rangle - \varepsilon \\
&= R(\widehat{q}) - \varepsilon.
\end{aligned}$$

Since p^t is the row player's strategy, and C is concave (i.e., the payoff on the average strategy $\widehat{p}$ is no more than the average of the payoffs:

$$\frac{1}{T}\sum \langle p^t, g^t \rangle = \frac{1}{T} \sum C(p^t) \le C\Big(\frac{1}{T}\sum p^t\Big) = C(\widehat{x}).$$

To see this, recall that

$$C(p) := \min_q p^\intercal M q.$$

Let q^* be the optimal value of q that minimizes $C(p)$. Then for any $a, b \in \Delta_m$, we have that

$$\begin{aligned}
C(a+b) &= (a+b)^\intercal M q^* = a^\intercal M q^* + b^\intercal M q^* \\
&\ge \min_q a^\intercal M q + \min_q b^\intercal M q = C(a) + C(b)
\end{aligned}$$

Putting it all together:

$$R(\widehat{q}) - \varepsilon \le C(\widehat{p}) \le R(\widehat{q}).$$

Now for any $\delta > 0$ we can choose $\varepsilon < \delta$ to get the contradiction. □

Observe that the proof gives us an explicit algorithm to find strategies $\widehat{p}, \widehat{q}$ that have a small gap. The minimax theorem is also implied by strong duality of linear programs: indeed, we can write $\min_{q \in \Delta_n} R(q)$ as a linear program, take its dual and observe that it computes $\min_{p \in \Delta_m} C(p)$. The natural question is: we can solve linear programs using low-regret algorithms. We now show how to do this. We should get a clean proof of strong duality this way?

14.2 Solving LPs Approximately

Consider an LP with constraint matrix $A \in \mathbb{R}^{m \times n}$:

$$\begin{aligned} \max \ \langle c, x\rangle & \\ Ax \leq b & \\ x \geq 0 & \end{aligned} \tag{14.2}$$

Suppose x^* is an optimal solution, with $OPT := \langle c, x^* \rangle$. Let $K \subseteq \mathbb{R}^n$ be the polyhedron defined by the "easy" constraints, i.e.,

$$K := \{x \in \mathbb{R}^n \mid \langle c, x\rangle = OPT, x \geq 0\},$$

where OPT is found by binary search over possible objective values. Binary search over the reals is typically not a good idea, since it may never reach the answer. (E.g., searching for $1/3$ by binary search over $[0, 1]$.) However, we defer this issue for now, and imagine we know the value of OPT. We now use low-regret algorithms to find $\widehat{x} \in K$ such that $\langle a_i, x\rangle \leq b_i + \varepsilon$ for all $i \in [m]$.

The fix to the "binary search over reals" problem is this: the optimal value of a linear program in n dimensions where all numbers integers using at most b bits is a rational p/q, whereboth p, q use at most poly(nb) bits. So once we the granularity of the search is fine enough, there is a unique rational close the query point, and we can snap to it. See, e.g., the problem on finding negative cycles in the homeworks.

14.2.1 The Oracle

The one assumption we make is that we can solve the feasibility problem obtained by intersecting the "easy" constraints K with a single linear constraint. Suppose $\alpha \in \mathbb{R}^n, \beta \in \mathbb{R}$, then we want to solve the problem:

$$\textsc{Oracle}: \text{find a point } x \in K \cap \{x \mid \langle \alpha, x\rangle \leq \beta\}. \tag{14.3}$$

Proposition 14.3. *There is an $O(n)$-time algorithm to solve (14.3), when $K = \{x \geq 0 \mid \langle c, x\rangle = OPT\}$.*

Proof. We give the proof only for the case where $c_i > 0$ for all i; the general case is left as an exercise. Let $j^* := \arg\min_j \alpha_j/c_j$, and define $x = (OPT/c_{j^*})\mathbf{e}_{j^*}$. Say "infeasible" if x does not satisfy $\langle \alpha, x\rangle \leq \beta$, else return x. □

Of course, this problem can be solved in time linear in the number of variables (as Proposition 14.3 above shows), but the situation can be more interesting when the number of variables is large. For instance, when we solve flow LPs, the number of variables will be exponential in the size of the graph, yet the oracle will be implementable in time poly(n).

14.2.2 The Algorithm

The key idea to solving general LPs is then similar to that for zero-sum games. We have m experts, one corresponding to each constraint. In each round, we combine the multiple constraints using a weighted sum, we call the above oracle on this single-constraint LP to get a solution, we construct a gain vector from this solution and feed this to Hedge, which then updates the weights that we use for the next round. The gain of an expert in a round is based based on how badly the constraint was violated by the current solution. The intuition is simple: greater violation means more gain, and hence more weight in the next iteration, which forces us to not violate the constraint as much.

An upper bound on the maximum possible violation is the ***width*** ρ of the LP, defined by

$$\rho := \max_{x \in K, i \in [m]} \{ | \langle a_i, x \rangle - b_i | \}. \qquad (14.4)$$

We assume that $\rho \geq 1$.

Algorithm 12: LP-Solver

```
12.1  $p^1 \leftarrow (1/m, \ldots, 1/m)$. $T \leftarrow \Theta(\rho^2 \ln m / \varepsilon^2)$
12.2  for $t = 1$ to $T$ do
12.3      Define $\alpha^t := \sum_{i=1}^m p_i^t a_i \in \mathbb{R}^n$ and $\beta^t = \sum_{i=1}^m p_i^t b_i \in \mathbb{R}$.
12.4      Use ORACLE to find $x \in K \cap \{ \langle \alpha^t, x^t \rangle \leq \beta^t \}$.
12.5      if oracle says infeasible then
12.6          return infeasible
12.7      else
12.8          $g_i^t \leftarrow \langle a_i, x^t \rangle - b_i$ for all $i$.
12.9          feed $g^t$ to Hedge($\varepsilon$) to get $p^{t+1}$.
12.10 return $\widehat{x} \leftarrow (x^1 + \cdots + x^T)/T$.
```

14.2.3 *The Analysis*

Theorem 14.4. *Fix $0 \leq \varepsilon \leq 1/4$. Then Algorithm 12 calls the oracle $O(\rho^2 \ln m / \varepsilon^2)$ times, and either correctly returns "infeasible", or returns $\widehat{x} \in K$ such that*

$$A\widehat{x} \leq b - \varepsilon \mathbb{1}.$$

Proof. Observe that if x^* is feasible for the original LP (14.2) then it is feasible for any of the calls to the oracle, since it satisfies any positive linear combination of the constraints. Hence, we are correct if we ever return "infeasible". Moreover, $x^t \in K$ in each iteration, and $\widehat{x}$ is an average of x^t's, so it also lies in K by convexity. So it remains to show that $\widehat{x}$ approximately satisfies the other linear constraints.

Recall the guarantee from Corollary 13.9:

$$\frac{1}{T} \sum_t \langle p^t, g^t \rangle \geq \max_i \frac{1}{T} \sum_t \langle e_i, g^t \rangle - \varepsilon, \qquad (14.5)$$

for precisely the choice of T in Algorithm 12, since the definition of width in (14.4) ensures that $g^t \in [-\rho, \rho]^m$.

Let $i \in [m]$, and recall the definitions of $\alpha^t = \sum_{i=1}^m p_i^t a_i$, $\beta^t = \sum_{i=1}^m p_i^t b_i$, and $g^t = Ax^t - b$ from the algorithm. Then

$$\begin{aligned}\langle p^t, g^t\rangle &= \langle p^t, Ax^t - b\rangle \\ &= \langle p^t, Ax^t\rangle - \langle p^t, b\rangle \\ &= \langle \alpha^t, x^t\rangle - \beta^t \leq 0,\end{aligned}$$

the last inequality because x^t satisfies the single linear constraint $\langle \alpha^t, x\rangle \leq \beta^t$. Averaging over all times, the left hand side of (14.5) is

$$\frac{1}{T}\sum_{t=1}^{T}\langle p^t, g^t\rangle \leq 0.$$

However, the average on the RHS in (14.5) for constraint/expert i is:

$$\begin{aligned}\frac{1}{T}\sum_{t=1}^{T}\langle e_i, g^t\rangle &= \left\langle e_i, \frac{1}{T}\sum_{t=1}^{T} g^t\right\rangle \\ &= \frac{1}{T}\sum_{t=1}^{T}\left(\langle a_i, \widehat{x}^t\rangle - b_i\right) \\ &= \langle a_i, \widehat{x}\rangle - b_i.\end{aligned}$$

Substituting into (14.5) we have

$$0 \geq \frac{1}{T}\sum_{t=1}^{T}\langle p^t, g^t\rangle \geq \max_i \left(\langle a_i, \widehat{x}\rangle - b_i\right) - \varepsilon.$$

This shows that $A\widehat{x} \leq b + \varepsilon\mathbb{1}$. □

14.2.4 *A Small Extension: Approximate Oracles*

Recall the definition of the problem width from (14.4). A few comments:

- In the above analysis, we do not care about the maximum value of $|a_i^\intercal x - b_i|$ over all points $x \in K$, but only about the largest this expression gets over points that are potentially returned by the oracle. This seems a pedantic point, but if there are many solutions to (14.3), we can return one with small width. But we can do more, as the next point outlines.

- We can also relax the oracle to satisfy $\langle \alpha, x\rangle \leq \beta + \delta$ for some small $\delta > 0$ instead. Define the ***width*** of the LP with respect such a relaxed oracle to be

$$\rho_{\text{rlx}} := \max_{i\in[m], x \text{ returned by relaxed oracle}} \{|a_i^\intercal x - b_i|\}. \tag{14.6}$$

Now running the algorithm with a relaxed oracle gives us a slightly worse guarantee that

$$A\widehat{x} \leq b + (\varepsilon + \delta)\mathbb{1},$$

but now the number of calls to the relaxed oracle can be even smaller, namely $O(\rho_{\mathrm{rlx}}^2 \ln m/\varepsilon^2)$.

- Of course, if we violations can be bounded in some better way, e.g., if we can ensure that violations are always positive or negative, then we can give stronger bounds on the regret, and hence reduce the number of calls even further. Details to come.

All these improvements will be crucial in the upcoming applications.

15 Approximate Max-Flows using Experts

We now use low-regret multiplicative-weight algorithms to give approximate solutions to the s-t-maximum-flow problem. In the previous chapter, we already saw how to get approximate solutions to general linear programs. We now show how a closer look at those algorithms give us improvements in the running time (albeit in the setting of undirected graphs), which go beyond those known via usual "combinatorial" techniques. The first set of results we give will hold for directed graphs as well, but the improved results will only hold for undirected graphs.

15.1 The Maximum Flow Problem

In the *s-t maximum flow problem*, we are given a graph $G = (V, E)$, and distinguished vertices s and t. Each edge has a capacities $u_e \geq 0$; we will mostly focus on the unit-capacity case of $u_e = 1$ in this chapter. The graph may be directed or undirected; an undirected edge can be modeled by two oppositely directed edges having the same capacity.

Recall that ***an s-t flow*** is an assignment $f : E \to \mathbb{R}^+$ such that

(a) $f(e) \in [0, u_e]$, i.e., capacity-respecting on all edges, and

(b) $\sum_{e=(u,v)\in E} f(e) = \sum_{e=(v,w)\in E} f(e)$, i.e., flow-conservation at all non-$\{s,t\}$-nodes.

The ***value*** of flow f is $\sum_{e=(s,w)\in E} f(e) - \sum_{e=(u,s)\in E} f(e)$, the net amount of flow leaving the source node s. The goal is to find an s-t flow in the network, that satisfies the edge capacities, and has maximum value.

Algorithms by Edmonds and Karp, by Yefim Dinitz, and many others can solve the s-t max-flow problem exactly in polynomial time. For the special case of (directed) graphs with unit capacities, Shimon Even and Bob Tarjan, and independently, Alexander Karzanov showed in 1975 that the Ford-Fulkerson algorithm finds the maximum flow in time $O(m \cdot \min(m^{1/2}, n^{2/3}))$. This runtime was eventually matched for general capacities (up to some poly-logarithmic factors) by an algorithm of Andrew Goldberg and Satish Rao in 1998. For the special case of $m = O(n)$, these results gave a runtime of $O(m^{1.5})$, but nothing better was known even for approximate max-flows, even for unit-capacity undirected graphs—until a breakthrough in 2010, which we will see at the end of this chapter.

15.1.1 A Linear Program for Maximum Flow

We formulate the max-flow problem as a linear program. There are many ways to do this, and we choose to write an enormous LP for it. Let $\mathcal{P}$ be the set of all s-t paths in G. Define a variable f_P denoting the amount of flow going on path $P \in \mathcal{P}$. We can now write:

$$\begin{aligned} \max \quad & \sum_{P \in \mathcal{P}} f_P & \quad (15.1) \\ & \sum_{P: e \in P} f_P \leq u_e & \forall e \in E \\ & f_P \geq 0 & \forall P \in \mathcal{P} \end{aligned}$$

The first set of constraints says that for each edge e, the contribution of all possible flows is no greater than the capacity u_e of that edge. The second set of constraints say that the contribution from each path must be non-negative. This is a gigantic linear program: there could be an exponential number of s-t paths. As we see, this will not be a hurdle.

15.2 A First Algorithm using the MW Framework

To using the framework from the previous section, we just need to implement the Oracle: i.e., we solve a problem with a single "average" constraint, as in (14.3). Specifically, suppose we want a flow value of F, then the "easy" constraints are:

$$K := \{f \mid \sum_{P \in \mathcal{P}} f_p = F, f \geq 0\}.$$

Moreover, the constraint $\langle \alpha, f \rangle \leq \beta$ is not an arbitrary constraint—it is one obtained by combining the original constraints. Specifically, given a vector $p^t \in \Delta_m$, the average constraint is obtained by the convex combination of these constraints:

$$\sum_{e \in E} p_e^t \Big(\sum_{P: e \in P} f_P \leq u_e \Big), \qquad (15.2)$$

where f_e represents the net flow over edge e. By swapping order of summations, and using the unit capacity assumption, we obtain

$$\sum_{P \in \mathcal{P}} f_P \Big(\sum_{e \in P} p_e^t \Big) \leq \sum_e p_e^t u_e = 1.$$

Now, the inner summation is the path length of P with respect to edge weights p_e^t, which we denote by $\text{len}_t(P) := \sum_{e \in P} p_e^t$. The constraint now becomes:

$$\sum_{P \in \mathcal{P}} f_P \, \text{len}_t(P) \leq 1, \qquad (15.3)$$

and we want a point $f \in K$ satisfying it. The best way to satisfy it is to place all F units of flow on the shortest path P, and zero everywhere else; we output "infeasible" if the shortest-path has a length more than 1. This step can be done by a single call to Dijkstra's algorithm, which takes $O(m + n \log n)$ time.

We already argued in Theorem 14.4 that if there exists a feasible flow of value F in the graph, we never output "infeasible". Here is a direct proof.

If there is a flow of value F, there are F disjoint s-t paths. The vector $p^t \in \Delta_m$, so its values sum to 1. Hence, one of the F s-t paths P^* has $\sum_{e \in P} p_e^t \leq 1/F$. Setting $f_P = F$ for that path satisfies the constraint.

Now Theorem 14.4 says that running this algorithm for $\Theta\Big(\frac{\rho^2 \log m}{\varepsilon^2}\Big)$ iterations gives a solution $f \in K$, that violates the constraints by an additive ε. Hence, the scaled-down flow $f/(1+\varepsilon)$ would satisfy all the capacity constraints, and have flow value $F/(1+\varepsilon)$, which is what we wanted. To complete the runtime analysis, it remains to bound the value of ρ, the maximum amount by which any constraint gets violated by a solution from the oracle. Since we send all the F units of flow on a single edge, the maximum violation is $F - 1$. Hence the total runtime is at most

$$O(m + n \log n) \cdot \frac{F^2 \log m}{\varepsilon^2}.$$

Moreover, the maximum flow F is m, by the unit capacity assumption, which gives us an upper bound of $O(m^3 \, \text{poly}(\log m/\varepsilon))$.

15.2.1 *A Better Bound, via an Asymmetric Guarantee for Hedge*

Let us state (without proof, for now) a refined version of the Hedge algorithm for the case of asymmetric gains, where the gains lie in the range $[-\gamma, \rho]$.

Theorem 15.1 (Asymmetric Hedge). *Let $\varepsilon \in (0, 1/2)$, and $\gamma, \rho \geq 1$. Moreover, let $T \geq \frac{\Theta(\gamma\rho \ln N)}{\varepsilon^2}$. There exists an algorithm for the experts problem such that for every sequence $g^1, \ldots, g^T$ of gains with $g \in [-\gamma, \rho]^N$, produces probability vectors $\{p^t \in \Delta_N\}_{t \in [T]}$ online such that for each i:*

$$\frac{1}{T} \sum_{t=1}^{T} \langle g^t, p^t \rangle \geq \frac{1}{T} \sum_{t=1}^{T} \langle g^t, e_i \rangle - \varepsilon.$$

The proof is a careful (though not difficult) reworking of the standard proof for Hedge. (We will add it soon; a hand-written proof is on the webpage.) Moreover, we can use this statement to prove that the approximate LP solver can stop after $\frac{\Theta(\gamma\rho \ln m)}{\varepsilon^2}$ calls to an oracle, as long as each of the oracle's answer x guarantee that $(Ax)_i - b_i \in [-\gamma, \rho]$.

Since a solution f found by our shortest-path oracle sends all F flow on a single path, and all capacities are 1, we have $\gamma = 1$ and $\rho = F - 1 \leq F$. The runtime now becomes

$$O(m + n \log n) \cdot \frac{1 \cdot (F-1) \log m}{\varepsilon^2}.$$

Again, using the naïve bound of $F \leq m$, we have a runtime of $O(m^2 \operatorname{poly}(\log m/\varepsilon))$ to find a $(1+\varepsilon)$-approximate max-flow, even in directed graphs.

15.2.2 *An Intuitive Explanation and an Example*

Observe that the algorithm repeats the following natural process:

1. it finds a shortest path in the graph,
2. it pushes F units of flow on it, and then
3. it increases the length of each edge on this path multiplicatively.

The factor happens to be $(1+\varepsilon/F)$, because of how we rescale the gains, but that does not matter for this intuition.

This length-increase makes congested edges (those with a lot of flow) be much longer, and hence become very undesirable when searching for short paths. Note that the process is repeated some number of times, and then we average all the flows we find. So unlike usual network flow algorithms based on residual networks, these algorithms are truly greedy and cannot "undo" past actions (which is what pushing flow in residual flow networks does, when we use an arc backwards). This means these MW-based algorithms must ensure that very little flow goes on edges that are "wasteful".

To illustrate this point, consider an example commonly used to show that the greedy algorithm does not work for max-flow: Change the figure to make it more instructive.

15.3 *Finding Max-Flows using Electrical Flows*

The approach of the previous sections suggests a way to get faster algorithms for max-flow: *reduce the width of the oracle*. The approach of the above section was to push all F flow along a single path, which is why we have a width of $\Omega(F)$. Can we implement the oracle in a way that spreads the flow over several paths, and hence has smaller width? Of course, one such solution is to use the max-flow as the oracle response, but that would defeat the purpose of the MW approach. Indeed, we want a fast way of implementing the oracle.

We use the notation $\widetilde{O}(f(n))$ to hide factors that are poly-logarithmic in $f(n)$. E.g., $O(n \log^2 n)$ lies in $\widetilde{O}(n)$, and $O(\log n \log\log n)$ lies in $\widetilde{O}(\log n)$, etc.

For undirected graphs, one good solution turns out to be to use ***electral flows***: to model the graph as an electrical network, set a voltage difference between s and t, and compute how electrical current would flow between them. We now show how this approach gives us an $\widetilde{O}(m^{1.5}/\varepsilon^{O(1)})$-time algorithm quite easily; then with some more work, we improved this to get a runtime of $\widetilde{O}(m^{4/3}/\varepsilon^{O(1)})$. While we focus only on unit-capacity graphs, the algorithm can be extended to all undirected graphs with a further loss of poly-logarithmic factors in the maximum capacity, and moreover to get a runtime of $\widetilde{O}(mn^{1/3}/\operatorname{poly}(\varepsilon))$.

At the time this result was announced (by Christiano et al.), it was the fastest algorithm for the approximate maximum s-t-problem in undirected graphs. Since then, works by Jonah Sherman, and Kelner et al. gave $O(m^{1+o(1)}/\varepsilon^{O(1)})$-time algorithms for the problem. The current best runtime is $O(m \operatorname{poly}\log m/\varepsilon^{O(1)})$-time, due to Richard Peng.

Christiano, Kelner, Madry, Spielman, Teng (2010)

Sherman (2013)

Kelner, Lee, Oracchia, Sidford (2013)

Peng (2014)

Interestingly, Shang-Hua Teng, Jonah Sherman, and Richard Peng are all CMU graduates.

15.3.1 Electrical Flows

Given a connected *undirected* graph with general edge-capacities, we can view it as an electrical circuit, where each edge e of the original graph represents a resistor with resistance $r_e = 1/u_e$, and we connect (say, a 1-volt) battery between s to t. This causes electrical current to flow from s (the node with higher potential) to t. Recall the following laws about electrical flows.

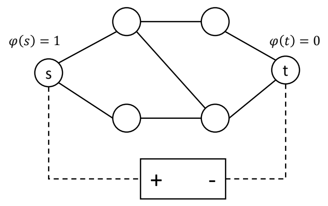


Figure 15.1: The currents on the wires would produce an electric flow (where all the wires within the graph have resistance 1).

Theorem 15.2 (Kirchoff's Voltage Law). *The directed potential changes along any cycle sum to* 0.

This means we can assign each node v a potential ϕ_v. Now the actual amount of current on any edge is given by Ohm's law, and is related to the potential drop across the edge.

Theorem 15.3 (Ohm's Law). *The electrical flow f_{uv} on the edge $e = uv$ is the ratio between the difference in potential ϕ (or voltage) between u, v and the resistance r_e of the edge:*

$$f_{uv} = \frac{\phi_u - \phi_v}{r_{uv}}.$$

Finally, we have flow conservation, much like in traditional network flows:

Theorem 15.4 (Kirchoff's Current Law). *If we set s and t to some voltages, the electrical current ensures flow-conservation at all nodes except s, t: the total current entering any non-terminal node equals the current leaving it.*

These laws give us a set of linear constraints that allow us to go between the voltages and currents. In order to show this, we define the ***Laplacian matrix*** of a graph.

15.3.2 *The Laplacian Matrix*

Given an undirected graph on n nodes and m edges, with non-negative *conductances* c_{uv} for each edge $e = uv$, we define the Laplacian matrix to be a $n \times n$ matrix L_G, with entries

The conductance of an edge is the reciprocal of the resistance of the edge: $c_e = 1/r_e$.

$$(L_G)_{uv} = \begin{cases} \sum_{w:uw \in E} c_{uw} & \text{if } u = v \\ -c_{uv} & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases} .$$

For example, if we take the 6-node graph in Figure 15.1 and assume that all edges have unit conductance, then its Laplacian L_G matrix is:

$$L_G = \begin{array}{c} \\ s \\ t \\ u \\ v \\ w \\ x \end{array} \begin{array}{c} \begin{array}{cccccc} s & t & u & v & w & x \end{array} \\ \begin{pmatrix} 2 & 0 & -1 & -1 & 0 & 0 \\ 0 & 2 & 0 & 0 & -1 & -1 \\ -1 & 0 & 3 & 0 & -1 & -1 \\ -1 & 0 & 0 & 2 & 0 & -1 \\ 0 & -1 & -1 & 0 & 2 & 0 \\ 0 & -1 & -1 & -1 & 0 & 3 \end{pmatrix} \end{array} .$$

Note that L_G is not a full-rank matrix since, e.g., the columns sum to zero. However, if the graph G is corrected, then the vector $\mathbb{1}$ is the only vector in the kernel of L_G, so its rank is $n - 1$. (proof?)

Equivalently, we can define the Laplacian matrix L^{uv} for the graph consisting of a single edge uv as

$$L^{uv} := c_{uv}\,(e_u - e_v)^\intercal (e_u - e_v).$$

Now for a general graph G, we define the Laplacian to be:

This Laplacian for the single edge uv has 1s on the diagonal at locations $(u,u), (v,v)$, and -1s at locations $(u,v), (v,u)$. Draw figure.

$$L_G = \sum_{uv \in E} L^{uv}.$$

In other words, L_G is the sum of little 'per-edge' Laplacians L^{uv}. (Since each of those Laplacians is clearly positive semidefinite (PSD), it follows that L_G is PSD too.)

A symmetric matrix $A \in \mathbb{R}^{n \times n}$ is called PSD if $x^\intercal A x \geq 0$ for all $x \in \mathbb{R}^n$, or equivalently, if all its eigenvalues are non-negative.

For yet another definition for the Laplacian, first consider the edge-vertex incidence matrix $B \in \{-1, 0, 1\}^{m \times n}$, where the rows are indexed by edges and the columns by vertices. The row corresponding to edge $e = uv$ has zeros in all columns other than u, v, it has an entry $+1$ in one of those columns (say u) and an entry -1 in the other (say v).

$$B = \begin{array}{c} \\ s \\ t \\ u \\ v \\ w \\ x \end{array} \begin{array}{c} \begin{array}{ccccccc} su & sv & uw & ux & vx & wt & xt \end{array} \\ \left(\begin{array}{ccccccc} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 \\ -1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 0 & 1 \end{array}\right) \end{array}.$$

The Laplacian matrix is now defined as $L_G := B^\intercal C B$, where $C \in R^{m\times m}$ is a diagonal matrix with entry C_{uv} containing the conductance for edge uv. E.g., for the example above, here's the edge-vertex incidence matrix, and since all conductances are 1, we have $L_G = B^\intercal B$.

15.3.3 *Solving for Electrical Flows: $Lx = b$*

Given the Laplacian matrix for the electrical network, we can figure out how the current flows by solving a linear system, i.e., a system of linear equations. Indeed, by Theorem 15.4, all the current flows from s to t. Suppose k units of current flows from s to t. By Theorem 15.3, the net current flow into a node v is precisely

$$\sum_{u:uv\in E} f_{uv} = \sum_{u:uv\in E} \frac{\phi_u - \phi_v}{r_{uv}}.$$

A little algebra shows this to be the v^{th} entry of the vector $L\phi$. Finally, by 15.4, this net current into v must be zero, unless v is either s or t, in which case it is either $-k$ or k respectively. Summarizing, if ϕ are the voltages at the nodes, they satisfy the linear system:

$$L\phi = k(e_s - e_t).$$

(Recall that k is the amount of current flowing from s to t, and e_s, e_t are elementary basis vectors.) It turns out the solutions ϕ to this linear system are unique up to translation, as long as the graph is connected: if ϕ is a solution, then $\{\phi + a \mid a \in \mathbb{R}\}$ is the set of all solutions.

Great: we have $n+1$ unknowns so far: the potentials at all the nodes, and the current value k. The above discussion gives us potentials at all the nodes in terms of the current value k. Now we can set unit potential at s, and ground t (i.e., set its potential to zero), and solve the linear system (with $n-1$ linearly independent constraints) for the remaining $n-1$ variables. The resulting value of k gives us the s-t current flow. Moreover, the potential settings at all the other nodes can now be read off from the ϕ vector. Then we can use Ohm's law to also read off the current on each edge, if we want.

How do we solve the linear system $L\phi = b$ (subject to these boundary conditions)? We can use Gaussian elimination, of course, but currently takes n^ω time in the worst-case. Thankfully, there are faster (approximate) methods, which we discuss in §15.3.5.

15.3.4 *Electrical Flows Minimize Energy Burn*

Here's another useful way of characterizing this current flow of k units from s and t: *the current flow is one minimizing the total energy dissipated*. Indeed, for a flow f, the ***energy burn*** on edge e is given by $(f_{uv})^2 r_{uv} = \frac{(\phi_u - \phi_v)^2}{r_{uv}}$, and the total energy burn is

$$\mathcal{E}(f) := \sum_{e \in E} f_e^2 r_e = \sum_{(u,v) \in E} \frac{(\phi_u - \phi_v)^2}{r_{uv}} = \phi^\intercal L \phi.$$

The electrical flow f produced happens to be

$$\arg\min_{f \text{ is an } s\text{-}t \text{ flow of value } k} \{\mathcal{E}(f)\}.$$

We often use this characterization when arguing about electrical flows.

15.3.5 *Solving Linear Systems*

We can solve a linear system $Lx = b$ fast? If L is a Laplacian matrix and we are fine with approximate solutions, we can do things much faster than Gaussian elimination. A line of work starting with Dan Spielman and Shang-Hua Teng, and then refined by Ioannis Koutis, Gary Miller, and Richard Peng shows how to (approximately) solve a Laplacian linear system in the time essentially near-linear in the number of non-zeros of the matrix L.

Spielman and Teng (200?)

Koutis, Miller, and Peng (2010)

Theorem 15.5 (Laplacian Solver). *There exists an algorithm that given a linear system $Lx = b$ with L being a Laplacian matrix (and having solution $\bar{x}$), find a vector $\hat{x}$ such that the error vector $z := L\hat{x} - b$ satisfies*

$$z^\intercal L z \leq \varepsilon(\bar{x}^\intercal L \bar{x}).$$

The algorithm is randomized? and runs in time $O(m \log^2 n \log 1/\varepsilon)$.

Given a positive semidefinite matrix A, the A-norm is defined as $\|x\|_A := \sqrt{x^\intercal A x}$. Hence the guarantee here says

$$\|L\hat{x} - b\|_L \leq \varepsilon \|\bar{x}\|_L.$$

Moreover, Theorem 15.5 can be converted to what we need; details appear in the Christiano et al. paper.

Corollary 15.6 (Laplacian Solver II). *There is an algorithm given a linear system $Lx = b$ corresponding to an electrical system as above, outputs an electrical flow f that satisfies*

$$\mathcal{E}(f) \leq (1 + \delta)\mathcal{E}(\widetilde{f}),$$

where $\widetilde{f}$ is the min-energy flow. The algorithm runs in $\widetilde{O}(\frac{m \log R}{\delta})$ time, where R is the ratio between the largest and smallest resistances in the network.

For the rest of this lecture we assume we can compute the corresponding minimum-energy flow *exactly* in time $\widetilde{O}(m)$. The arguments can easily be extended to incorporate the errors.

15.4 An $\widetilde{O}(m^{3/2})$-time Algorithm

Recall the setup from §15.2: given the polytope

$$K = \{f \mid \sum_{P \in \mathcal{P}} f_P = F, f \geq 0\},$$

and some edge weights p_e, we wanted a vector in K that satisfies

$$\sum_e p_e f_e \leq 1. \tag{15.4}$$

where $f_e := \sum_{P:e \in P} f_P$. Previously, we set $f_{P^*} = F$ for P^* being the shortest s-t path according to edge weights p_e, but that resulted in the *width*—the maximum capacity violation—being too as large as $\Omega(F)$. So we want to spread the flow over more paths.

Our solution will now be to have the oracle return a flow with width $O(\sqrt{m/\varepsilon})$, and which satisfies the following weaker version of the length bound (15.4) above:

$$\sum_{e \in E} p_e f_e \leq (1+\varepsilon) \sum_{e \in E} p_e + \varepsilon = 1 + 2\varepsilon.$$

It is a simple exercise to check that this weaker oracle changes the analysis of Theorem 14.4 only slightly, still showing that the multiplicative-weights-based process finds an s-t-flow of value F, but now the edge-capacities are violated by $1 + O(\varepsilon)$ instead of just $1 + \varepsilon$.

Indeed, we replace the shortest-path implementation of the oracle by the following electrical-flow implementation: we construct a *weighted* electrical network, where the resistance for each edge e is defined to be

$$r_e^t := p_e^t + \frac{\varepsilon}{m}.$$

This idea of setting the edge length to be p_e *plus a small constant term* is a general technique useful in controlling the width in other settings, as we will see in a HW problem.

We now compute currents f_e^t by solving the linear system $L^t \phi = F(e_s - e_t)$ and return the resulting flow. It remains to show that this flow spreads its mass around, and yet achieves a small "length" on average.

Theorem 15.7. *If f^* is a flow with value F and f is the minimum-energy flow returned by the oracle, then*

1. *(length)* $\sum_{e \in E} p_e f_e \leq (1+\varepsilon) \sum_{e \in E} p_e$,

2. *(width)* $\max_e f_e \le O(\sqrt{m/\varepsilon})$.

Proof. Since the flow f^* satisfies all the constraints, it burns energy

$$\mathcal{E}(f^*) = \sum_e (f_e^*)^2 r_e \le \sum_e r_e = \sum_e (p_e + \frac{\varepsilon}{m}) = 1 + \varepsilon.$$

Here we use that $\sum_e p_e = 1$. But since f is the flow K that minimizes the energy,

$$\mathcal{E}(f) \le \mathcal{E}(f^*) \le 1 + \varepsilon.$$

Now, using Cauchy-Schwarz,

$$\sum_e r_e f_e = \sum_e (\sqrt{r_e} f_e \cdot \sqrt{r_e}) \le \sqrt{(\sum_e r_e f_e^2)(\sum_e r_e)} \le \sqrt{1+\varepsilon}\sqrt{1+\varepsilon} = 1 + \varepsilon.$$

This proves the first part of the theorem. For the second part, we may use the bound on energy burnt to obtain

$$\sum_e f_e^2 \frac{\varepsilon}{m} \le \sum_e f_e^2 \left(p_e + \frac{\varepsilon}{m}\right) = \sum_e f_e^2 r_e = \mathcal{E}(f) \le 1 + \varepsilon.$$

Since each term in the leftmost summation is non-negative,

$$f_e^2 \frac{\varepsilon}{m} \le 1 + \varepsilon \implies f_e \le \sqrt{\frac{m(1+\varepsilon)}{\varepsilon}} \le \sqrt{\frac{2m}{\varepsilon}}$$

for each edge e. □

Using this oracle within the MW framework means the width is $\rho = O(\sqrt{m})$, and each of the $O\big(\frac{\rho \log m}{\varepsilon^2}\big)$ iterations takes $\widetilde{O}(m)$ time by Corollary 15.6, giving a runtime of $\widetilde{O}(m^{3/2})$.

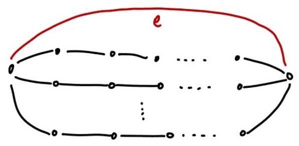


Figure 15.2: There are $k = \Theta(\sqrt{m})$ black paths of length k each. All edges have unit capacities.

In fact, this bound on the width is tight: consider the example network on the right. The effective resistance of the entire collection of black edges is 1, which matches the effective resistance of the red edge, so half the current goes on the top red edge. If we set $F = k + 1$ (which is the max-flow), this means a current of $\Theta(\sqrt{m})$ goes on the top edge.

Sadly, while the idea of using electrical flows is very cool, the runtime of $O(m^{3/2})$ is not that impressive. The algorithms of Karzanov, and of Even and Tarjan, for exact flow on *directed* unit-capacity graphs in time $O(m \min(m^{1/2}, n^{2/3}))$ were known even back in the 1970s. (Algorithms with similar runtime are known for capacitated cases too.) Thankfully, this is not the end of the story: we can take the idea of electrical flows further to get a better algorithm, as we show in the next section.

15.5 *Optional: An $\widetilde{O}(m^{4/3})$-time Algorithm*

The idea to get an improved bound on the width is to use a crude but effective trick: if we have an edge with electrical flow of more than

$\rho \approx m^{1/3}$ in some iteration, we delete it for that iteration (and for the rest of the process), and find a new flow. Clearly, no edge now carries a flow more than ρ. The main thrust of the proof is to show that we do not end up butchering the graph, and that the maximum flow value reduces by only a small amount due to these edge deletions. Formally, we set:

$$\rho = \frac{m^{1/3} \log m}{\varepsilon}. \tag{15.5}$$

and show that at most εF edges are ever deleted by the process. The crucial ingredient in this proof is this observation: every time we delete an edge, the effective resistance between s and t increases by a lot.

We assume that a flow value of F is feasible; moreover, $F \geq \rho$, else Ford-Fulkerson can be implemented in time $O(mF) \leq \widetilde{O}(m^{4/3})$.

Since we need to argue about how many edges are deleted in the entire algortihm (and not just in one call to the oracle), we explicitly maintain edge-weights w_e^t, instead of using the results from the previous sections as a black-box.

15.5.1 The Effective Resistance

Loosely speaking, the ***effective resistance*** between nodes u and v is the resistance offered by the network to electrical flows between u and v. There are many ways of formalizing this: the most useful one in this context is the following.

Definition 15.8 (Effective Resistance)**.** The effective resistance between s and t, denoted by $R_{\text{eff}}(st)$, is the energy burned if we send one unit of electrical current from s to t.

Since we only consider the effective resistance between s and t in this lecture, we simply write R_{eff}. The following results relate the effective resistances before and after we change the resistances of some edges.

Lemma 15.9. *Consider an electrical network with edge resistances r_e.*

1. *(Rayleigh Monotonicity) If we increase the resistances to $r'_e \geq r_e$ for all e, the resulting effective resistance is*

$$R'_{\text{eff}} \geq R_{\text{eff}}.$$

2. *Suppose f is an s-t electrical flow, suppose e is an edge with energy burn $f_e^2 r_e \geq \beta \mathcal{E}(f)$. If we set $r'_e \leftarrow \infty$, then the new effective resistance*

$$R'_{\text{eff}} \geq \left(\frac{R_{\text{eff}}}{1-\beta}\right).$$

Proof. Recall that if we send electrical flow from s to t, the resulting flow f minimizes the total energy burned $\mathcal{E}(f) = \sum_e f_e^2 r_e$. To prove the first statement: for each flow, the energy burned with the new resistances is at least that with the old resistances. Need to add in second part. □

15.5.2 *A Modified Algorithm*

Let's give our algorithm that explicitly maintains the edge weights: We start off with weights $w_e^1 = 1$ for all $e \in E$. At step t of the algorithm:

1. Find the min-energy flow f^t of value F in the remaining graph with respect to edge resistances $r_e^t := w_e^t + \frac{\varepsilon}{m} W^t$.

2. If there is an edge e with $f_e^t > \rho$, delete e (for the rest of the algorithm), and go back to Item 1.

3. Update the edge weights $w_e^{t+1} \leftarrow w_e^t(1 + \frac{\varepsilon}{\rho} f_e^t)$. This division by ρ accounts for the edge-capacity violations being as large as ρ.

Stop after $T := \frac{\rho \log m}{\varepsilon^2}$ iterations, and output $\widehat{f} = \frac{1}{T}\sum_t f^t$.

15.5.3 *The Analysis*

Let us first comment on the runtime: each time we find an electrical flow, we either delete an edge, or we push flow and increment t. The latter happens for T steps by construction; the next lemma shows that we only delete edges in a few iterations.

Lemma 15.10. *We delete at most $m^{1/3} \leq \varepsilon F$ edges over the run of the algorithm.*

We defer the proof to later, and observe that the total number of electrical flows computed is therefore $O(T)$. Each such computation takes $\widetilde{O}(m/\varepsilon)$ by Corollary 15.6, so the overall runtime of our algorithm is $O(m^{4/3}/\operatorname{poly}(\varepsilon))$.

Next, we show that the flow $\widehat{f}$ is an $(1 + O(\varepsilon)$-approximate maximum s-t flow. We start with an analog of Theorem 15.7 that accounts for edge deletions.

Lemma 15.11. *Suppose $\varepsilon \leq 1/10$. If we delete at most εF edges from G:*

1. *the flow f^t at step t burns energy $\mathcal{E}(f^t) \leq (1 + 3\varepsilon)W^t$,*

2. *$\sum_e w_e^t f_e^t \leq (1 + 3\varepsilon)W^t \leq 2W^t$, and*

3. *if $\widehat{f} \in K$ is the flow eventually returned, then $\widehat{f}_e \leq (1 + O(\varepsilon))$.*

Proof. We assumed there exists a flow f^* of value F that respects all capacities. Deleting εF edges can only hit εF of these flow paths, so there exists a capacity-respecting flow of value at least $(1-\varepsilon)F$. Scaling up by $\frac{1}{(1-\varepsilon)}$, there exists a flow f' of value F using each edge to extent $\frac{1}{(1-\varepsilon)}$. The energy of this flow according to resistances r_e^t is at most

$$\mathcal{E}(f') = \sum_e r_e^t (f'_e)^2 \leq \frac{1}{(1-\varepsilon)^2} \sum_e r_e^t \leq \frac{W^t}{(1-\varepsilon)^2} \leq (1+3\varepsilon)W^t,$$

for ε small enough. Since we find the minimum energy flow, $\mathcal{E}(f^t) \leq \mathcal{E}(f') \leq W^t(1+3\varepsilon)$. For the second part, we again use the Cauchy-Schwarz inequality:

$$\sum_e w_e^t f_e^t \leq \sqrt{\sum_e w_e^t} \sqrt{\sum_e w_e^t (f_e^t)^2} \leq \sqrt{W^t \cdot W^t(1+3\varepsilon)} \leq (1+3\varepsilon)W^t \leq 2W^t.$$

The last step is very loose, but it will suffice for our purposes.

To calculate the congestion of the final flow, observe that even though the algorithm above explicitly maintains weights, we can just appeal directly to the guarantees . Indeed, define $p_e^t := \frac{w_e^t}{W^t}$ for each time t; the previous part implies that the flow f^t satisfies

$$\sum_e p_e^t f_e^t \leq 1 + 3\varepsilon$$

for precisely the p^t values that the Hedge-based LP solver would return if we gave it the flows $f^0, f^1, \ldots, f^{t-1}$. Using the guarantees of that LP solver, the average flow $\widehat{f}$ uses any edge e to at most $(1+3\varepsilon)+\varepsilon$. □

Finally, it remains to prove Lemma 15.10.

Proof of Lemma 15.10. We track two quantities: the total weight W^t and the s-t-effective resistance R_{eff}. First, the weight starts at $W^0 = m$, and when we do an update,

$$\begin{aligned} W^{t+1} &= \sum_e w_e^t \left(1 + \frac{\varepsilon}{\rho} f_e^t\right) = W_t + \frac{\varepsilon}{\rho} \sum_e w_e^t f_e^t \\ &\leq W^t + \frac{\varepsilon}{\rho}(2W^t) \qquad \text{(From Claim 15.11)} \end{aligned}$$

Hence we get that for $T = \frac{\rho \ln m}{\varepsilon^2}$,

$$W^T \leq W^0 \cdot \left(1 + \frac{2\varepsilon}{\rho}\right)^T \leq m \cdot \exp\left(\frac{2\varepsilon \cdot T}{\rho}\right) = m \cdot \exp\left(\frac{2 \ln m}{\varepsilon}\right).$$

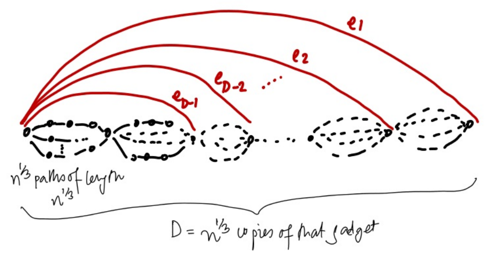
Therefore, the total weight is at most $m^{1+2/\varepsilon}$. Next, we consider the s-t-effective resistance R_{eff}.

1. At the beginning, all edges have resistance $1 + \varepsilon$. When we send F flow, some edge has at least F/m flow on it, so the energy burn is at least $(F/m)^2$. This means R_{eff} at the beginning is at least $(F/m)^2 \geq 1/m^2$.
2. The weights increase each time we do an update, so R_{eff} does not decrease. (This is one place it is more convenience to argue about weights w_e^t explicitly, and not just the probabilities p_e^t.)
3. Each deleted edge e has flow at least ρ, and hence energy burn at least $(\rho^2)\, w_e^t \geq (\rho^2)\, \frac{\varepsilon}{m}\, W^t$. Since the total energy burn is at most $2W^t$ from Lemma 15.11, the deleted edge e was burning at least $\beta := \frac{\rho^2\varepsilon}{2m}$ fraction of the total energy. Hence
$$R_{\text{eff}}^{new} \geq \frac{R_{\text{eff}}^{old}}{(1 - \frac{\rho^2\varepsilon}{2m})} \geq R_{\text{eff}}^{old} \cdot \exp\left(\frac{\rho^2\varepsilon}{2m}\right)$$
if we use $\frac{1}{1-x} \geq e^{x/2}$ when $x \in [0, 1/4]$.
4. For the final effective resistance, note that we send F flow with total energy burn $2W^T$; since the energy depends on the square of the flow, we have $R_{\text{eff}}^{final} \leq \frac{2W^T}{F^2} \leq 2W^T$.

(All these calculations hold as long as we have not deleted more than εF edges.) Now, to show that this invariant is maintained, suppose D edges are deleted over the course of the T steps. Then
$$R_{\text{eff}}^0 \exp\left(D \cdot \frac{\rho^2\varepsilon}{2m}\right) \leq R_{\text{eff}}^{final} \leq 2W^T \leq 2m \cdot \exp\left(\frac{2\ln m}{\varepsilon}\right).$$
Taking logs and simplifying, we get that
$$\begin{aligned} \frac{\varepsilon\rho^2 D}{2m} &\leq \ln(2m^3) + \frac{2\ln m}{\varepsilon} \\ \implies D &\leq \frac{2m}{\varepsilon\rho^2}\left(\frac{(\ln m)(1 + O(\varepsilon))}{\varepsilon}\right) \ll m^{1/3} \leq \varepsilon F. \end{aligned}$$
This bounds the number of deleted edges D as desired. $\square$

15.5.4 *Tightness of the Analysis*

This analysis of the algorithm is tight. Indeed, the algorithm needs $\Omega(m^{1/3})$ iterations, and deletes $\Omega(m^{1/3})$ edges for the example on the right. In this example, $m = \Theta(n)$. Each black gadget has a unit effective resistance, and if we do the calculations, the effective resistance between s and t tends to the golden ratio. If we set $F = n^{1/3}$ (which is almost the max-flow), a constant fraction of the current (about $\Theta(n^{1/3})$) uses the edge e_1. Once that edge is deleted, the next red edge e_2 carries a lot of current, etc., until all red edges get deleted.


e_1
e_2
e_{D-2}
e_{D-1}
$n^{1/3}$ paths of length $n^{1/3}$
$D = n^{1/3}$ copies of that gadget
Figure 15.3: Again, all edges have unit capacities.

15.5.5 Subsequent Work

A couple years after this work, Sherman, and independently, Kelner et al. gave $O(m^{1+o(1)}/\varepsilon^{O(1)})$-time algorithms for approximate max-flow problem on undirected graphs. This was improved, using some more ideas, to a runtime of $O(m \operatorname{poly}\log m/\varepsilon^{O(1)})$-time by Richard Peng. These are based on the ideas of ***oblivious routings***, and ***non-Euclidean gradient descent***, and we hope to cover this in an upcoming lecture.

Sherman (2013)

Kelner, Lee, Oracchia, Sidford (2013)

Peng (2014)

There has also been work on faster directed flows: work by Madry, and thereafter by more refs here, have improved the current best result for max-flow in unweighted directed graphs to $\widetilde{O}(m^{4/3})$, matching the above result.

16
The Gradient Descent Framework

Consider the problem of finding the minimum-energy s-t electrical unit flow: we wanted to minimize the total energy burn

$$\mathcal{E}(f) = \sum_e f_e^2 r_e$$

for flow values f that represent a unit flow from s to t (these form a polytope). We alluded to algorithms that solve this problem, but one can also observe that $\mathcal{E}(f)$ is a convex function, and we want to find a minimizer within some polytope K. Equivalently, we wanted to solve the linear system

$$L\phi = (e_s - e_t),$$

which can be cast as finding a minimizer of the convex function

$$\|L\phi - (e_s - e_t)\|^2.$$

How can we minimize these functions efficiently? In this lecture, we will study the gradient descent framework for the general problem of minimizing functions, and give concrete performance guarantees for the case of convex optimization.

16.1 Convex Sets and Functions

First, recall the following definitions:

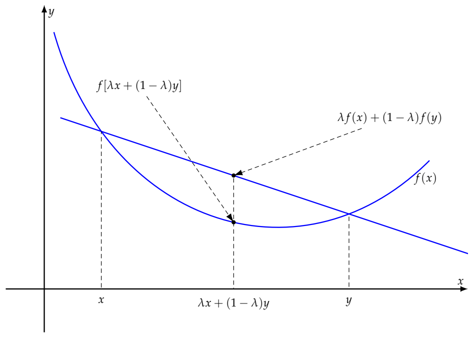
Definition 16.1 (Convex Set)**.** A set $K \subseteq \mathbb{R}^n$ is called ***convex*** if for all $x, y \in K$,

$$\lambda x + (1-\lambda)y \in K, \tag{16.1}$$

for all values of $\lambda \in [0,1]$. Geometrically, this means that for any two points in K, the line connecting them is contained in K.

Definition 16.2 (Convex Function)**.** A function $f : K \to \mathbb{R}$ defined on a convex set K is called ***convex*** if for all $x, y \in K$,

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y), \tag{16.2}$$


y
$f[\lambda x + (1-\lambda)y]$
$\lambda f(x) + (1-\lambda) f(y)$
$f(x)$
x
x
$\lambda x + (1-\lambda)y$
y

for all values of $\lambda \in [0, 1]$.

There are two kinds of problems that we will study. The most basic question is that of ***unconstrained convex minimization*** (UCM): given a convex function f, we want to find

$$\min_{x \in \mathbb{R}^n} f(x).$$

In some cases we will be concerned with the constrained convex minimization (CCM) problem: given a convex function f and a convex set K, we want to find

$$\min_{x \in K} f(x).$$

Note that setting $K = \mathbb{R}^n$ gives us the unconstrained case.

16.1.1 Gradient

For most of the following discussion, we assume that the function f is differentiable. In that case, we can give an equivalent characterization, based on the notion of the ***gradient*** $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$.

The *directional derivative* of f at x (in the direction y) is defined as

$$f'(x; y) := \lim_{\varepsilon \to 0} \frac{f(x + \varepsilon y) - f(x)}{\varepsilon}.$$

If there exists a vector g such that $\langle g, y \rangle = f'(x; y)$ for all y, then f is called *differentiable* at x, and g is called the *gradient*. It follows that the gradient must be of the form

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \cdots, \frac{\partial f}{\partial x_n}(x) \right).$$

Fact 16.3 (First-order condition). A function $f : K \to \mathbb{R}$ is convex if and only if

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle, \tag{16.3}$$

for all $x, y \in K$.

Geometrically, Fact 16.3 states that the function always lies above its tangent plane, for all points in K. If the function f is twice-differentiable, and if $H_f(x)$ is its ***Hessian matrix***, i.e. its matrix of second derivatives at $x \in K$:

$$(H_f)_{i,j}(x) := \frac{\partial^2 f}{\partial x_i \, \partial x_j}(x), \tag{16.4}$$

then we get yet another characterization of convex functions.

Fact 16.4 (Second-order condition). A twice-differentiable function f is convex if and only if $H_f(x)$ is positive semidefinite for all $x \in K$.

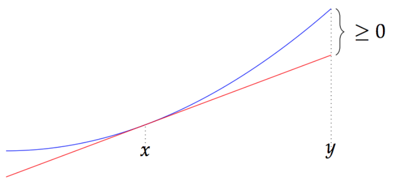


Figure 16.1: The blue line denotes the function and the red line is the tangent line at x. (Figure from Nisheeth Vishnoi.)

16.1.2 Lipschitz Functions

We will need a notion of "niceness" for functions:

Definition 16.5 (Lipschitz continuity)**.** For a convex set $K \subseteq \mathbb{R}^n$, a function $f : K \to \mathbb{R}$ is called *G-Lipschitz* (or *G-Lipschitz continuous*) with respect to the norm $\|\cdot\|$ if

$$|f(x) - f(y)| \leq G \, \|x - y\|,$$

for all $x, y \in K$.

In this chapter we focus on the Euclidean or ℓ_2-norm, denoted by $\|\cdot\|_2$. General norms arise in the next chapter, when we talk about mirror descent. Again, assuming that the function is differentiable allows us to give an alternative characterization of Lipschitzness.

Fact 16.6. A differentiable function $f : K \to \mathbb{R}^n$ is G-Lipschitz with respect to $\|\cdot\|_2$ if and only if

$$\|\nabla f(x)\|_2 \leq G, \tag{16.5}$$

for all $x \in K$.

16.2 *Unconstrained Convex Minimization*

If the function f is convex, any *stationary point* (i.e., a point x^* where $\nabla f(x^*) = 0$) is also a *global minimum*: just use Fact 16.3 to infer that $f(y) \geq f(x^*)$ for all y. Now given a convex function, we can just solve the equation

$$\nabla f(x) = 0$$

to compute the global minima exactly. This is often easier said than done: for instance, if the function f we want to minimize may not be given explicitly. Instead we may only have a gradient oracle that given x, returns $\nabla f(x)$.

Even when f is explicit, it may be expensive to solve the equation $\nabla f(x) = 0$, and gradient descent may be a faster way. One example arises when solving linear systems: given a quadratic function $f(x) = \frac{1}{2}x^\intercal Ax - bx$ for a symmetric matrix A (say having full rank), a simple calculation shows that

$$\nabla f(x) = 0 \iff Ax = b \iff x = A^{-1}b.$$

This can be solved in $O(n^\omega)$ (i.e., matrix-multiplication) time using Gaussian elimination—but for "nice" matrices A we are often able to approximate a solution much faster using the gradient-based methods we will soon see.

16.2.1 *The Basic Gradient Descent Method*

Gradient descent is an iterative algorithm to approximate the optimal solution x^*. The main idea is simple: since the gradient tells us the direction of steepest increase, we'd like to move opposite to the direction of the gradient to decrease the fastest. So by selecting an initial position x_0 and a step size η_t at each time t, we can repeatedly perform the update:

$$x_{t+1} \leftarrow x_t - \eta_t \cdot \nabla f(x_t). \tag{16.6}$$

There are many choices to be made: where should we start? What are the step sizes? When do we stop? While each of these decisions depend on the properties of the particular instance at hand, we can show fairly general results for general convex functions.

16.2.2 *An Algorithm for General Convex Functions*

The algorithm fixes a step size for all times t, performs the update (16.6) for some number of steps T, and then returns the average of all the points seen during the process.

Algorithm 13: Gradient Descent

```
13.1 $x_1 \leftarrow$ starting point
13.2 for $t \leftarrow 1$ to $T$ do
13.3     $x_{t+1} \leftarrow x_t - \eta \cdot \nabla f(x_t)$
13.4 return $\widehat{x} := \frac{1}{T}\sum_{t=1}^{T} x_i$.
```

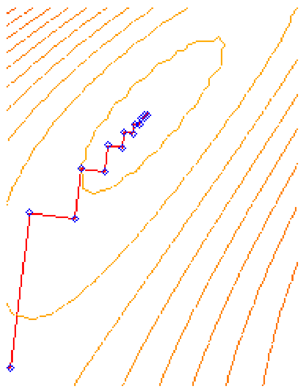


Figure 16.2: The yellow lines denote the level sets of the function f and the red walk denotes the steps of gradient descent. (Figure from Wikipedia.)

This is easy to visualize in two dimensions: draw the level sets of the function f, and the gradient at a point is a scaled version of normal to the tangent line at that point. Now the algorithm's path is often a zig-zagging walk towards the optimum (see Fig 16.2).

Interestingly, we can give rigorous bounds on the convergence of this algorithm to the optimum, based on the distance of the starting point from the optimum, and bounds on the Lipschitzness of the function. If both these are assumed to be constant, then our error is smaller than ε in only $O(1/\varepsilon^2)$ steps.

Proposition 16.7. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be convex, differentiable and G-Lipschitz. Let x^* be any point in $\mathbb{R}^d$. If we define $T := \frac{G^2\|x_0-x^*\|^2}{\varepsilon^2}$ and $\eta := \frac{\|x_0-x^*\|}{G\sqrt{T}}$, then the solution $\widehat{x}$ returned by gradient descent satisfies*

$$f(\widehat{x}) \le f(x^*) + \varepsilon. \tag{16.7}$$

In particular, this holds when x^* is a minimizer of f.

The core of this proposition lies in the following theorem

Theorem 16.8. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be convex, differentiable and G-Lipschitz. Then the gradient descent algorithm ensures that*

$$\sum_{t=1}^{T} f(x_t) \le \sum_{t=1}^{T} f(x^*) + \frac{1}{2}\eta T G^2 + \frac{1}{2\eta}\|x_0 - x^*\|^2. \tag{16.8}$$

We will prove Theorem 16.8 in the next section, but let's first use it to prove Proposition 16.7.

Proof of Proposition 16.7. By definition of $\widehat{x}$ and the convexity of f,

$$f(\widehat{x}) = f\Big(\frac{1}{T}\sum_{t=1}^{T} x_t\Big) \le \frac{1}{T}\sum_{t=1}^{T} f(x_t).$$

By Theorem 16.8,

$$\frac{1}{T}\sum_{t=1}^{T} f(x_t) \leq f(x^*) + \underbrace{\frac{1}{2}\eta G^2 + \frac{1}{2\eta T}\|x_0 - x^*\|^2}_{\text{error}}.$$

The error terms balance when $\eta = \frac{\|x_0 - x^*\|}{G\sqrt{T}}$, giving

$$f(\widehat{x}) \leq f(x^*) + \frac{\|x_0 - x^*\|G}{\sqrt{T}}.$$

Finally, we set $T = \frac{1}{\varepsilon^2}G^2\|x_0 - x^*\|^2$ to obtain

$$f(\widehat{x}) \leq f(x^*) + \varepsilon. \qquad \square$$

Observe: we do not (and cannot) show that the point $\widehat{x}$ is close in distance to x^*; we just show that the function value $f(\widehat{x}) \approx f(x^*)$. Indeed, if the function is very flat close to x^* and we start off at some remote point, we make tiny steps as we get close to x^*, and we cannot hope to get close to it.

The $1/\varepsilon^2$ dependence of the number of oracle calls was shown to be tight for gradient-based methods by Yurii Nesterov, if we allow f to be any G-Lipschitz function. However, if we assume that the function is "well-behaved", we can indeed improve on the $1/\varepsilon^2$ dependence. Moreover, if the function is strongly convex, we can show that x^* and $\widehat{x}$ are close to each other as well: see §16.5 for such results.

The convergence guarantee in Proposition 16.7 is for the time-averaged point $\widehat{x}$. Indeed, using a fixed step size means that our iterates may get stuck in a situation where $x_{t+2} = x_t$ after some point and hence we never improve, even though $\widehat{x}$ is at the minimizer. One can also show that $f(x_T) \leq f(x^*) + \varepsilon$ if we use a time-varying step size $\eta_t = O(1/\sqrt{t})$, and increase the time horizon slightly to $O(1/\varepsilon^2 \log 1/\varepsilon)$. We refer to the work of Shamir and Zhang.

16.2.3 *Proof of Theorem 16.8*

Like in the proof of the multiplicative weights algorithm, we will use a potential function. Define

$$\Phi_t := \frac{\|x_t - x^*\|^2}{2\eta}. \qquad (16.9)$$

We start the proof of Theorem 16.8 by understanding the one-step change in potential:

Lemma 16.9. $f(x_t) + (\Phi_{t+1} - \Phi_t) \leq f(x^*) + \frac{1}{2}\eta G^2$.

Proof. Using the identity

$$\|a+b\|^2 = \|a\|^2 + 2\langle a,b\rangle + \|b\|^2,$$

with $a+b = x_{t+1} - x^*$ and $a = x_t - x^*$, we get

$$\begin{aligned}\Phi_{t+1} - \Phi_t &= \frac{1}{2\eta}\left(\|x_{t+1} - x^*\|^2 - \|x_t - x^*\|^2\right) \qquad (16.10)\\ &= \frac{1}{2\eta}\Big(2\underbrace{\langle x_{t+1} - x_t, x_t - x^*\rangle}_{\langle b,a\rangle} + \underbrace{\|x_{t+1} - x_t\|^2}_{\|b\|^2}\Big);\end{aligned}$$

now using $x_{t+1} - x_t = -\eta\,\nabla f(x_t)$ from gradient descent,

$$= \frac{1}{2\eta}\left(2\langle -\eta\nabla f(x_t), x_t - x^*\rangle + \|\eta\nabla f(x_t)\|^2\right).$$

Since f is G-Lipschitz, $\|\nabla f(x)\| \le G$ for all x. Thus,

$$f(x_t) + (\Phi_{t+1} - \Phi_t) \le f(x_t) + \langle\nabla f(x_t), x^* - x_t\rangle + \frac{1}{2}\eta G^2$$

Since f is convex, we know that $f(x_t) + \langle\nabla f(x_t), x^* - x_t\rangle \le f(x^*)$. Thus, we conclude that

$$f(x_t) + (\Phi_{t+1} - \Phi_t) \le f(x^*) + \frac{1}{2}\eta G^2. \qquad \square$$

Now that we understand how our potential changes over time, proving the theorem is straightforward.

Proof of Theorem 16.8. We start with the inequality we proved above:

$$f(x_t) + (\Phi_{t+1} - \Phi_t) \le f(x^*) + \frac{1}{2}\eta G^2.$$

Summing over $t = 1, \dots, T$,

$$\sum_{t=1}^{T} f(x_t) + \sum_{t=1}^{T}(\Phi_{t+1} - \Phi_t) \le \sum_{t=1}^{T} f(x^*) + \frac{1}{2}\eta G^2 T$$

The sum of potentials on the left telescopes to give:

$$\sum_{t=1}^{T} f(x_t) + \Phi_{T+1} - \Phi_1 \le \sum_{t=1}^{T} f(x^*) + \frac{1}{2}\eta G^2 T$$

Since the potentials are nonnegative, we can drop the Φ_T term:

$$\sum_{t=1}^{T} f(x_t) - \Phi_1 \le \sum_{t=1}^{T} f(x^*) + \frac{1}{2}\eta G^2 T$$

Substituting in the definition of Φ_1 and moving it over to the right hand side completes the proof. $\square$

16.2.4 *Some Remarks on the Algorithm*

We assume a gradient oracle for the function: given a point x, it returns the gradient $\nabla f(x)$ at that point. If the function f is not given explicitly, we may have to estimate the gradient using, e.g., random sampling. One particularly sample-efficient solution is to pick a uniformly random point $u \sim \mathbb{S}^{n-1}$ from the sphere in $\mathbb{R}^n$, and return

$$d\left[\frac{f(x+\delta u)}{\delta}u\right]$$

As $\delta \to 0$, the expectation of this expression tends to $\nabla f(x)$, using Stokes' theorem.

for some tiny $\delta > 0$. It is slightly mysterious, so perhaps it is useful to consider its expectation in the case of a univariate function:

$$\mathbb{E}_{u\sim\{-1,+1\}}\left[\frac{f(x+\delta u)}{\delta}\,u\right] = \frac{f(x+\delta)-f(x-\delta)}{2\delta} \approx f'(x).$$

In general, randomized strategies form the basis of ***stochastic gradient descent***, where we use an unbiased estimator of the gradient, instead of computing the gradient itself (because it is slow to compute, or because enough information is not available). The challenge is now to control the variance of this estimator.

Another concern is that the step-size η and the number of steps T both require knowledge of the distance $\|x_1 - x^*\|$ as well as the bound on the gradient. More here. As an exercise, show that using the time-varying step-size $\eta_t := \frac{\|x_0-x^*\|}{G\sqrt{t}}$ also gives a very similar convergence rate.

Finally, the guarantee is for $f(\widehat{x})$, where $\widehat{x}$ is the time-average of the iterates. What about returning the final iterate? It turns out this has comparable guarantees, but the proof is slightly more involved. Add references.

16.3 *Constrained Convex Minimization*

Unlike the unconstrained case, the gradient at the minimizer may not be zero in the constrained case—it may be at the boundary. In this case, the condition for a convex function $f : K \to \mathbb{R}$ to be minimized at $x^* \in K$ is now

This is the analog of the minimizer of a single variable function being achieved either at a point where the derivative is zero, or at the boundary.

$$\langle \nabla f(x^*), y - x^*\rangle \geq 0 \qquad \text{for all } y \in K. \tag{16.11}$$

In other words, all vectors $y - x^*$ pointing within K are "positively correlated" with the gradient.

When x^* is in the interior of K, the condition (16.11) is equivalent to $\nabla f(x^*) = 0$.

16.3.1 *Projected Gradient Descent*

While the gradient descent algorithm still makes sense: moving in the direction opposite to the gradient still moves us towards lower function values. But we must change our algorithm to ensure that the new point x_{t+1} lies within K. To ensure this, we simply project the new iterate x_{t+1} back onto K. Let $\text{proj}_K : \mathbb{R}^n \to K$ be defined as

$$\text{proj}_K(y) = \arg\min_{x \in K} \|x - y\|_2 .$$

The modified algorithm is given below in Algorithm 14, with the changes highlighted in blue.

Algorithm 14: Projected Gradient Descent For CCM

```
14.1 $x_1 \leftarrow$ starting point
14.2 for $t \leftarrow 1$ to $T$ do
14.3     $x'_{t+1} \leftarrow x_t - \eta \cdot \nabla f(x_t)$
14.4     $x_{t+1} \leftarrow \text{proj}_K(x'_{t+1})$
14.5 return $\widehat{x} := \frac{1}{T}\sum_{t=1}^{T} x_t$
```

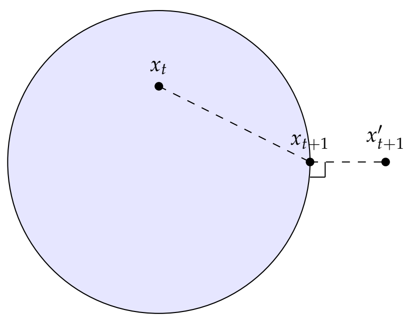


Figure 16.3: Projection onto a convex body

We will show below that a result almost identical to that of Theorem 16.8, and hence that of Proposition 16.7 holds.

Proposition 16.10. *Let K be a closed convex set, and $f : K \to \mathbb{R}$ be convex, differentiable and G-Lipschitz. Let $x^* \in K$, and define $T := \frac{G^2\|x_0 - x^*\|^2}{\varepsilon^2}$ and $\eta := \frac{\|x_0 - x^*\|}{G\sqrt{T}}$. Then the solution $\widehat{x}$ returned by projected gradient descent satisfies*

$$f(\widehat{x}) \leq f(x^*) + \varepsilon. \tag{16.12}$$

In particular, this holds when x^* is a minimizer of f.

Proof. We can reduce to an analogous constrained version of Theorem 16.8. Let us start the proof as before:

$$\Phi_{t+1} - \Phi_t = \frac{1}{2\eta}\left(\|x_{t+1} - x^*\|^2 - \|x_t - x^*\|^2\right) \tag{16.13}$$

But x_{t+1} is the projection of x'_{t+1} onto K, which is difficult to reason about. Also, we know that $-\eta \nabla f(x_t) = x'_{t+1} - x^*$, not $x_{t+1} - x^*$, so we would like to move to the point x'_{t+1}. Indeed, we claim that $\left\|x'_{t+1} - x^*\right\| \geq \|x_{t+1} - x^*\|$, and hence we get

$$\Phi_{t+1} - \Phi_t = \frac{1}{2\eta}\left(\|x'_{t+1} - x^*\|^2 - \|x_t - x^*\|^2\right). \tag{16.14}$$

Now the rest of the proof of Theorem 16.8 goes through unchanged.

Why is the claim $\left\|x'_{t+1} - x^*\right\| \geq \|x_{t+1} - x^*\|$ true? Since K is convex, projecting onto it *gets us closer to every point* in K, in particular to $x^* \in K$. To formally prove this fact about projections, consider the angle $x^* \to x_{t+1} \to x'_{t+1}$. This is a non-acute angle, since the orthogonal projection means K likes to one side of the hyperplane defined by the vector $x'_{t+1} - x_{t+1}$, as in the figure on the right. □

Note that restricting the play to K can be helpful in two ways: we can upper-bound the distance $\|x^* - x_1\|$ by the diameter of K, and moreover we need only consider the Lipschitzness of f for points within K.

16.4 *Online Gradient Descent, and Relationship with MW*

We considered gradient descent for the *offline* convex minimization problem, but one can use it even when the function changes over time. Indeed, consider the ***online convex optimization (OCO)*** problem: at each time step t, the algorithm proposes a point $x_t \in K$ and an adversary gives a function $f_t : K \to \mathbb{R}$ with $\|\nabla f_t\| \leq G$. The cost of each time step is $f_t(x_t)$ and your objective is to minimize

$$\text{regret } = \sum_t f_t(x_t) - \min_{x^* \in K} \sum_t f_t(x^*).$$

For instance if $K = \Delta_n$, and $f_t(x) := \langle \ell_t, x \rangle$ for some loss vector $\ell_t \in [-1, 1]^n$, then we are back in the experts setting of the previous chapters. Of course, the OCO problem is far more general, allowing arbitrary convex functions.

Surprisingly, we can use the almost same algorithm to solve the OCO problem, with one natural modification: the update rule is now taken with respect to gradient of the *current* function f_t:

This was first observed by Martin Zinkevich in 2002, when he was a Ph.D. student here at CMU.

$$x_{t+1} \leftarrow x_t - \eta \cdot \nabla f_t(x_t).$$

Looking back at the proof in §16.2, the proof of Lemma 16.9 immediately extends to give us

$$f_t(x_t) + \Phi_{t+1} - \Phi_t \leq f_t(x^*) + \frac{1}{2}\eta G^2.$$

Now summing this over all times t gives

$$\begin{aligned}\sum_{t=1}^{T} \big(f_t(x_t) - f_t(x^*)\big) &\leq \sum_{t=1}^{T} (\Phi_t - \Phi_{t+1}) + \frac{1}{2}\eta T G^2 \\ &\leq \Phi_1 + \frac{1}{2}\eta T G^2,\end{aligned}$$

since $\Phi_{T+1} \geq 0$. The proof is now unchanged: setting $T \geq \frac{\|x_1 - x^*\|^2 G^2}{\varepsilon^2}$ and $\eta = \frac{\|x_1 - x^*\|}{G\sqrt{T}}$, and doing some elementary algebra as above,

$$\frac{1}{T}\sum_{t=0}^{T} \big(f_t(x_t) - f_t(x^*)\big) \leq \frac{\|x_1 - x^*\| G}{\sqrt{T}} \leq \varepsilon.$$

16.4.1 *Comparison to the MW/Hedge Algorithms*

One advantage of the gradient descent approach (and analysis) over the multiplicative weight-based ones is that the guarantees here hold for all convex bodies K and all convex functions, as opposed to being just for the unit simplex Δ_n and linear losses $f_t(x) = \langle \ell_t, x \rangle$, say for $\ell_t \in [-1,1]^n$. However, in order to make a fair comparison, suppose we restrict ourselves to Δ_n and linear losses, and consider the number of rounds T before we get an average regret of ε.

- If we consider $\|x_1 - x^*\|$ (which, in the worst case, is the diameter of K), and G (which is an upper bound on $\|\nabla f_t(x)\|$ over points in K) as constants, then the $T = \Theta(\frac{1}{\varepsilon^2})$ dependence is the same.
- For a more quantitative comparison, note that $\|x_1 - x^*\| \leq \sqrt{2}$ for $x_1, x^* \in \Delta_n$, and $\|\nabla f_t(x)\| = \|\ell_t\| \leq \sqrt{n}$ for $\ell_t \in [-1,1]^n$. Hence, Proposition 16.10 gives us $T = \Theta(\frac{\sqrt{n}}{\varepsilon^2})$, as opposed to $T = \Theta(\frac{\log n}{\varepsilon^2})$ for multiplicative weights.

The problem, at a high level, is that we are "choosing the wrong norm": when dealing with probabilities, the "right" norm is the ℓ_1 norm and not the Euclidean ℓ_2 norm. In the next lecture we will formalize what this means, and how this dependence on n be improved via the Mirror Descent framework.

16.5 *Stronger Assumptions*

If the function f is "well-behaved", we can improve the guarantees for gradient descent in two ways: we can reduce the dependence on ε, and we can weaken (or remove) the dependence on the parameters G and $\|x_1 - x^*\|$. There are two standard assumptions to make on the convex function: that it is "not too flat" (captured by the idea of ***strong convexity***), and it is not "not too curved" (i.e., it is ***smooth***). We now use these assumptions to improve the guarantees.

16.5.1 *Strongly-Convex Functions*

Definition 16.11 (Strong Convexity)**.** A function $f : K \to \mathbb{R}$ is ***α-strongly convex*** if for all $x, y \in K$, any of the following holds:

1. (Zeroth order) $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) - \frac{\alpha}{2}\lambda(1-\lambda)\|x-y\|^2$ for all $\lambda \in [0,1]$.
2. (First order) If f is differentiable, then
$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{\alpha}{2}\|x-y\|^2. \tag{16.15}$$
3. (Second order) If f is twice-differentiable, then all eigenvalues of $H_f(x)$ are at least α at every point $x \in K$.

We will work with the first-order definition, and show that the gradient descent algorithm with (time-varying) step size $\eta_t = O(\frac{1}{\alpha t})$ converges to a value at most $f(x^*) + \varepsilon$ in time $T = \Theta(\frac{G^2}{\alpha\varepsilon})$. Note there is no more dependence on the diameter of the polytope. Before we give this proof, let us give the other relevant definitions.

16.5.2 *Smooth Functions*

Definition 16.12 (Lipschitz Smoothness). A function $f : K \to \mathbb{R}$ is ***β-(Lipschitz)-smooth*** if for all $x, y \in K$, any of the following holds:

1. (Zeroth order) $f(\lambda x + (1-\lambda)y) \geq \lambda f(x) + (1-\lambda)f(y) - \frac{\beta}{2}\lambda(1-\lambda)\|x-y\|^2$ for all $\lambda \in [0,1]$.
2. (First order) If f is differentiable, then
$$f(y) \leq f(x) + \langle \nabla f(x), y - x\rangle + \frac{\beta}{2}\|x-y\|^2. \qquad (16.16)$$
3. (Second order) If f is twice-differentiable, then all eigenvalues of $H_f(x)$ are at most β at every point $x \in K$.

In this case, the gradient descent algorithm with fixed step size $\eta_t = \eta = O(\frac{1}{\beta})$ yields an $\widehat{x}$ which satisfies $f(\widehat{x}) - f(x^*) \leq \varepsilon$ when $T = \Theta(\frac{\beta\|x_1 - x^*\|}{\varepsilon})$. In this case, note we have no dependence on the Lipschitzness G any more; we only depend on the diameter of the polytope. Again, we defer the proof for the moment.

16.5.3 *Well-conditioned Functions*

Functions that are both β-smooth and α-strongly convex are called ***well-conditioned functions***. From the facts above, the eigenvalues of their Hessian H_f must lie in the interval $[\alpha, \beta]$ at all points $x \in K$. In this case, we get a much stronger convergence—we can achieve ε-closeness in time $T = \Theta(\log\frac{1}{\varepsilon})$, where the constant depends on the ***condition number*** $\kappa = \beta/\alpha$.

Theorem 16.13. *For a function f which is β-smooth and α-strongly convex, let x^* be the solution to the unconstrained convex minimization problem $\arg\min_{x\in\mathbb{R}^n} f(x)$. Then running gradient descent with $\eta_t = 1/\beta$ gives*
$$f(x_t) - f(x^*) \leq \frac{\beta}{2}\exp\left(\frac{-t}{\kappa}\right)\|x_1 - x^*\|^2.$$

Proof. For β-smooth f, we can use Definition 16.12 to get
$$f(x_{t+1}) \leq f(x_t) - \eta\|\nabla f(x_t)\|^2 + \eta^2\frac{\beta}{2}\|\nabla f(x_t)\|^2.$$

The right hand side is minimized by setting $\eta = \frac{1}{\beta}$, when we get

$$f(x_{t+1}) - f(x_t) \leq -\frac{1}{2\beta}\|\nabla f(x_t)\|^2. \tag{16.17}$$

For α-strongly-convex f, we can use Definition 16.11 to get:

$$\begin{aligned} f(x_t) - f(x^*) &\leq \langle \nabla f(x_t), x_t - x^* \rangle - \frac{\alpha}{2}\left\|x_t - x^*\right\|^2, \\ &\leq \left\|\nabla f(x_t)\right\| \left\|x_t - x^*\right\| - \frac{\alpha}{2}\left\|x_t - x^*\right\|^2, \\ &\leq \frac{1}{2\alpha}\left\|\nabla f(x_t)\right\|^2, \end{aligned} \tag{16.18}$$

where we use that the right hand side is maximized when $\|x_t - x^*\| = \|\nabla f(x_t)\| / \alpha$. Now combining with (16.17) we have that

$$f(x_{t+1}) - f(x_t) \leq -\frac{\alpha}{\beta}\Big(f(x_t) - f(x^*)\Big), \tag{16.19}$$

or setting $\Delta_t = f(x_t) - f(x^*)$ and rearranging, we get

$$\Delta_{t+1} \leq \left(1 - \frac{\alpha}{\beta}\right)\Delta_t \leq \left(1 - \frac{1}{\kappa}\right)^t \Delta_1 \leq \exp\left(-\frac{t}{\kappa}\right)\cdot \Delta_1.$$

We can control the value of Δ_1 by using (16.16) in $x = x^*, y = x_1$; since $\nabla f(x^*) = 0$, get $\Delta_1 = f(x_1) - f(x^*) \leq \frac{\beta}{2}\left\|x_1 - x^*\right\|^2$. □

Strongly-convex (and hence well-conditioned) functions have the nice property that if $f(x)$ is close to $f(x^*)$ then x is close to x^*: intuitively, since the function is curving at least quadratically, the function values at points far from the minimizer must be significant. Formally, use (16.15) with $x = x^*, y = x_t$ and the fact that $\nabla f(x^*) = 0$ to get

$$\left\|x_t - x^*\right\|^2 \leq \frac{2}{\alpha}(f(x_t) - f(x^*)).$$

We leave it as an exercise to show the claimed convergence bounds using just strong convexity, or just smoothness. (Hint: use the statements proved in (16.17) and (16.18)).

Before we end, a comment on the strong $O(\log 1/\varepsilon)$ convergence result for well-conditioned functions. Suppose the function values lies in $[0,1]$. The $\Theta(\log 1/\varepsilon)$ error bound means that we are correct up to b bits of precision—i.e., have error smaller than $\varepsilon = 2^{-b}$—after $\Theta(b)$ steps. In other words, the number of bits of precision is linear in the number of iterations. The optimization literature refers to this as ***linear convergence***, which can be confusing when you first see it.

16.6 *Extensions and Loose Ends*

16.6.1 *Subgradients*

What if the convex function f is not differentiable? Staring at the proofs above, all we need is the following:

Definition 16.14 (Subgradient)**.** A vector z_x is called a ***subgradient*** at point x if

$$f(y) \geq f(x) + \langle z_x, y - x \rangle \qquad \text{for all } y \in \mathbb{R}^n.$$

Now we can use subgradients at the point x wherever we used $\nabla f(x)$, and the entire proof goes through. In some cases, an approximate subgradient may also suffice.

16.6.2 *Stochastic Gradients, and Coordinate Descent*

16.6.3 *Acceleration*

16.6.4 *Reducing to the Well-conditioned Case*

17 Mirror Descent

The gradient descent algorithm of the previous chapter is general and powerful: it allows us to (approximately) minimize convex functions over convex bodies. Moreover, it also works in the model of online convex optimization, where the convex function can vary over time, and we want to find a low-regret strategy—one which performs well against every fixed point x^*.

This power and broad applicability means the algorithm is not always the best for specific classes of functions and bodies: for instance, for minimizing linear functions over the probability simplex Δ_n, we saw in §16.4.1 that the generic gradient descent algorithm does significantly worse than the specialized Hedge algorithm. This suggests asking: *can we somehow change gradient descent to adapt to the "geometry" of the problem?*

The ***mirror descent*** framework of this section allows us to do precisely this. There are many different (and essentially equivalent) ways to explain this framework, each with its positives. We present two of them here: the ***proximal point*** view, and the ***mirror map*** view, and only mention the others (the ***preconditioned or quasi-Newton gradient flow*** view, and the ***follow the regularized leader*** view) in passing.

17.1 Mirror Descent: the Proximal Point View

Here is a different way to arrive at the gradient descent algorithm from the last lecture: Indeed, we can get an expression for x_{t+1} by

Algorithm 15: Proximal Gradient Descent Algorithm

```
15.1 $x_1 \leftarrow$ starting point
15.2 for $t \leftarrow 1$ to $T$ do
15.3     $x_{t+1} \leftarrow \arg\min_x \{\eta\langle\nabla f_t(x_t), x\rangle + \frac{1}{2}\|x - x_t\|^2\}$
```

setting the gradient of the function to zero; this gives us the expression

$$\eta \cdot \nabla f_t(x_t) + (x_{t+1} - x_t) = 0 \qquad \Longrightarrow \qquad x_{t+1} = x_t - \eta \cdot \nabla f_t(x_t),$$

which matches the normal gradient descent algorithm. Moreover, the intuition for this algorithm also makes sense: if we want to minimize the function $f_t(x)$, we could try to minimize its linear approximation $f_t(x_t) + \langle \nabla f_t(x_t), x - x_t\rangle$ instead. But we should be careful not to "over-fit": this linear approximation is good only close to the point x_t, so we could add in a penalty function (a "regularizer") to prevent us from straying too far from the point x_t. This means we should minimize

$$x_{t+1} \leftarrow \arg\min_x \{ f_t(x_t) + \langle \nabla f_t(x_t), x - x_t\rangle \ + \ \frac{1}{2}\|x - x_t\|^2\}$$

or dropping the terms that don't depend on x,

$$x_{t+1} \leftarrow \arg\min_x \{ \langle \nabla f_t(x_t), x\rangle \ + \ \frac{1}{2}\|x - x_t\|^2\} \tag{17.1}$$

If we have a constrained problem, we can change the update step to:

$$x_{t+1} \leftarrow \arg\min_{x \in K} \{ \eta\langle \nabla f_t(x_t), x\rangle \ + \ \frac{1}{2}\|x - x_t\|^2\} \tag{17.2}$$

The optimality conditions are a bit more complicated now, but they again can show this algorithm is equivalent to projected gradient descent from the previous chapter.

Given this perspective, we can now replace the squared Euclidean norm by other distances to get different algorithms. A particularly useful class of distance functions are *Bregman divergences*, which we now define and use.

17.1.1 *Bregman Divergences*

Given a *strictly* convex function h, we can define a distance based on how the function differs from its linear approximation:

Definition 17.1. The *Bregman divergence* from x to y with respect to function h is

$$D_h(y\|x) := h(y) - h(x) - \langle \nabla h(x), y - x\rangle.$$

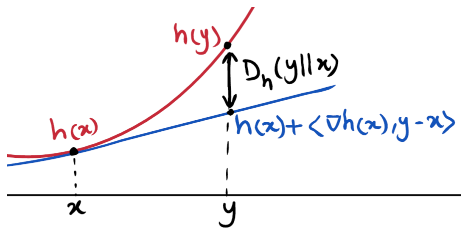


Figure 17.1: $D_h(y\|x)$ for function $h : \mathbb{R} \to \mathbb{R}$.

The figure on the right illustrates this definition geometrically for a univariate function $h : \mathbb{R} \to \mathbb{R}$. Here are a few examples:

1. For the function $h(x) = \frac{1}{2}\|x\|^2$ from $\mathbb{R}^n$ to $\mathbb{R}$, the associated Bregman divergence is

   $$D_h(y\|x) = \tfrac{1}{2}\|y - x\|^2,$$

   the squared Euclidean distance.

2. For the (un-normalized) negative entropy function $h(x) = \sum_{i=1}^n (x_i \ln x_i - x_i)$,
$$D_h(y\|x) = \textstyle\sum_i \left(y_i \ln \frac{y_i}{x_i} - y_i + x_i\right).$$
Using that $\sum_i y_i = \sum_i x_i = 1$ for $y, x \in \Delta_n$ gives us $D_h(y\|x) = \sum_i y_i \ln \frac{y_i}{x_i}$ for $x, y \in \Delta_n$: this is the *Kullback-Leibler (KL) divergence* between probability distributions.

Many other interesting Bregman divergences can be defined.

17.1.2 *Changing the Distance Function*

Since the distance function $\frac{1}{2}\|x-y\|^2$ in (17.1) is a Bregman divergence, what if we replace it by a generic Bregman divergence: what algorithm do we get in that case? Again, let us first consider the unconstrained problem, with the update:

$$x_{t+1} \leftarrow \arg\min_x \{\eta\langle \nabla f_t(x_t), x\rangle \ + \ D_h(x\|x_t)\}.$$

Again, setting the gradient at x_{t+1} to zero (i.e., the optimality condition for x_{t+1}) now gives:

$$\eta \nabla f_t(x_t) + \nabla h(x_{t+1}) - \nabla h(x_t) = 0,$$

or, rephrasing

$$\nabla h(x_{t+1}) = \nabla h(x_t) - \eta \nabla f_t(x_t) \tag{17.3}$$
$$\implies x_{t+1} = \nabla h^{-1}\big(\nabla h(x_t) - \eta \nabla f_t(x_t)\big) \tag{17.4}$$

Let's consider this for our two running examples:

1. When $h(x) = \frac{1}{2}\|x\|^2$, the gradient $\nabla h(x) = x$. So we get
$$x_{t+1} = x_t - \eta \nabla f_t(x_t),$$
the standard gradient descent update.

2. When $h(x) = \sum_i (x_i \ln x_i - x_i)$, then $\nabla h(x) = (\ln x_1, \ldots, \ln x_n)$, so
$$(x_{t+1})_i = e^{\ln (x_t)_i - \eta \nabla f_t(x_t)} = (x_t)_i \, e^{-\eta \nabla f_t(x_t)}.$$
Now if $f_t(x) = \langle \ell_t, x\rangle$, its gradient is just the vector ℓ_t, and we get back precisely the *weights* maintained by the Hedge algorithm!

The same ideas also hold for constrained convex minimization: we now have to search for the minimizer within the set K. In this case the algorithm using negative entropy results in the same Hedge-like update, followed by scaling the point down to get a probability vector, thereby giving the *probability values* in Hedge.

To summarize: this algorithm that tries to minimize the linear ap-

What would be the "right" choice of h to minimize the function f? It would be $h = f$, because adding $D_f(x\|x_t)$ to the linear approximation of f at x_t gives us back exactly f. Of course, the update now requires us to minimize $f(x)$, which is the original problem. So we should choose an h that is "similar" to f, and yet such that the update step is tractable.

Algorithm 16: Proximal Gradient Descent Algorithm

```
16.1 $x_1 \leftarrow$ starting point
16.2 for $t \leftarrow 1$ to $T$ do
16.3     $x_{t+1} \leftarrow \arg\min_{x \in K}\{\eta\langle\nabla f_t(x_t), x\rangle + D_h(x\|x_t)\}$
```

proximation of the function, regularized by a Bregman distance D_h, gives us vanilla gradient descent for one choice of h (which is good for quadratic-like functions over Euclidean space), and Hedge for another choice of h (which is good for linear functions over the space of probability distributions). Indeed, depending on how we choose the function, we can get different properties from this algorithm—this is the *mirror descent framework*.

17.2 *Mirror Descent: The Mirror Map View*

A different view of the mirror descent framework is the one originally presented by Nemirovski and Yudin. They observe that in gradient descent, at each step we set $x_{t+1} = x_t - \eta\nabla f_t(x_t)$. However, the gradient was actually defined as a linear functional on $\mathbb{R}^n$ and hence naturally belongs to the dual space of $\mathbb{R}^n$. The fact that we represent this functional (i.e., this *covector*) as a *vector* is a matter of convenience, and we should exercise care.

A *linear functional* on vector space X is a linear map from X into its underlying field $\mathbb{F}$.

In the vanilla gradient descent method, we were working in $\mathbb{R}^n$ endowed with ℓ_2-norm, and this normed space is self-dual, so it is perhaps reasonable to combine points in the primal space (the iterates x_t of our algorithm) with objects in the dual space (the gradients). But when working with other normed spaces, adding a covector $\nabla f_t(x_t)$ to a vector x_t might not be the right thing to do. Instead, Nemirovski and Yudin propose the following:

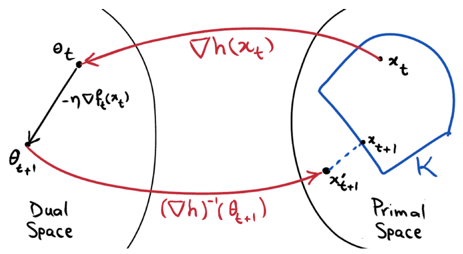


Figure 17.2: The four basic steps in each iteration of the mirror descent algorithm

1. we map our current point x_t to a point θ_t in the dual space using a ***mirror map***.

2. Next, we take the gradient step
$$\theta_{t+1} \leftarrow \theta_t - \eta\nabla f_t(x_t).$$

3. We map θ_{t+1} back to a point in the primal space x'_{t+1} using the inverse of the mirror map from Step 1.

4. If we are in the constrained case, this point x'_{t+1} might not be in the convex feasible region K, so we to project x'_{t+1} back to a "close-by" x_{t+1} in K.

The name of the process comes from thinking of the *dual space as being a mirror image of the primal space*. But how do we choose these mirror maps? Again, this comes down to understanding the geometry of the problem, the kinds of functions and the set K we care about, and the kinds of guarantees we want. In order to discuss these, let us discuss the notion of norms in some more detail.

17.2.1 *Norms and their Duals*

Definition 17.2 (Norm)**.** A function $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$ is a *norm* if

- If $\|x\| = 0$ for $x \in \mathbb{R}^n$, then $x = 0$;
- for $\alpha \in \mathbb{R}$ and $x \in \mathbb{R}^n$ we have $\|\alpha x\| = |\alpha| \|x\|$; and
- for $x, y \in \mathbb{R}^n$ we have $\|x + y\| \leq \|x\| + \|y\|$.

The well-known ℓ_p-norms for $p \geq 1$ are defined by

$$\|x\|_p := (\sum_{i=1}^{n} |x_i|^p)^{1/p}$$

for $x \in \mathbb{R}^n$. The ℓ_∞-norm is given by

$$\|x\|_\infty := \max_{i=1}^{n} |x_i|$$

for $x \in \mathbb{R}^n$.

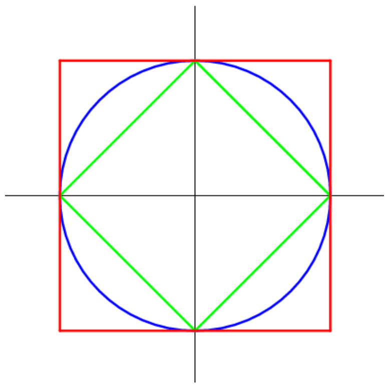


Figure 17.3: The unit ball in ℓ_1-norm (Green), ℓ_2-norm (Blue), and ℓ_∞-norm (Red).

Definition 17.3 (Dual Norm)**.** Let $\|\cdot\|$ be a norm. The dual norm of $\|\cdot\|$ is a function $\|\cdot\|_*$ defined as

$$\|y\|_* := \sup\{\langle x, y\rangle \ : \ \|x\| \leq 1\}.$$

The dual norm of the ℓ_2-norm is again the ℓ_2-norm; the Euclidean norm is self-dual. The dual for the ℓ_p-norm is the ℓ_q-norm, where $1/p + 1/q = 1$.

Corollary 17.4 (Cauchy-Schwarz for General Norms)**.** *For $x, y \in \mathbb{R}^n$, we have $\langle x, y\rangle \leq \|x\| \, \|y\|_*$.*

Proof. Assume $\|x\| \neq 0$, otherwise both sides are 0. Since $\|x/\|x\|\| = 1$, we have $\langle x/\|x\|, y\rangle \leq \|y\|_*$. □

Theorem 17.5. *For a finite-dimensional space with norm $\|\cdot\|$, we have $(\|\cdot\|_*)_* = \|\cdot\|$.*

Using the notion of dual norms, we can give an alternative characterization of Lipschitz continuity for a norm $\|\cdot\|$, much like Fact 16.6 for Euclidean norms:

Fact 17.6. For f be a differentiable function. Then f is G-Lipschitz with respect to norm $\|\cdot\|$ if and only if for all $x \in \mathbb{R}$,

$$\|\nabla f(x)\|_* \leq G.$$

17.2.2 *Defining the Mirror Maps*

To define a mirror map, we first fix a norm $\|\cdot\|$, and then choose a differentiable convex function $h : \mathbb{R}^n \to \mathbb{R}$ that is α-strongly-convex with respect to this norm. Recall from §16.5.1 that such a function must satisfy

$$h(y) \geq h(x) + \langle \nabla h(x), y - x \rangle + \frac{\alpha}{2}\|y - x\|^2.$$

We use two familiar examples:

1. $h(x) = \frac{1}{2}\|x\|_2^2$ is 1-strongly convex with respect to $\|\cdot\|_2$, and
2. $h(x) := \sum_{i=1}^n x_i(\log x_i - 1)$ is 1-strongly convex with respect to $\|\cdot\|_1$; the proof of this is called ***Pinsker's inequality***.

Check out the two proofs pointed to by Aryeh Kontorovich, or this proof (part 1, part 2) by Madhur Tulsiani.

Having fixed $\|\cdot\|$ and h, the ***mirror map*** is

$$\nabla(h) : \mathbb{R}^n \to \mathbb{R}^n.$$

Since h is differentiable and strongly-convex, we can define the inverse map as well. This defines the mappings that we use in the Nemirovski-Yudin process: we set

The function h used in this way is often called a *distance-generating* function.

$$\theta_t = \nabla h(x_t) \qquad \text{and} \qquad x'_{t+1} = (\nabla h)^{-1}(\theta_{t+1}).$$

For our first running example of $h(x) = \frac{1}{2}\|x\|^2$, the gradient (and hence its inverse) is the identity map. For the (un-normalized) negative entropy example, $(\nabla h(x))_i = \ln x_i$, and hence $(\nabla h)^{-1}(\theta)_i = e^{\theta_i}$.

17.2.3 *The Algorithm (Again)*

Let us formally state the algorithm again, before we state and prove a theorem about it. Suppose we want to minimize a convex function f over a convex body $K \subseteq \mathbb{R}^n$. We first fix a norm $\|\cdot\|$ on $\mathbb{R}^n$ and choose a distance-generating function $h : \mathbb{R}^n \to \mathbb{R}$, which gives the mirror map $\nabla h : \mathbb{R}^n \to \mathbb{R}^n$. In each iteration of the algorithm, we do the following:

(i) Map to the dual space $\theta_t \leftarrow \nabla h(x_t)$.

(ii) Take a gradient step in the dual space: $\theta_{t+1} \leftarrow \theta_t - \eta_t \cdot \nabla f_t(x_t)$.

(iii) Map θ_{t+1} back to the primal space $x'_{t+1} \leftarrow (\nabla h)^{-1}(\theta_{t+1})$.

(iv) Project x'_{t+1} back into the feasible region K by using the Bregman divergence: $x_{t+1} \leftarrow \min_{x \in K} D_h(x\|x'_{t+1})$. In case $x'_{t+1} \in K$, e.g., in the unconstrained case, we get $x_{t+1} = x'_{t+1}$.

Note that the choice of h affects almost every step of this algorithm.

17.3 The Analysis

We prove the following guarantee for mirror descent, which captures the guarantees for both Hedge and gradient descent, and for other variants that you may use.

The theorem is stated for the unconstrained version, but extending it to the constrained version is an easy exercise.

Theorem 17.7 (Mirror Descent Regret Bound)**.** *Let $\|\cdot\|$ be a norm on $\mathbb{R}^n$, and h be an α-strongly convex function with respect to $\|\cdot\|$. Given $f_1, \ldots, f_T$ be convex, differentiable functions such that $\|\nabla f_t\|_* \leq G$, the mirror descent algorithm starting with x_0 and taking constant step size η in every iteration produces $x_1, \ldots, x_T$ such that for any $x^* \in \mathbb{R}^n$,*

$$\sum_{t=1}^{T} f_t(x_t) \leq \sum_{t=1}^{T} f_t(x^*) + \underbrace{\frac{D_h(x^* \| x_1)}{\eta} + \frac{\eta \sum_{t=1}^{T} \|\nabla f_t(x_t)\|_*^2}{2\alpha}}_{regret}. \qquad (17.5)$$

Before proving Theorem 17.7, observe that when $\|\cdot\|$ is the ℓ_2-norm and $h = \frac{1}{2}\|\cdot\|^2$, the regret term is

$$\frac{\|x^* - x_1\|_2^2}{2\eta} + \frac{\eta \sum_{t=1}^{T} \|\nabla f_t(x_t)\|_2^2}{2},$$

which is what Theorem 16.8 guarantees. Similarly, if $\|\cdot\|$ is the ℓ_1-norm and h is the negative entropy, the regret versus any point $x^* \in \Delta_n$ is

$$\frac{1}{\eta} \sum_{i=1}^{n} x_i^* \ln \frac{x_i^*}{(x_1)_i} + \frac{\eta \sum_{t=1}^{T} \|\nabla f_t(x_t)\|_\infty^2}{2/\ln 2}.$$

For linear functions $f_t(x) = \langle \ell_t, x \rangle$ with $\ell_t \in [-1,1]^n$, and $x_1 = 1/n\mathbb{1}$, the regret is

$$\frac{KL(x^* \| x_1)}{\eta} + \frac{\eta T}{2/\ln 2} \quad \leq \quad \frac{\ln n}{\eta} + \eta T.$$

The last inequality uses that the KL divergence to the uniform distribution is at most $\ln n$. (Exercise!) In fact, this suggests a way to improve the regret bound: if we start with a distribution x_1 that is closer to x^*, the first term of the regret gets smaller.

17.3.1 The Proof of Theorem 17.7

The proof here is very similar in spirit to that of Theorem 16.8: we give a potential function

$$\Phi_t = \frac{D_h(x^* \| x_t)}{\eta}$$

and bound the amortized cost at time t as follows:

$$f_t(x_t) - f_t(x^*) + (\Phi_{t+1} - \Phi_t) \leq f_t(x^*) + \text{blah}_t. \qquad (17.6)$$

Summing over all times,

$$\begin{aligned}\sum_{t=1}^{T} f_t(x_t) - \sum_{t=1}^{T} f_t(x^*) &\leq \Phi_1 - \Phi_{T+1} + \sum_{t=1}^{T} \text{blah}_t \\ &\leq \Phi_1 + \sum_{t=1}^{T} \text{blah}_t = \frac{D_h(x^* \| x_1)}{\eta} + \sum_{t=1}^{T} \text{blah}_t.\end{aligned}$$

The last inequality above uses that the Bregman divergence is always non-negative for convex functions.

To complete the proof, it remains to show that blah_t in inequality (17.6) can be made $\frac{\eta}{2\alpha}\|\nabla f_t(x_t)\|_*^2$. Let us focus on the unconstrained case, where $x_{t+1} = x'_{t+1}$. The calculations below are fairly routine, and can be skipped at the first reading:

$$\begin{aligned}\Phi_{t+1} - \Phi_t &= \frac{1}{\eta}\big(D_h(x^* \| x_{t+1}) - D_h(x^* \| x_t)\big) \\ &= \frac{1}{\eta}\big(h(x^*) - h(x_{t+1}) - \langle \underbrace{\nabla h(x_{t+1})}_{\theta_{t+1}}, x^* - x_{t+1}\rangle - h(x^*) + h(x_t) + \langle \underbrace{\nabla h(x_t)}_{\theta_t}, x^* - x_t\rangle\big) \\ &= \frac{1}{\eta}\big(h(x_t) - h(x_{t+1}) - \langle \theta_t - \eta \underbrace{\nabla f_t(x_t)}_{\nabla_t}, x^* - x_{t+1}\rangle + \langle \theta_t, x^* - x_t\rangle\big) \\ &= \frac{1}{\eta}\big(h(x_t) - h(x_{t+1}) - \langle \theta_t, x_t - x_{t+1}\rangle + \eta\langle \nabla f_t(x_t), x^* - x_{t+1}\rangle\big) \\ &\leq \frac{1}{\eta}\Big(-\frac{\alpha}{2}\|x_{t+1} - x_t\|^2 + \eta\langle \nabla f_t(x_t), x^* - x_{t+1}\rangle\Big) \qquad \text{(By } \alpha\text{-strong convexity of } h \text{ wrt to } \|\cdot\|\text{)}\end{aligned}$$

Substituting this back into (17.6):

$$\begin{aligned}&f_t(x_t) - f_t(x^*) + (\Phi_{t+1} - \Phi_t) \\ &\quad\leq f_t(x_t) - f_t(x^*) - \frac{\alpha}{2\eta}\|x_{t+1} - x_t\|^2 + \langle \nabla f_t(x_t), x^* - x_{t+1}\rangle \\ &\quad\leq \underbrace{f_t(x_t) - f_t(x^*) + \langle \nabla f_t(x_t), x^* - x_t\rangle}_{\leq 0 \text{ by convexity of } f_t} - \frac{\alpha}{2\eta}\|x_{t+1} - x_t\|^2 + \langle \nabla f_t(x_t), x_t - x_{t+1}\rangle \\ &\quad\leq -\frac{\alpha}{2\eta}\|x_{t+1} - x_t\|^2 + \|\nabla f_t(x_t)\|_* \|x_t - x_{t+1}\| \qquad \text{(By Corollary 17.4)} \\ &\quad\leq -\frac{\alpha}{2\eta}\|x_{t+1} - x_t\|^2 + \frac{1}{2}\Big(\frac{\eta}{\alpha}\|\nabla f_t(x_t)\|_*^2 + \frac{\alpha}{\eta}\|x_t - x_{t+1}\|^2\Big) \quad \text{(By AM-GM)} \\ &\quad= \frac{\eta}{2\alpha}\|\nabla f_t(x_t)\|_*^2.\end{aligned}$$

This completes the proof of Theorem 17.7. As you observe, it is syntactically similar to the original proof of gradient descent, just using more general language. In order to extend this to the constrained case, we will need to show that if $x'_{t+1} \notin K$, and $x_{t+1} = \arg\min_{x \in K} D_h(x \| x'_{t+1})$, then

$$D_h(x^* \| x_{t+1}) \leq D_h(x^* \| x'_{t+1})$$

for any $x^* \in K$. This is a "Generalized Pythagoras Theorem" for Bregman distance, and is left as an exercise.

17.4 *Alternative Views of Mirror Descent*

To complete and flesh out. In this lecture, we reviewed mirror descent algorithm as a gradient descent scheme where we do the gradient step in the dual space. We now provide some alternative views of mirror descent.

17.4.1 *Preconditioned Gradient Descent*

For any given space which we use a descent method on, we can linearly transform the space with some map Q to make the geometry more regular. This technique is known as preconditioning, and improves the speed of the descent. Using the linear transformation Q, our descent rule becomes

$$x_{t+1} = x_t - \eta\, H_h(x_t)^{-1}\, \nabla f(x_t).$$

Some of you may have seen Newton's method for minimizing convex functions, which has the following update rule:

$$x_{t+1} = x_t - \eta\, H_f(x_t)^{-1}\, \nabla f(x_t).$$

This means mirror descent replaces the Hessian of the function itself by the Hessian of a strongly convex function h. Newton's method has very strong convergence properties (it gets error ε in $O(\log\log 1/\varepsilon)$ iterations!) but is not "robust"—it is only guaranteed to converge when the starting point is "close" to the minimizer. We can view mirror descent as trading off the convergence time for robustness. Fill in more on this view.

17.4.2 *As Follow the Regularized Leader*

18
The Centroid and Ellipsoid Algorithms

In this chapter, we discuss some algorithms for convex programming that have $O(\log 1/\varepsilon)$-type convergence guarantees (under suitable assumptions). This leads to polynomial-time algorithms for Linear Programming problems. In particular, we examine the Center-of-Gravity and Ellipsoid algorithms in depth.

18.1 The Centroid Algorithm

In this section, we discuss the Centroid Algorithm in the context of constrained convex minimization. Besides being interesting in its own right, it is a good lead-in to Ellipsoid, since it gives some intuition about high-dimensional bodies and their volumes.

Given a convex body $K \subseteq \mathbb{R}^n$ and a convex function $f : \mathbb{R}^n \to \mathbb{R}$, we want to approximately minimize $f(x)$ over $x \in K$. First, recall that the ***centroid*** of a set K is the point $c \in \mathbb{R}^n$ such that

This is the natural analog of the centroid of n points $x_1, x_2, \ldots, x_N$, which is defined as $\frac{\sum_i x_i}{N}$. See this blog post for a discussion about the centroid of an arbitrary measure μ defined over $\mathbb{R}^n$.

$$c := \frac{\int_{x \in K} x \, dx}{\text{vol}(K)} = \frac{\int_{x \in K} x \, dx}{\int_{x \in K} dx},$$

where $\text{vol}(K)$ is the volume of the set K. The following lemma captures the crucial fact about the center-of-gravity that we use in our algorithm.

Lemma 18.1 (Grünbaum's Lemma). *For any convex set $K \in \mathbb{R}^n$ with a centroid $c \in \mathbb{R}^n$, and any halfspace $H = \{x \mid a^\intercal(x - c) \geq 0\}$ passing through c,*

$$\frac{1}{e} \leq \frac{\text{vol}(K \cap H)}{\text{vol}(K)} \leq \left(1 - \frac{1}{e}\right).$$

This bound is the best possible: e.g., consider the probability simplex Δ_n with centroid $\frac{1}{n}\mathbb{1}$. Finish this argument.

18.1.1 The Algorithm

In 1965, A. Yu. Levin and Donald Newman independently (and on

Levin (1965)
Newman (1965)

opposite sides of the iron curtain) proposed the following algorithm.

Algorithm 17: Centroid(K, f, T)

```
17.1 $K_1 \leftarrow K$
17.2 for $t = 1, \ldots T$ do
17.3     at step $t$, let $c_t \leftarrow$ centroid of $K_t$
17.4     $K_{t+1} \leftarrow K_t \cap \{x \mid \langle \nabla f(c_t), x - c_t \rangle \leq 0\}$
17.5 return $\widehat{x} \leftarrow \arg\min_{t \in \{1,\ldots,T\}} f(c_t)$
```

The figure to the right shows a sample execution of the algorithm, where K is initially a ball. (Ignore the body K^ε for now.) We find the centroid c_1 and compute the gradient $\nabla f(c_1)$. Instead of moving in the direction opposite to the gradient, we consider the halfspace H_1 of vectors negatively correlated with the gradient, restrict our search to $K \leftarrow K \cap H_1$, and continue. We repeat this step some number of times, and then return the smallest of the function value at all the centroids seen by the algorithm.

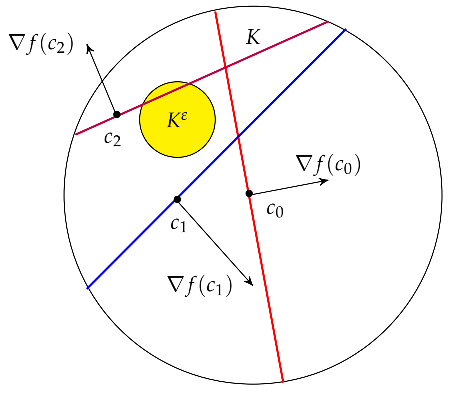


Figure 18.1: Sample execution of first three steps of the Centroid Algorithm.

18.1.2 An Analysis of the Centroid Algorithm

Theorem 18.2. *Let $B \geq 0$ such that $f : K \to [-B, B]$. If $\widehat{x}$ is the result of the algorithm, and $x^* = \arg\min_{x \in K} f(x)$, then*

$$f(\widehat{x}) - f(x^*) \leq 4B \cdot \exp(-T/3n).$$

Hence, for any $\varepsilon \leq 1$, as long as $T \geq 3n \ln \frac{4B}{\varepsilon}$,

$$f(\widehat{x}) - f(x^*) \leq \varepsilon.$$

Proof. For some $\delta \leq 1$, define the body

$$K^\delta := \{(1-\delta)x^* + \delta x \mid x \in K\}$$

as a scaled-down version of K centered at x^*. The following facts are immediate:

1. $\operatorname{vol}(K^\delta) = \delta^n \cdot \operatorname{vol}(K)$.

2. The value of f on any point $y = (1-\delta)x^* + \delta x \in K^\delta$ is

$$\begin{aligned} f(y) = f((1-\delta)x^* + \delta x) &\leq (1-\delta)f(x^*) + \delta f(x) \leq (1-\delta)f(x^*) + \delta B \\ &\leq f(x^*) + \delta(B - f(x^*)) \leq f(x^*) + 2\delta B. \end{aligned}$$

Using Grünbaum's lemma, the volume falls by a constant factor in each iteration, so $\operatorname{vol}(K_t) \leq \operatorname{vol}(K) \cdot (1 - \frac{1}{e})^t$. If we define $\delta := 2(1 - 1/e)^{T/n}$, then after T steps the volume of K_T is smaller than that of K^δ, so some point of K^δ must have been cut off.

Consider such a step t such that $K^\delta \subseteq K_t$ but $K^\delta \not\subseteq K_{t+1}$. Let $y \in K^\delta \cap (K_t \setminus K_{t+1})$ be a point that is "cut off". By convexity we have

$$f(y) \geq f(c_t) + \langle \nabla f(c_t), y - c_t \rangle;$$

moreover, $\langle \nabla f(c_t), y - c_t \rangle > 0$ since the cut-off point $y \in K_t \setminus K_{t+1}$. Hence the corresponding centroid has value$f(c_t) < f(y) \leq f(x^*) + 2\delta B$. Since $\widehat{x}$ is the centroid with the smallest function value, we get

$$f(\widehat{x}) - f(x^*) \leq 2B \cdot 2(1 - 1/e)^{T/n}) \leq 4B \exp(-T/3n).$$

The second claim follows by substituting $T \geq 3n \ln \frac{4B}{\varepsilon}$ into the first claim, and simplifying. □

18.1.3 *Some Comments on the Runtime*

The number of iterations T to get an error of ε depends on $\log(1/\varepsilon)$; compare this linear convergence to gradient descent requiring $O(1/\varepsilon^2)$ steps. One downside with this approach is that the number of iterations explicitly depends on the number of dimensions n, whereas gradient descent depends on other factors (a bound on the gradient, and the diameter of the polytope), but not explicitly on the dimension.

However, the all-important question is: *how do we compute the centroid*? This is a difficult problem—it is #P-hard to do exactly, which means it is at least as hard as counting the number of satisfying assignments to a SAT instance. In 2002, Bertsimas and Vempala suggested a way to find approximate centroids by sampling random points from convex bodies (which in turn is done via random walks). More details here.

18.2 *The Ellipsoid Algorithm*

The Ellipsoid algorithm is usually attributed to Naum Shor; the fact that this algorithm gives a polynomial-time algorithm for linear programming was a breakthrough result due to Khachiyan, and was front page news at the time. A great source of information about this algorithm is the Grötschel-Lovász-Schrijver book. A historical perspective appears in this this survey by Bland, Goldfarb, and Todd.

N. Z. Šor and N. G. Žurbenko (1971)

Khachiyan (1979)

Grötschel, Lovász, and Schrijver (1988)

Let us mention some theorem statements about the Ellipsoid algorithm that are most useful in designing algorithms. The second-most important theorem is the following. Recall the notion of an extreme point or basic feasible solution (bfs) from §7.1.2. Let $\langle A \rangle, \langle b \rangle, \langle c \rangle$ denote the number of bits required to represent of A, b, c respectively.

Theorem 18.3 (Linear Programming in Polynomial Time)**.** *Given a linear program* $\min\{c^\intercal x \mid Ax \geq b\}$, *the Ellipsoid algorithm produces an optimal vertex solution for the LP, in time polynomial in* $\langle A \rangle$, $\langle b \rangle$, $\langle c \rangle$.

One may ask: does the runtime depend on the bit-complexity of the input because doing basic arithmetic on these numbers may require large amounts of time. Unfortunately, that is not the case. Even if we count the number of arithmetic operations we need to perform, the Ellipsoid algorithm performs $\text{poly}(\langle A\rangle + \langle b\rangle + \langle c\rangle)$ operations. A stronger guarantee would have been for the number of arithmetic operations to be $\text{poly}(m, n)$, where the matrix $A \in \mathbb{Q}^{m\times n}$: such an algorithm would be called a ***strongly polynomial-time*** algorithm. Obtaining such an algorithm remains a major open question.

18.2.1 *Separation Implies Optimization*

In order to talk about the Ellipsoid algorithm, as well as to state the next (and most important) theorem about Ellipsoid, we need a definition.

Definition 18.4 (Strong Separation Oracle)**.** For a convex set $K \subseteq \mathbb{R}^n$, a *strong separation oracle* for K is an algorithm that takes a point $z \in \mathbb{R}^n$ and correctly outputs one of:

(i) `Yes` (i.e., $z \in K$), or

(ii) `No` (i.e., $z \notin K$), as well as a *separating hyperplane* given by $a \in \mathbb{R}^n, b \in \mathbb{R}$ such that $K \subseteq \{x \mid \langle a, x\rangle \le b\}$ but $\langle a, x\rangle > b$.

The example on the right shows a separating hyperplane.

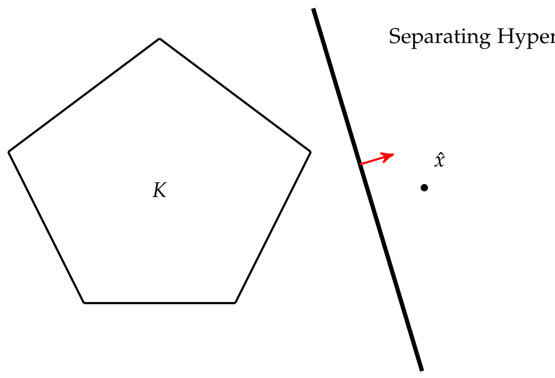


Figure 18.2: Example of separating hyperplanes

Theorem 18.5 (Separation implies Optimization)**.** *Given an LP*

$$\min\{c^\intercal x \mid x \in K\}$$

for a polytope $K = \{x \mid Ax \ge b\} \subseteq \mathbb{R}^n$, and given access to a strong separation oracle for K, the Ellipsoid algorithm produces a vertex solution for the LP in time $\text{poly}(n, \max_i \langle a_i\rangle, \max_i \langle b_i\rangle, \langle c\rangle)$.

There is no dependence on the number of constraints in the LP; we can get a basic solution to any finite LP as long as each constraint has a reasonable bit complexity, and we can define a separation oracle for the polytope. This is often summarized by saying: *"separation implies optimization"*. Let us give two examples of exponential-sized LPs, for which we can give a separation oracles, and hence optimize over them.

18.3 *Ellipsoid for LP Feasibility*

Instead of solving a linear program, suppose we are given a description of some polytope K, and want to either find some point $x \in K$, or to report that K is the empty set. This *feasibility* problem is no harder than optimization over the polytope; in fact, the GLS book shows that feasibility is not much easier than optimization: under certain conditions, the two problems are essentially equivalent to each other. (Very loosely, we can do binary search on the objective function.)

Given this, let's discuss solving a feasibility problem; this will allow us to illustrate some of the major ideas of the Ellipsoid algorithm. Given some description of a polytope K, and two scalars $R, r > 0$, suppose we are guaranteed that

(a) $K \subseteq \text{Ball}(0, R)$, and

(b) either $K = \emptyset$, or else some ball $\text{Ball}(c, r) \subseteq K$ for some $c \in \mathbb{R}^n$.

The feasibility problem is to figure out which of the two cases in condition (b) holds; moreover, if $K \neq \emptyset$ then we also need to find a point $x \in K$. We assume that K is given by a strong separation oracle:

Theorem 18.6 (Idealized Feasibility using Ellipsoid)**.** *Given K, r, R as above (and a strong separation oracle for K), the feasibility problem can be solved using $O(n \log(R/r))$ oracle calls.*

For this result, and the rest of the chapter: let us assume that we can perform ***exact arithmetic on real numbers***. This assumption is with considerable loss in generality, since the algorithm takes square-roots when computing the new ellipsoid. If were to round numbers when doing this, that could create all sorts of numerical problems, and a large part of the complication in the actual algorithms comes from these numerical issues.

Proof. The basic structure is simple, and reminiscent of the Centroid algorithm. At each iteration t, we have a current ellipsoid $\mathcal{E}_t$ guaranteed to contain the set K (assuming we have not found a point $x \in K$ yet). The initial ellipsoid is $\mathcal{E}_0 = \text{Ball}(0, R)$, so condition (a) guarantees that $K \subseteq \mathcal{E}_0$.

In iteration t, we ask the separation oracle whether the center c_t of ellipsoid $\mathcal{E}_t$ belongs to K? If the oracle answers `Yes`, we are done. Else the oracle returns a separating hyperplane $\langle a, x \rangle = b$, such that $\langle a, c_t \rangle > b$ but $K \subseteq H_t := \{x : \langle a, x \rangle \leq b\}$. Consequently, $K \subseteq \mathcal{E}_t \cap H_t$. Moreover, the half-space H_t does not contain the center c_t of the ellipsoid, so $\mathcal{E}_t \cap H_t$ is less than half the entire ellipsoid. The crucial idea is to find another (small-volume) ellipsoid $\mathcal{E}_{t+1}$ containing this piece $\mathcal{E}_t \cap H_t$ (and hence also K). This allows us to continue.

To show progress, we need that the volume $\text{vol}(\mathcal{E}_{t+1})$ is considerably smaller than $\text{vol}(\mathcal{E}_t)$. In §18.5 we show that the ellipsoid $\mathcal{E}_{t+1} \supseteq \mathcal{E}_t \cap H_t$ has volume

$$\frac{\text{vol}(\mathcal{E}_{t+1})}{\text{vol}(\mathcal{E}_t)} \leq e^{-\frac{1}{2(n+1)}}.$$

Therefore, after $2(n+1)$ iterations, the ratio of the volumes falls by at least a factor of $\frac{1}{e}$. Now we are done, because our assumptions say that

This volume reduction is much weaker by a factor of n compared to that of the Centroid algorithm, so it is often worth considering if applications of the Ellipsoid algorithm can be replaced by the Centroid algorithm.

$$\text{vol}(K) \leq \text{vol}(\text{Ball}(0, R)),$$

and that

$$K \neq \emptyset \iff \text{vol}(K) \geq \text{vol}(\text{Ball}(0, r)).$$

Hence, if after $2(n+1)\ln(R/r)$ steps, none of the ellipsoid centers have been inside K, we know that K must be empty. □

18.3.1 Finding Vertex Solutions for LPs

There are several issues that we need to handle when solving LPs using this approach. For instance, the polytope may not be full-dimensional, and hence we do not have any non-trivial ball within K. Our separation oracles may only be approximate. Moreover, all the numerical calculations may only be approximate.

Even after we take care of these issues, we are working over the rationals so binary search-type techniques may not be able to get us to a vertex solution. So finally, when we have a solution x_t that is "close enough" to x^*, we need to "round" it and get a vertex solution. In a single dimension we can do the following (and this idea already appeared in a homework problem): we know that the optimal solution x^* is a rational whose denominator (when written in reduced terms) uses at most some b bits. So we find a solution within distance to x^* is smaller than some δ. Moreover δ is chosen to be small enough such that there is a unique rational with denominator smaller than 2^b in the δ-ball around x_t. This rational can only be x^*, so we can "round" x_t to it.

Consider the case where we perform binary-search over the interval $[0, 1]$ and want to find the point $1/3$: no number of steps will get us exactly to the answer.

In higher dimensions, the analog of this is a technique (due to Lovász) called *simultaneous Diophantine equations*.

18.4 Ellipsoid for Convex Optimization

Now we want to solve $\min\{f(x) \mid x \in K\}$. Again, assume that K is given by a strong separation oracle, and we have numbers R, r such that $K \subseteq \text{Ball}(0, R)$, and K is either empty or contains a ball of radius r. The general structure is a one familiar by now, and combines ideas from both the previous sections.

1. Let the starting point $x_1 \leftarrow 0$, the starting ellipsoid be $\mathcal{E}_1 \leftarrow \text{Ball}(0, R)$, and the starting convex set $K_1 \leftarrow K$.

2. At time t, ask the separation oracle: "Is the center c_t of ellipsoid $\mathcal{E}_t$ in the convex body K_t?"

   Yes: Define half-space $H_t := \{x \mid \langle \nabla f(c_t), x - c_t \rangle \leq 0\}$. Observe that $K_t \cap H_t$ contains all points in K_t with value at most $f(c_t)$.

   No: In this case the separation oracle also gives us a separating hyperplane. This defines a half-space H_t such that $c_t \notin H_t$, but $K_t \subseteq H_t$.

   In both cases, set $K_{t+1} \leftarrow K_t \cap H_t$, and $\mathcal{E}_{t+1}$ to an ellipsoid containing $\mathcal{E}_t \cap H_t$. Since we knew that $K_t \subseteq \mathcal{E}_t$, we maintain that $K_{t+1} \subseteq \mathcal{E}_{t+1}$.

3. Finally, after $T = 2n(n+1)\ln(R/r)$ rounds either we have not seen any point in K—in which case we say "K is empty"—or else we output
$$\widehat{x} \leftarrow \arg\min\{f(c_t) \mid c_t \in K_t, t \in 1 \ldots T\}.$$

One subtle issue: we make queries to a separation oracle for K_t, but we are promised only a separation oracle for $K_1 = K$. However, we can build separation oracles for H_t inductively: indeed, given strong separation oracle for K_{t-1}, we build one for $K_t = K_{t-1} \cap H_{t-1}$ as follows:

Given $z \in \mathbb{R}^n$, query the oracle for K_{t-1} at z. If $z \notin K_{t-1}$, the separating hyperplane for K_{t-1} also works for K_t. Else, if $z \in K_{t-1}$, check if $z \in H_{t-1}$. If so, $z \in K_t = K_{t-1} \cap H_{t-1}$. Otherwise, the defining hyperplane for halfspace H_{t-1} is a separating hyperplane between z and K_t.

Now adapting the analysis from the previous sections gives us the following result (assuming exact arithmetic again):

Theorem 18.7 (Idealized Convex Minimization using Ellipsoid)**.** *Given K, r, R as above (and a strong separation oracle K), and a function $f : K \to [-B, B]$, the Ellipsoid algorithm run for T steps either correctly reports that $K = \emptyset$, or else produces a point $\widehat{x}$ such that*
$$f(\widehat{x}) - f(x^*) \leq \frac{2BR}{r} \exp\left\{-\frac{T}{2n(n+1)}\right\}.$$

Note the similarity to Theorem 18.2, as well as the differences: the exponential term is slower by a factor of $2(n+1)$. This is because the volume of the successive ellipsoids shrinks much slower than in Grünbaum's lemma. Also, we lose a factor of R/r because K is potentially smaller than the starting body by precisely this factor. (Again, this presentation ignores precision issues, and assumes we can do exact real arithmetic.)

18.5 Getting the New Ellipsoid

This brings us to the final missing piece: given a current ellipsoid $\mathcal{E}$ and a half-space H that does not contain its center, we want an ellipsoid $\mathcal{E}'$ that contains $\mathcal{E} \cap H$, and as small as possible. To start off, let us recall some basic facts about ellipsoids. The simplest ellipses in $\mathbb{R}^2$ are axis aligned, say with principal semi-axes having length a and b, and written as:
$$\frac{x^2}{a^2} + \frac{y^2}{b^2} \leq 1.$$
Or in matrix notation we could also say
$$\begin{bmatrix} x \\ y \end{bmatrix}^{\intercal} \begin{bmatrix} 1/a^2 & 0 \\ 0 & 1/b^2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \leq 1$$

More generally, any ellipsoid $\mathcal{E}$ is perhaps best thought of as a invertible linear transformation L applied to the unit ball $B(0,1)$, and then it being shifted to the correct center c. The linear transformation yields:

$$\begin{aligned} L(\text{Ball}(0,1)) &= \{Lx : x^\intercal x \leq 1\} \\ &= \{y : (L^{-1}y)^\intercal (L^{-1}y) \leq 1\} \\ &= \{y : y^\intercal (LL^\intercal)^{-1} y \leq 1\} \\ &= \{y : y^\intercal Q^{-1} y \leq 1\}, \end{aligned}$$

where $Q^{-1} := LL^\intercal$ is a positive semidefinite matrix. For an ellipsoid centered at c we simply write

$$\{y + 1 : y^\intercal Q^{-1} y \leq 1\} = \{y : (y-c)^\intercal Q^{-1}(y-c) \leq 1\}.$$

It is helpful to note that for any ball A,

$$\text{vol}(L(A)) = \text{vol}(A) \cdot |\det(L)| = \text{vol}(A)\ \sqrt{\det(Q)}$$

In the above problems, we are given an ellipsoid $\mathcal{E}_t$ and a half-space H_t that does not contain the center of $\mathcal{E}_t$. We want to find a matrix Q_{t+1} and a center c_{t+1} such that the resulting ellipsoid $\mathcal{E}_{t+1}$ contains $\mathcal{E}_t \cap H_t$, and satisfies

$$\frac{\text{vol}(\mathcal{E}_{t+1})}{\text{vol}(\mathcal{E}_t)} \leq e^{-1/2(n+1)}.$$

Given the above discussion, it suffices to do this when $\mathcal{E}_t$ is a unit ball: indeed, when $\mathcal{E}_t$ is a general ellipsoid, we apply the inverse linear transformation to convert it to a ball, find the smaller ellipsoid for it, and then apply the transformation to get the final smaller ellipsoid. (The volume changes due to the two transformations cancel each other out.)

We give the construction for the unit ball below, but first let us record the claim for general ellipsoids:

Theorem 18.8. *Given an ellipsoid $\mathcal{E}_t$ given by (c_t, Q_t) and a separating hyperplane $a_t^\intercal (x - c_t) \leq 0$ through its center, the new ellipsoid $\mathcal{E}_{t+1}$ with center c_{t+1} and psd matrix Q_{t+1}) is found by taking*

$$c_{t+1} := c_t - \frac{1}{n+1} h$$

and

$$Q_{t+1} = \frac{n^2}{n^2-1}\left(Q_k - \frac{2}{n+1} hh^\intercal\right)$$

where $h = \sqrt{a_t^\intercal Q_t a_t}$.

Note that the construction requires us to take square-roots: this may result in irrational numbers which we then have to either truncate, or represent implicitly. In either case, we face numerical issues; ensuring that these issues are not real problems lies at the heart of the formal analysis. We refer to the GLS book, or other textbooks for details and references.

18.5.1 Halving a Ball

Before we end, we show that the problem of finding a smaller ellipsoid that contains half a ball is, in fact, completely straight-forward. By rotational symmetry, we might as well find a small ellipsoid that contains

$$K = \text{Ball}(0,1) \cap \{x \mid x_1 \geq 0\}.$$

By symmetry, it makes sense that the center of this new ellipsoid $\mathcal{E}$ should be of the form

$$c = (c_1, 0, \dots, 0).$$

Again by symmetry, the ellipsoid can be axis-aligned, with semi-axes of length a along e_1, and $b > a$ along all the other coordinate axes. Moreover, for $\mathcal{E}$ to contain the unit ball, it should contain the points $(1,0)$ and $(0,1)$, say. So

$$\frac{(1-c_1)^2}{a^2} \leq 1 \quad \text{and} \quad \frac{c_1^2}{a^2} + \frac{1}{b^2} \leq 1.$$

Suppose these two inequalities are tight, then we get

$$a = 1 - c_1, \qquad b = \sqrt{\frac{(1-c_1)^2}{(1-c_1)^2 - c_1^2}} = \sqrt{\frac{(1-c_1)^2}{(1-2c_1}},$$

and moreover the ratio of volume of the ellipsoid to that of the ball is

$$ab^{n-1} = (1-c_1) \cdot \left(\frac{(1-c_1)^2}{1-2c_1} \right)^{(n-1)/2}.$$

This is minimized by setting $c_1 = \frac{1}{n+1}$ gives us

$$\frac{\text{vol}(\mathcal{E})}{\text{vol}(\text{Ball}(0,1))} = \dots \leq e^{-\frac{1}{2(n+1)}}.$$

For a more detailed description and proof of this process, see these notes from our LP/SDP course for details.

In fact, we can view the question of finding the minimum-volume ellipsoid that contains the half-ball K: this is a convex program, and looking at the optimality conditions for this gives us the same construction above (without having to make the assumptions of symmetry).

18.6 Algorithms for Solving LPs

While the Centroid and Ellipsoid algorithms for convex programming are powerful, giving us linear convergence, they are not typically used to solve LPs in practice. There are several other algorithms: let us mention them in passing. Let $K := \{x \mid Ax \geq b\} \subseteq \mathbb{R}^n$, and we want to minimize $\{c^\intercal x \mid x \in K\}$.

Simplex: This is perhaps the first algorithm for solving LPs that most of us see. It was also the first general-purpose linear program solver known, having been developed by George Dantzig in 1947. This is a local-search algorithm: it maintains a vertex of the polyhedron K, and at each step it moves to a neighboring vertex without decreasing the objective function value, until it reaches an optimal vertex. (The convexity of K ensures that such a sequence of steps is possible.) The strategy to choose the next vertex is called the ***pivot rule***. Unfortunately, for most known pivot rules, there are examples on which the following the pivot rule takes exponential (or at least a super-polynomial) number of steps. Despite that, it is often used in practice: e.g., the Excel software contains an implementation of simplex.

Interior Point: A very different approach to get algorithms for LPs is via interior-point algorithms: these happen to be good both in theory and in practice. The first polynomial-time interior-point algorithm was proposed by Karmarkar in 1984. We discuss this in the next chapter.

Geometric Algorithms for LPs: These approaches are geared towards solving LPs fast when the number of dimensions n is small. If m is the number of constraints, these algorithms often allow a poor runtime in n, at the expense of getting a good dependence on m. As an example, a randomized algorithm of Raimund Seidel's has a runtime of $O(m \cdot n!) = O(m \cdot n^{n/2})$; a different algorithm of Ken Clarkson (based on the multiplicative weights approach!) has a runtime of $O(n^2 m) + n^{O(n)} O(\log m)^{O(\log n)}$. One of the fastest such algorithm is by Jiri Matoušek, Micha Sharir, and Emo Welzl, and has a runtime of

$$O(n^2 m) + e^{O(\sqrt{n \log n})}.$$

For details and references, see this survey by Martin Dyer, Nimrod Megiddo, and Emo Welzl.

Naturally, there are other approaches to solve linear programs as well: write more here.

19
Interior-Point Methods

In this chapter, we continue our discussion of polynomial-time algorithms for linear programming, and cover the high-level details of an interior-point algorithm. The runtime for these linear programs has recently been improved both qualitatively and quantitatively, so this is an active area of research that you may be interested in. Moreover, these algorithms contain sophisticated general ideas (duality and the method of Lagrange multipliers, and the use of barrier functions) that are important even beyond this context.

Another advantage of delving into the details of these methods is that we can work on getting better algorithms for special kinds of linear programs of interest to us. For instance, the line of work on faster max-flow algorithms for directed graphs, starting with the work of Madry, and currently resulting in the $O(m^{4/3+\varepsilon})$-time algorithms of Kathuria, and of Liu and Sidford, are based on a better understanding of interior-point methods.

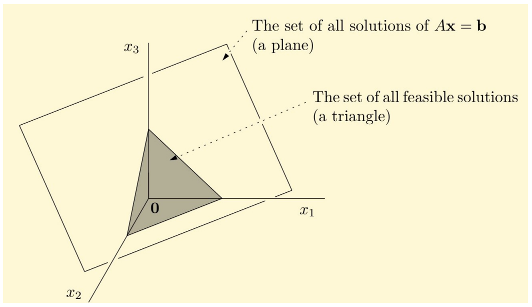

The set of all solutions of $A\mathbf{x} = \mathbf{b}$ (a plane)
The set of all feasible solutions (a triangle)
x_3
x_1
x_2
$\mathbf{0}$

Figure 19.1: The feasible region for an LP in equational form (from the Matoušek and Gärtner book).

We will consider the following LP with *equality* constraints:

$$\begin{aligned} \min\ c^\intercal x \\ Ax = b \\ x \geq 0 \end{aligned}$$

where $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$ and $c, x \in \mathbb{R}^n$. Let $K := \{x \mid Ax = b, x \geq 0\}$ be the polyhedron, and $x^* = \arg\min\{c^\intercal x \mid x \in K\}$ an optimal solution.

To get the main ideas across, we make some simplifying assumptions and skip over some portions of the algorithm. For more details, please refer to the book by Jiri Matoušek and Bernd Gärtner (which has more details), or the one by Steve Wright (which has most details).

19.1 Barrier Functions

The first step in solving the LP using an interior-point method will be to introduce a parameter $\eta > 0$ and exchange our constrained linear optimization problem for an unconstrained but *nonlinear* one:

$$f_\eta(x) := c^\intercal x + \eta\Big(\sum_{i=1}^{n} \log \frac{1}{x_i}\Big).$$

Let $x^*_\eta := \arg\min\{f_\eta(x) \mid Ax = b\}$ be the minimizer of this function over the subspace given by the equality constraints. Note that we've added in η times a ***barrier function***

$$B(x) := \sum_{i=1}^{n} \log \frac{1}{x_i}.$$

The intuition is that when x approaches the boundary $x \geq 0$ of the feasible region, the barrier function $B(x)$ will approach $+\infty$. The parameter η lets us control the influence of this barrier function. If η is sufficiently large, the contribution of the barrier function dominates in $f_\eta(x)$, and the minimizer x^*_η will be close to the "center" of the feasible region. However, as η gets close to 0, the effect of $B(x)$ will diminish and the term $c^\intercal x$ will now dominate, causing that x^*_η to approach x^*.

If we had *inequality constraints* $Ax \geq b$ as well, we would have added $\sum_{i=1}^{m} \log \frac{1}{a_i^\intercal x - b_i}$ to the barrier function.

Now consider the trajectory of the minimizer x^*_η as we lower η continuously, starting at some large value and tending to zero: this path is called the ***central path***. The idea of our ***path-following*** algorithm will be to approximately follow this path. In essence, such algorithms conceptually perform the following steps (although we will only approximate these steps in practice):

1. Pick a sufficiently large η_0 and a starting point $x^{(0)}$ that is the minimizer of $f_{\eta_0}(x)$. (We will ignore this step in our discussion, for now.)

2. At step t, move to the corresponding minimizer $x^{(t+1)}$ for $f_{\eta_{t+1}}$, where
$$\eta_{t+1} := \eta_t \cdot (1 - \epsilon).$$
Since η_t is close to η_{t+1}, we hope that the previous minimizer $x^{(t)}$ is close enough to the current goal $x^{(t+1)}$ for us to find it efficiently.

3. Repeat until η is small enough that x^*_η is very close to an optimal solution x^*. At this point, round it to get a vertex solution, like in §18.3.1.

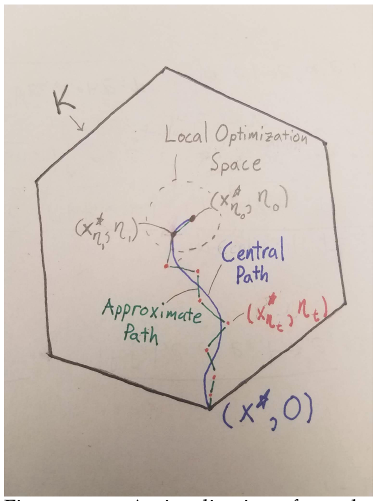


Figure 19.2: A visualization of a path-following algorithm.

We will only sketch the high-level idea behind Step 1 (finding the starting solution), and will skip Step 2 (the rounding); our focus will be on the update step. To understand this step, let us look at the structure of the minimizers for $f_\eta(x)$.

19.1.1 *The Primal and Dual LPs, and the Duality Gap*

Recall the primal linear program:

$$\begin{aligned}(P)\ \min\ & c^\intercal x \\ Ax &= b \\ x &\geq 0,\end{aligned}$$

and its dual:

$$\begin{aligned}(D)\ \max\ & b^\intercal y \\ A^\intercal y &\leq c.\end{aligned}$$

We can rewrite the dual using non-negative *slack variables* s:

$$\begin{aligned}(D')\ \max\ & b^\intercal y \\ A^\intercal y + s &= c \\ s &\geq 0.\end{aligned}$$

We assume that both the primal (P) and dual (D) are *strictly feasible*: i.e., they have solutions even if we replace the inequalities with strict ones). Then we can prove the following result, which relates the optimizer for f_η to feasible primal and dual solutions:

Lemma 19.1 (Optimality Conditions)**.** *The point $x \in \mathbb{R}^n_{\geq 0}$ is a minimizer of $f_\eta(x)$ if and only if there exist $y \in \mathbb{R}^m$ and $s \in \mathbb{R}^n_{\geq 0}$ such that:*

$$Ax - b = 0 \tag{19.1}$$
$$A^\intercal y + s = c \tag{19.2}$$
$$\forall i \in [n] : s_i x_i = \eta \tag{19.3}$$

The conditions (19.1) and (19.2) show that x and (y, s) are feasible for the primal (P) and dual (D') respectively. The condition (19.3) is an analog of the usual complementary slackness result that arises when $\eta = 0$. To prove this lemma, we use the method of Lagrange multipliers.

Observe: we get that ***if*** there exists a maximum x^*, then x^* satisfies these conditions.

Theorem 19.2 (The Method of Lagrange Multipliers)**.** *Let functions f and $g_1, \cdots, g_m$ be continuously differentiable, and defined on some open subset of $\mathbb{R}^n$. If x^* is a local optimum of the following optimization problem*

$$\begin{aligned}\min\ & f(x) \\ \text{s.t. } \forall i \in [m] : & g_i(x) = 0\end{aligned}$$

then there exists $y^* \in \mathbb{R}^m$ such that $\nabla f(x^*) = \sum_{i=1}^m y_i^* \cdot \nabla g_i(x^*)$.

Proof Sketch of Lemma 19.1. We need to show three things:

1. The function $f_\eta(x)$ achieves its maximum x^* in the feasible region.
2. The point x^* satisfies the conditions (19.1)–(19.3).
3. And that no other x satisfies these conditions.

The first step uses that if there are strictly feasible primal and dual solutions $(\hat{x}, \hat{y}, \hat{s})$, then the region $\{x \mid Ax = b, f_\mu(x) \leq f_\mu \hat{x}\}$ is bounded (and clearly closed) and hence the continuous function $f_\mu(x)$ achieves its minimum at some point x^* inside this region, by the Extreme Value theorem. (See Lemma 7.2.1 of Matoušek and Gärtner, say.)

For the second step, we use the functions $f_\mu(x)$, and $g_i(x) = a_i^\intercal x - b_i$ in Theorem 19.2 to get the existence of $y^* \in \mathbb{R}^m$ such that:

$$f_\eta(x^*) = \sum_{i=1}^m y_i^* \cdot \nabla(a_i^\intercal x^* - b_i) \quad \Longleftrightarrow \quad c - \eta \cdot \left(1/x_1^*, \cdots, 1/x_n^*\right)^\intercal = \sum_{i=1}^m y_i^* \, a_i.$$

Define a vector s^* with $s_i^* = \eta / x_i^*$. The above condition is now equivalent to setting $A^\intercal y^* + s^* = c$ and $s_i^* x_i^* = \eta$ for all i.

Finally, for the third step of the proof, the function $f_\eta(x)$ is strictly convex and has a unique local/global optimum. Finish this proof. □

By weak duality, the optimal value of the linear program lies between the values of any feasible primal and dual solution, so the ***duality gap*** $c^\intercal x - b^\intercal y$ bounds the suboptimality $c^\intercal x - OPT$ of our current solution. Lemma 19.1 allows us to relate the duality gap to η as follows.

$$c^\intercal x - b^\intercal y = c^\intercal x - (Ax)^\intercal y = x^\intercal c - x^\intercal (c - s) = x^\intercal s = n \cdot \eta.$$

If the representation size of the original LP is $L := \langle A \rangle + \langle b \rangle + \langle c \rangle$, then making $\eta \leq 2^{-\operatorname{poly}(L)}$ means we have primal and dual solutions whose values are close enough to optimal, and can be rounded (using the usual simultaneous Diophantine equations approach used for Ellipsoid).

19.2 *The Update Step*

Let us now return to the question of obtaining $x^{(t+1)}$ from $x^{(t)}$ at step t? Recall, we want $x^{(t+1)}$ to satisfy the optimality conditions (19.1)–(19.3) for $f_{\eta_{t+1}}$. The hurdles to finding this point directly are: (a) the non-negativity of the x, s variables means this is not just a linear system, there are *inequalities* to contend with. And more worryingly, (b) it is not a *linear* system at all: we have non-linearity in the constraints (19.3) because of multiplying x_i with s_i.

To get around this, we use a "local-search" method. We start with the solution $x^{(t)}$ "close to" the optimal solution $x^*_{\eta_t}$ for f_{η_t}, and take a small step, so that we remain non-negative, and also get "close to" the optimal solution $x^*_{\eta_{t+1}}$ for $f_{\eta_{t+1}}$. Then we lower η and repeat this process.

Let us make these precise. First, to avoid all the superscripts, we use (x, y, s) and η to denote $(x^{(t)}, y^{(t)}, s^{(t)})$ and η_t. Similarly, (x', y', s') and η' denote the corresponding values at time $t+1$. Now we assume we have (x, y, s) with $x, s > 0$, and also:

$$Ax = b \tag{19.4}$$

$$A^\intercal y + s = c \tag{19.5}$$

$$\sum_{i=1}^{n} \left(s_i x_i - \eta_t\right)^2 \leq ({}^{\eta_t}/_4)^2. \tag{19.6}$$

The first two are again feasibility conditions for (P) and (D'). The third condition is new, and is an approximate version of (19.3). Suppose that

$$\eta' := \eta' \cdot (1 - \frac{1}{4\sqrt{n}}).$$

Our goal is a new solution $x' = x + \Delta x$, $y' = y + \Delta y$, $s' = s + \Delta s$, which satisfies non-negativity, and ideally also satisfies the original optimality conditions (19.1)–(19.3) for the new η'. (Of course, we will fail and only satisfy the weaker condition (19.6) instead of (19.3), but we should aim high.)

Let us write the goal explicitly, by substituting (x', y', s') into (19.4)–(19.6) and using the feasibility of (x, y, s). This means the increments $\Delta x, \Delta y, \Delta s$ satisfy

$$\begin{aligned} A\,(\Delta x) &= 0 \\ A^\intercal\,(\Delta y) + (\Delta s) &= 0 \\ s_i(\Delta x_i) + (\Delta s_i)x_i + (\Delta s_i)(\Delta x_i) &= \eta' - x_i s_i. \end{aligned}$$

Note the quadratic term in blue. Since we are aiming for an approximation anyways, and these increments are meant to be tiny, we drop the quadratic term to get a *system of linear equations* in these increments:

$$\begin{aligned} A\,(\Delta x) &= 0 \\ A^\intercal\,(\Delta y) + (\Delta s) &= 0 \\ s_i(\Delta x_i) + (\Delta s_i)x_i &= \eta' - x_i s_i. \end{aligned}$$

This is often written in the following matrix notation (which I am putting down just so that you recognize it the next time you see it):

$$\begin{bmatrix} A & 0 & 0 \\ 0 & A^\intercal & I \\ \mathrm{diag}(x) & 0 & \mathrm{diag}(s) \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \eta' \mathbb{1} - x \circ s \end{bmatrix}.$$

Here $x \circ s$ stands for the component-wise product of the vectors x and s. The bottom line: this is a linear system and we can solve it, say using Gaussian elimination. Now we can set $x' = x + \Delta x$, etc., to get the new point (x', y', s'). It remains to check the non-negativity and also the weakened conditions (19.4)–(19.6) with respect to η'.

The goal of many modern algorithms is to get faster ways to solve this linear system. E.g., if it were a Laplacian system we could (approximately) solve it in near-linear time.

19.2.1 *Properties of the New Solution*

While discarding the quadratic terms means we do not satisfy $x_i s_i = \eta$ for each coordinate i, we can show that we satisfy it on average, allowing us to bound the duality gap.

Lemma 19.3. *The new duality gap is $\langle x', s' \rangle = n\,\eta'$.*

Proof. The last set of equalities in the linear system ensure that

$$s_i x_i + s_i(\Delta x_i) + (\Delta s_i) x_i = \eta', \tag{19.7}$$

so we get

$$\begin{aligned} \langle x', s' \rangle &= \langle x + \Delta x, s + \Delta s \rangle \\ &= \sum_i \big(s_i x_i + s_i(\Delta x_i) + (\Delta s_i) x_i \big) + \langle \Delta x, \Delta s \rangle \\ &= n\eta' + \langle \Delta x, -A^\intercal(\Delta y) \rangle \\ &= n \cdot \eta' - \langle A(\Delta x), \Delta y \rangle \\ &= n \cdot \eta', \end{aligned}$$

using the first two equality constraints of the linear system. □

We explicitly maintained the invariants given by (19.4), (19.5), so it remains to check (19.6). This requires just a bit of algebra (that also causes the $\sqrt{n}$ to pop out).

Lemma 19.4. $\sum_{i=1}^n \left(s'_i x'_i - \eta' \right)^2 \leq (\eta'/4)^2$.

Proof. As in the proof of Lemma 19.3, we get that $s'_i x'_i - \eta' = (\Delta s_i)(\Delta x_i)$, so it suffices to show that

$$\sqrt{\sum_{i=1}^n (\Delta s_i)^2 (\Delta x_i)^2} \leq \eta'/4.$$

We can use the inequality

$$\sqrt{\sum_i a_i^2 b_i^2} \leq \frac{1}{4} \sum_i (a_i + b_i)^2 = \frac{1}{4} \sum_i (a_i^2 + b_i^2 + 2 a_i b_i),$$

where we set $a_i^2 = \frac{x_i(\Delta s_i)^2}{s_i}$ and $b_i^2 = \frac{s_i(\Delta x_i)^2}{x_i}$. Hence

$$\begin{aligned}
\sqrt{\sum_{i=1}^{n}(\Delta s_i \Delta x_i)^2} &\leq \frac{1}{4}\sum_{i=1}^{n}\left(\frac{x_i}{s_i}\cdot(\Delta s_i)^2 + \frac{s_i}{x_i}\cdot(\Delta x_i)^2 + 2(\Delta s_i)(\Delta x_i)\right) \\
&= \frac{1}{4}\sum_{i=1}^{n}\frac{(x_i\Delta s_i)^2 + (s_i\Delta x_i)^2}{s_i x_i} \qquad [\text{since } (\Delta s)^\intercal \Delta x = 0 \text{ by Claim 19.3}] \\
&\leq \frac{1}{4}\frac{\sum_{i=1}^{n}(x_i\Delta s_i + s_i\Delta x_i)^2}{\min_{i\in[n]} s_i x_i} \\
&= \frac{1}{4}\frac{\sum_{i=1}^{n}(\eta' - s_i x_i)^2}{\min_{i\in[n]} s_i x_i}. \qquad (19.8)
\end{aligned}$$

We now bound the numerator and denominator separately.

Claim 19.5 (Denominator). $\min_i s_i x_i \geq \eta\left(1 - \frac{1}{4\sqrt{n}}\right)$.

Proof. By the inductive hypothesis, $\sum_i (s_i x_i - \eta)^2 \leq (\eta/4)^2$. This means that $\max_i |s_i x_i - \eta| \leq \frac{\eta}{4\sqrt{n}}$, which proves the claim. □

Claim 19.6 (Numerator). $\sum_{i=1}^{n}(\eta' - s_i x_i)^2 \leq \eta^2/8$.

Proof. Let $\delta = \frac{1}{4\sqrt{n}}$. Then,

$$\begin{aligned}
\sum_{i=1}^{n}(\eta' - s_i x_i)^2 &= \sum_{i=1}^{n}((1-\delta)\eta - s_i x_i)^2 \\
&= \sum_{i=1}^{n}(\eta - s_i x_i)^2 + \sum_{i=1}^{n}(\delta\eta)^2 + 2\delta\eta \sum_{i=1}^{n}(\eta - s_i x_i).
\end{aligned}$$

The first term is at most $(\eta/4)^2$, by the induction hypothesis. On the other hand, by Claim 19.3 we have

$$\sum_{i=1}^{n}(\eta - s_i x_i) = n\eta - \sum_{i=1}^{n} s_i x_i = 0.$$

Thus

$$\sum_{i=1}^{n}(\eta' - s_i x_i)^2 \leq (\eta/4)^2 + n\frac{1}{(4\sqrt{n})^2}\eta^2 = \eta^2/8. \qquad \square$$

Substituting these results into (19.8), we get

$$\sqrt{\sum_{i=1}^{n}(\Delta s_i \Delta x_i)^2} \leq \frac{1}{4}\frac{\eta^2/8}{(1-\frac{1}{4\sqrt{n}})\eta} = \frac{1}{32}\frac{\eta'}{(1-\frac{1}{4\sqrt{n}})^2}.$$

This expression is smaller than $\eta'/4$, which completes the proof. □

Lemma 19.7. *The new values x', s' are non-negative.*

Proof. By induction, we assume the previous point has $x_i > 0$ and $s_i > 0$. (For the base case we need to ensure that the starting solution $(x^{(0)}, s^{(0)})$ also satisfies this property.) Now for a scalar $\alpha \in [0,1]$ we define $x'' := x + \alpha\Delta x$, $s'' := s + \alpha\Delta s$, and $\eta'' := (1-\alpha)\eta + \alpha\eta'$, to linearly interpolate between the old values and the new ones. Then we can show $\langle x'', s''\rangle = n\,\eta''$, and also

$$\sum_i (s_i'' x_i'' - \eta'')^2 \le (\eta''/4)^2, \tag{19.9}$$

which are analogs of Lemmas 19.3 and 19.4 respectively. The latter inequality means that $|s_i''x_i'' - \eta''| \le \eta''/4$ for each coordinate i, else that coordinate itself would violate inequality (19.9). Specifically, this means that neither x_i'' nor s_i'' ever becomes zero for any value of $\alpha \in [0,1]$. Now since (x_i'', s_i'') is a linear interpolation between (x_i, s_i) and (x_i', s_i'), and the former were strictly positive, the latter cannot be non-positive. ☐

Theorem 19.8. *Given an LP* $\min\{c^\intercal x \mid Ax = b, x \ge 0\}$ *with an initial feasible* $(x^{(0)}, \eta_0)$ *pair, the interior-point algorithm produces a primal-dual pair with duality gap at most* ε *in* $O(\sqrt{n}\log\frac{n\eta_0}{\varepsilon})$ *iterations, each involving solving one linear system.*

The proof of the above theorem follows immediately from the fact that the duality gap at the beginning is $n\eta_0$, and the value of η drops by $(1 - \frac{1}{4\sqrt{n}})$ in each iteration. If the LP has representation size $L := \langle A\rangle + \langle b\rangle + \langle c\rangle$, we can stop when $\varepsilon = \exp(-\text{poly}(L))$, and then round this solution to an vertex solution of the LP.

The one missing piece is finding the initial $(x^{(0)}, \eta_0)$ pair: this is a somewhat non-trivial step. One possible approach is to run the interior-point algorithm "in reverse". The idea is that we can start with some vertex of the feasible region, and then to successively *increase* η through a similar mechanism as the one above, until the value of η is sufficiently large to begin the algorithm.

19.3 The Newton-Raphson Method

A more "modern" way of viewing interior-point methods is via the notion of *self-concordance*. To do this, let us revisit the classic Newton-Raphson method for finding roots.

19.3.1 Finding Zeros of Functions

The basic Newton-Raphson method for finding a zero of a univariate function is the following: given a function g, we start with a point x_1, and at each time t, set

$$x_{t+1} \leftarrow x_t - \frac{g(x_t)}{g'(x_t)}. \tag{19.10}$$

We now show that if f is "nice enough" and we are "close enough" to a zero x^*, then this process converges very rapidly to x^*.

Since we take $O(\log\log 1/\varepsilon)$ iterations to get to error ε, the number of bits of accuracy squares each time. This is called ***quadratic convergence*** in the optimization literature.

Theorem 19.9. *Suppose g has continuous second-derivatives, then if x^* is a zero of g, then if we start at x_1 "close enough" to X^*, the error goes to ε in $O(\log\log 1/\varepsilon)$ steps. Make this formal!*

Proof. By Taylor's theorem, the existence of continuous second derivatives means we can approximate f around x_t as:

$$f(x^*) = f(x_t) + f'(x_t)(x^* - x_t) + {}^1\!/\!_2 f''(\xi_t)(x^* - x_t)^2,$$

where ξ_t is some point in the interval $[x^*, x_t]$. However, x^* is a zero of f, so $f(x^*) = 0$. Moreover, using (19.10) to replace $x_t f'(x_t) - f(x_t)$ by $x_{t+1} f'(x_t)$, and rearranging, we get

$$\underbrace{x^* - x_{t+1}}_{=:\delta_{t+1}} = \frac{-f''(\xi_t)}{2\,f'(x_t)} \cdot \underbrace{(x^* - x_t)^2}_{=:\delta_t^2}.$$

Above, we use δ_t to denote the error $x^* - x_t$. Taking absolute values

$$|\delta_{t+1}| = \left|\frac{f''(\xi_t)}{2\,f'(x_t)}\right| \cdot \delta_t^2.$$

Hence, if we can ensure that $|\frac{f''(\xi)}{2\,f'(x)}| \leq M$ for each x and each ξ that lies bewteen x^* and x, then once we have δ_0 small enough, then each subsequent error drops quadratically. This means the number of significant bits of accuracy double each step. More careful analysis. ∎

19.3.2 An Example

Given an n-bit integer $a \in \mathbb{Z}$, suppose we want to compute its reciprocal ${}^1\!/\!_a$ without using divisions. This reciprocal is a zero of the expression

$$g(x) = {}^1\!/\!_x - a.$$

This method for computing reciprocals appears in the classic book of Aho, Hopcroft, Ullman, without any elaboration—it always mystified me until I realized the connection to the Newton-Raphson method. I guess they expected their readers to be familiar with these connections, since computer science used to have closer connections to numerical analysis in the 1970s.

Hence, the Newton-Raphson method says, we can start with $x_1 = 1$, say, and then use (19.10) to get

$$x_{t+1} \leftarrow x_t - \frac{({}^1\!/\!_{x_t} - a)}{(-{}^1\!/\!_{x_t^2})} = x_t + x_t(1 - a\,x_t) = 2x_t - a\,x_t^2.$$

If we define $\varepsilon_t := 1 - a\,x_t$, then

$$\varepsilon_{t+1} = 1 - a\,x_{t+1} = 1 - (2a\,x_t - a^2\,x_t^2) = (1 - a\,x_t)^2 = \varepsilon_t^2.$$

Hence, if $\varepsilon_1 \leq {}^1\!/\!_2$, say, the number of bits of accuracy double at each step. Moreover, if we are careful, we can store x_t using integers (by instead keeping track of $2^k x_t$ for suitably chosen values $k \approx 2^t$).

19.3.3 *Minimizing Convex Functions*

To find the minimum of a function f (especially a convex function) we can focus on finding a stationary point, i.e., a point x such that $f'(x) = 0$. Setting $g = f'$, the update rule just changes to

$$x_{t+1} \leftarrow x_t - \frac{f'(x_t)}{f''(x_t)}. \tag{19.11}$$

19.3.4 *On To Higher Dimensions*

For general functions $f : \mathbb{R}^n \to \mathbb{R}$, the rule remains the same, with the natural changes:

$$x_{t+1} \leftarrow x_t - [H_f(x_t)]^{-1} \cdot \nabla f(x_t). \tag{19.12}$$

19.4 *Self-Concordance*

Analogy between self-concordance and the convergence conditions for the 1-d case?

Present the view using the "modern view" of self-concordance. Mention that the current bound is really $O(m)$-self-concordant. That universal barrier is $O(n)$ self-concordant, but not efficient. Vaidya's volumetric barrier? The entropic barrier? The Lee-Sidford barrier, based on leverage scores. What's the cleanest way, without getting lost in the algebra?

Part IV

Combating Intractability

20
Approximation Algorithms

In this chapter, we turn to the problem of combating intractability: many combinatorial optimization problems are **NP**-hard, and hence are unlikely to have polynomial-time algorithms. Hence we consider approximation algorithms: algorithms that run in polynomial-time, but output solutions whose quality is close the optimal solution's quality. We illustrate some of the basic ideas in the context of two **NP**-hard problems: Set Cover and Bin Packing. Both have been studied since the 1970s.

Let us start with some definitions: having fixed an optimization problem, let I denote an instance of the problem, and Alg denote an algorithm. Then $\text{Alg}(I)$ is the output/solution produced by the algorithm, and $c(\text{Alg}(I))$ its cost. Similarly, let $\text{Opt}(I)$ denote the optimal output for input I, and let $c(\text{Opt}(I))$ denote its cost. For minimization problems, the ***approximation ratio*** of the algorithm $\mathcal{A}$ is defined to be the worst-case ratio between the costs of the algorithm's solution and the optimum:

Typically, if the instance is clear from context, we use the notation

$$\text{Alg} \leq r \cdot \text{Opt}$$

to denote that

$$c(\text{Alg}(I)) \leq r \cdot c(\text{Opt}(I)).$$

$$\rho = \rho_{\mathcal{A}} := \max_I \frac{c(\text{Alg}(I))}{c(\text{Opt}(I))}.$$

In this case, we say that Alg is an ρ-approximation algorithm. For *maximization* problems, we define *rho* to be

$$\rho = \rho_{\mathcal{A}} := \min_I \frac{c(\text{Alg}(I))}{c(\text{Opt}(I))},$$

and therefore a number in $[0, 1]$.

20.1 A Rough Classification into Hardness Classes

In the late 1990s, there was an attempt to classify combinatorial optimization problems into a small number of hardness classes: while this ultimately failed, a rough classification of **NP**-bard problems is still useful.

- *Fully Poly-Time Approximation Scheme (FPTAS):* For problems in this category, there exist approximation algorithms that take in a parameter ε, and output a solution with approximation ratio $1+\varepsilon$ in time $\text{poly}(\langle I \rangle, 1/\varepsilon)$. E.g., one such problem is KNAPSACK, where given a collection of n items, with each item i having size $s_i \in \mathbb{Q}_+$ and value $v_i \in \mathbb{Q}_+$, find the subset of items with maximum value that fit into a knapsack of unit size.

As always, let $\langle I \rangle$ denote the bit complexity of the input I.

- *Poly-Time Approximation Scheme (PTAS):* For a problem in this category, for any $\varepsilon > 0$, there exists an approximation algorithm with approximation ratio $1+\varepsilon$, that runs in time $O(n^{f(\varepsilon)})$ for some function $f(\cdot)$. For instance, the TRAVELING SALESMAN PROBLEM in d-dimensional Euclidean space has an algorithm due to Sanjeev Arora (1996) that computes a $(1+\varepsilon)$-approximation in time $O(n^{f(\varepsilon)})$, where $f(\varepsilon) = \exp\{(1/\varepsilon)^d\}$. Moreover, it is known that this dependence on ε, with the doubly-exponential dependence on d, is unavoidable.

The runtime has been improved to $O(n \log n + n \exp\{(1/\varepsilon)^d\})$.

- *Constant-Factor Approximation:* Examples in this class include the TRAVELING SALESMAN PROBLEM on general metrics. In the late 1970s, Nicos Christofides and Anatoliy Serdyukov discovered the same 1.5-approximation algorithm for metric TSP, using the Blossom algorithm to connect up the odd-degree vertices of an MST of the metric space to get an Eulerian spanning subgraph, and hence a TSP tour. This was improved only in 2020, when Anna Karlin, Nathan Klein, and Shayan Oveis-Gharan gave an $(1.5-\varepsilon)$-approximation, which we hope to briefly outline in a later chapter. Meanwhile, it has been shown that metric TSP can't be approximated with a ratio better than $\frac{123}{122}$ under the assumption of $\mathbf{P} \neq \mathbf{NP}$, by Karpinski, Lampis, Schmied.

Christofides' result only ever appeared as a CMU GSIA technical report in 1976. Serdyukov's result only came to be known a couple years back.

Karlin, Klein, and Oveis Gharan (2020)

- *Logarithmic Approximation:* An example of this is SET COVER, which we will discuss in some detail.

- *Polynomial Approximation:* One example is the INDEPENDENT SET problem, for which any algorithm with an approximation ratio $n^{1-\varepsilon}$ for some constant $\varepsilon > 0$ implies that $\mathbf{P} = \mathbf{NP}$. The best approximation algorithm for INDEPENDENT SET known has an approximation ratio of $O(n/\log^3 n)$.

However, there are problems that do not fall into any of these clean categories, such as ASYMMETRIC k-CENTER, for which there exists a $O(\log^* n)$-approximation algorithm, and this is best possible unless $\mathbf{P} = \mathbf{NP}$. Or GROUP STEINER TREE, where the approximation ratio is $O(\log^2 n)$ on trees, and this is also best possible.

20.2 The Surrogate

Given that it is difficult to find an optimal solution, how can we argue that the output of some algorithm has cost comparable to that of $\mathrm{Opt}(I)$. An important idea in proving the approximation guarantee involves the use of a ***surrogate***, or a ***lower bound***, as follows: Given an algorithm Alg and an instance I, if we want to calculate the approximation ratio of Alg, we first find a *surrogate map* S from instances to the reals. To bound the approximation ratio, we typically do the following:

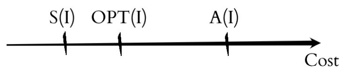


Figure 20.1: The cost diagram on instance I (costs increase from left to right).

1. We show that $S(I) \leq \mathrm{Opt}(I)$ for all I, and

2. then show that $\mathrm{Alg}(I) \leq \alpha S(I)$ for all I.

This shows that $\mathrm{Alg}(I) \leq \alpha \, \mathrm{Opt}(I)$. Which leaves us with the question of how to construct the surrogate. Sometimes we use the combinatorial properties of the problem to get a surrogate, and at other times we use a linear programming relaxation.

20.3 The Set Cover Problem

In the SET COVER problem, we are given a universe U with n elements, and a family $\mathcal{S} = \{S_1, \ldots, S_m\}$ of m subsets of U, such that $U = \cup_{S \in \mathcal{S}} S$. We want to find a subset $\mathcal{S}' \subseteq \mathcal{S}$, such that $U = \cup_{S \in \mathcal{S}} S$ while minimizing the size $|\mathcal{S}'|$.

In the weighted version of SET COVER, we have a cost c_S for each set $S \in \mathcal{S}$, and want to minimize $c(\mathcal{S}') = \sum_{S \in \mathcal{S}'} c_S$. We will focus on the unweighted version for now, and indicate the changes to the algorithm and analysis to extend the results to the weighted case.

The SET COVER problem is **NP**-complete, even for the unweighted version. Several approximation algorithms are known: the greedy algorithm is a $\ln n$-approximation algorithm, with different analyses given by Vašek Chvátal, David Johnson, Laci Lovász, Stein, and others. Since then, the same approximation guarantee was given based on the relax-and-round paradigm.

This was complemented by a hardness result in 1998 by Uri Feige (building on previous work of Carsten Lund and Mihalis Yannakakis), who showed that a $(1 - \varepsilon) \ln n$-approximation algorithm for any constant $\varepsilon > 0$ would imply that NP has algorithms that run in time $O(n^{\log \log n})$. This was improved by Irit Dinur and David Steurer, who tightened the result to show that such an approximation algorithm would in fact imply that NP has polynomial-time algorithm (i.e., that $\mathbf{P} = \mathbf{NP}$).

20.3.1 The Greedy Algorithm for Set Cover

The greedy algorithm is simple: *Repeatedly pick the set $S \in \mathcal{S}$ that covers the most uncovered elements, until all elements of U are covered.*

Theorem 20.1. *The greedy algorithm is a $\ln n$-approximation.*

The greedy algorithm does not achieve a better ratio than $\Omega(\log n)$: one example is given by the figure to the right. The optimal sets are the two rows, whereas the greedy algorithm may break ties poorly and pick the set covering the left half, and then half the remainder, etc. A more sophisticated example can show a matching gap of $\ln n$.

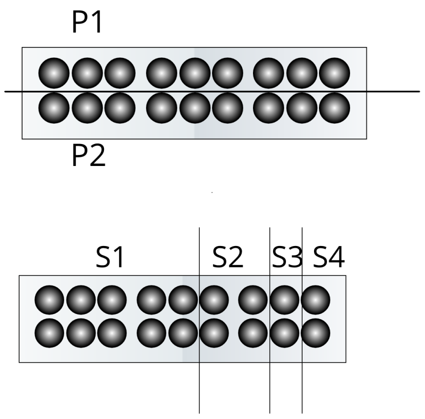


Figure 20.2:

Proof of Theorem 20.1. Suppose Opt picks k sets from $\mathcal{S}$. Let n_i be the number of still uncovered when the algorithm has picked i sets. Then $n_0 = n = |U|$. Since the k sets in Opt cover all the elements of U, they also cover the uncovered elements in n_i. By averaging, there must exist a set in $\mathcal{S}$ that covers n_i/k of the yet-uncovered elements. Hence,

$$n_{i+1} \le n_i - n_i/k = n_i(1 - 1/k).$$

Iterating, we get $n_t \le n_0(1 - 1/k)^t < n \cdot e^{-t/k}$. So setting $T = k \ln n$, we get $n_T < 1$. Since n_T must be an integer, it is zero, so we have covered all elements using $T = k \ln n$ sets. □

As always, we use $1 + x \le e^x$, and here we can use that the inequality is strict whenever $x \neq 0$.

If the sets are of size at most B, we can show that the greedy algorithm is an $(1 + \ln B)$-approximation. Moreover, for the weighted case, the greedy algorithm changes to picking the set S in that maximizes:

$$\frac{\text{number of yet-uncovered elements in } S}{c_S}.$$

One can give an analysis like the one above for this weighted case as well. Not quite, the proof here changes a fair bit, need to rephrase and give the proof?

20.4 A Relax-and-Round Algorithm for Set Cover

The second algorithm for Set Cover uses the popular relax-and-round framework. The steps of this process are as follows:

1. Write an integer linear program for the problem. This will also be **NP**-hard to solve, naturally.

2. Relax the integrality constraints to get a linear program. Since this is a minimization problem, relaxing the constraints causes the optimal LP value to be no larger than the optimal IP value (which is just Opt). This optimal value LP value is the ***surrogate***.

3. Now solve the linear program, and round the fractional variables to integer values, while ensuring that the cost of this integer solution is not much higher than the LP value.

Let's see this in action: here is the integer linear program (ILP) that precisely models SET COVER:

$$\begin{aligned} \min \sum_{S \in \mathcal{S}} c_S x_S & & \text{(ILP-SC)} \\ s.t. \sum_{S: e \in S} x_S \geq 1 & \quad \forall e \in U & \\ x_S \in \{0,1\} & \quad \forall S \in \mathcal{S}. & \end{aligned}$$

The LP relaxation just drops the integrality constraints:

$$\begin{aligned} \min \sum_{S \in \mathcal{S}} c_S x_S & & \text{(LP-SC)} \\ s.t. \sum_{S: e \in S} x_S \geq 1 & \quad \forall e \in U & \\ x_S \geq 0 & \quad \forall S \in \mathcal{S}. & \end{aligned}$$

If $\text{LP}(I)$ is the optimal value for the linear program, then we get:

$$\text{LP}(I) \leq \text{Opt}(I).$$

Finally, how do we round? Suppose x^* is the fractional solution obtained by solving the LP optimally. We do the following two phases:

1. *Phase 1:* Repeat $t = \ln n$ times: for each set S, pick S with probability x_S^* independently.

2. *Phase 2:* For each element e yet uncovered, pick any set covering it.

Clearly the solution produced by the algorithm is feasible; it just remains to bound the number of sets picked by it.

Theorem 20.2. *The expected number of sets picked by this algorithm is* $(\ln n)\ LP(I) + 1$.

Proof. Clearly, the expected number of sets covered in each round in phase 1 is $\sum_S x_S^* = \text{LP}(I)$, and hence the expected number of sets in phase 1 is at most $\ln n$ times as much.

For the second phase, the number of sets not picked is precisely the the expected number of elements ***not covered*** in Phase 1. To calculate this, consider an arbitrary element e.

$$\begin{aligned} \Pr[e \text{ not covered in phase 1}] &= (\Pi_{S: e \in S}(1 - x_S^*))^t \\ &\leq (e^{-\sum_{S:e \in S} x_S})^t \\ &\leq (e^{-1})^t \\ &= \frac{1}{n}, \end{aligned}$$

since $t = \ln n$. By linearity of expectations, the expected number of uncovered elements in Phase 2 should be 1, so in expectation we'll pick 1 set in Phase 2. This completes the proof. □

One can derandomize this algorithm to get a deterministic algorithm with the same guarantee. We may see this in an exercise. Also, Neal Young has a way to solve this problem without solving the LP at all!

In a homework problem, we will show that if the sizes of the sets are bounded by B, then we can get a $(1 + \ln B)$-approximation as well. And that the analysis can extend to the weighted case, where sets have costs.

20.5 The Bin Packing Problem

Bin Packing is another classic **NP**-hard optimization problem. We are given$n$ items, each item i having some size $s_i \in [0, 1]$. We want to find the minimum number of bins, each with capacity 1, such that we can pack all n items into them. Formally, we want to find the partition of $[n]$ into $S_1 \cup S_2 \cup \ldots \cup S_k$ such that $\sum_{i \in S_j} s_i \leq 1$ for each set S_j, and moreover, the number of parts k is minimized.

The Bin Packing is **NP**-hard, and this can be shown via a reduction from the Partition problem (where we are given n positive integers $s_1, s_2, \ldots, s_n$ and an integer K such that $\sum_{i=1}^{n} s_i = 2K$, and we want to decide whether we can find a partition of these integers into two disjoint sets A, B such that $\sum_{i \in A} s_i = \sum_{j \in B} s_j = K$). Since this partition instance corresponds gives us Bin Packing instances where the optimum is either 2 or at least 3, the reduction shows that getting an approximation factor of smaller than 3/2 for Bin Packing is also **NP**-hard.

We show two algorithms for this problem. The first algorithm First-Fit uses at most 2 Opt bins, whereas the second algorithm uses at most $(1 + \varepsilon)$ Opt $+O(1/\varepsilon^2)$ bins. These are not the best results possible: e.g., a recent result by Rebecca Hoberg and Thomas Rothvoß gives a solution using at most Opt $+O(\log \text{Opt})$ bins, and it is conceivable that we can get an algorithm that uses Opt $+O(1)$ bins.

20.5.1 A Class of Greedy Algorithms: X-Fit

We can define a collection of greedy algorithms that consider the items in some arbitrary order: for each item they try to fit it into some "open" bins; if the item does not fit into any of these bins, then they open a new bin and put it there. Here are some of these algorithms:

1. First-Fit: add the item to the earliest opened bin where it fits.

2. Next-Fit: add the item to the single most-recently opened bin.

3. Best-Fit: consider the bins in increasing order of free space, and add the item to the first one that can take it.

4. Worst-Fit: consider the open bins in *decreasing*(!) order of free space, and add the item to the first one that can take it. The idea is to ensure that no bin has small amounts of free space remaining, which is likely to then get wasted.

All these algorithms are 2-approximations. Let us give a proof for First-Fit, the others have similar proofs.

Theorem 20.3. $\text{Alg}_{FF}(I) \leq 2 \cdot \text{Opt}(I)$.

Proof. The surrogate in this case is the total volume $V(I) = \sum_i s_i$ of items. Clearly, $\lceil V(I) \rceil \leq OPT(I)$. Now consider the bins in the order they were opened. For any pair of consecutive bins $2j-1, 2j$, the first item in bin $2j$ could not have fit into bin $2j-1$ (else we would not have opened the new bin). So the total size of items in these two consecutive bins is *strictly* more than 1.

Another way to say this: at most one bin is at-most-half-full, because if there were two, the later of these bins would not have been opened.

Hence, if we open K bins, the total volume of items in these bins is strictly more than $\lfloor K/2 \rfloor$. Hence,

$$\lfloor K/2 \rfloor < V(I) \implies K \leq 2 \lceil V(I) \rceil \leq 2 \,\text{Opt}(I). \qquad \square$$

Exercise: if all the items were of size at most ε, then each bin (except the last one) would have at least $1-\varepsilon$ total size, thereby giving an approximation of

$$\frac{1}{1-\varepsilon}\,\text{Opt}(I) + 1 \approx (1+\varepsilon)\,\text{Opt}(I) + 1.$$

20.6 The Linear Grouping Algorithm for Bin Packing

The next algorithm was given by Wenceslas Fernandez de la Vega and G.S. Luecker, and it uses a clever linear programming idea to get an "almost-PTAS" for Bin Packing. Observe that we cannot hope to get a PTAS, because of the hardness result we showed above. But we will show that if we allow ourselves a small additive term, the hardness goes away. The main ideas here will be the following:

Recall, a PTAS (*polynomial-time approximation scheme*) is an algorithm that for any $\varepsilon > 0$ outputs a $(1+\varepsilon)$-approximation in time $n^{f(\varepsilon)}$. Hence, we can get the approximation factor to any constant above 1 as we want, and still get polynomial-time—just the degree of the polynomial in the runtime gets larger.

1. We can discretize the item sizes down to a constant number of values by losing at most $\varepsilon\,\text{Opt}$ (where the constant depends on ε).

2. The problem for a constant number of sizes can be solved almost exactly (up to an additive constant) in polynomial-time.

3. Items of size at most ε can be added to any instance while maintaining an approximation factor of $(1+\varepsilon)$.

20.6.1 *The Linear Grouping Procedure*

Lemma 20.4 (Linear Grouping). *Given an instance $I = (s_1, s_2, \dots, s_n)$ of* BIN PACKING, *and a parameter $D \in \mathbb{N}$, we can efficiently produce another instance $I' = ({s_1}', {s_2}', \dots, {s_n}')$ with increased sizes ${s_i}' > s_i$ and at most D distinct item sizes, such that*

$$\text{Opt}(I') \leq \text{Opt}(I) + \lceil n/D \rceil .$$

Proof. The instance I' is constructed as follows:

- Sort the sizes s_i in non-increasing order to get $s_1 \geq s_2 \geq \dots \geq s_n$.
- Group items into D groups of $\lceil n/D \rceil$ consecutive items, with the last group being potentially slightly smaller.
- Define the new size ${s_i}'$ for each item i to be the size of the largest element in i's group.

There are D distinct item sizes, and all sizes are only increased, so it remains to show a packing for the items in I' that uses at most $\text{Opt}(I) + \lceil n/D \rceil$ bins. Indeed, suppose $\text{Opt}(I)$ assigns item i to some bin b. Then we assign item $(i + \lceil n/D \rceil)$ to bin b. Since the sizes of the items only get smaller, this allocates all the items except items in first group, without violating the sizes. Now we assign each item in the first group into a new bin, thereby opening up $\lceil n/D \rceil$ more bins. □

20.6.2 *An Algorithm for a Constant Number of Item Sizes*

Suppose we have an instance with at most D distinct item sizes: let the sizes be $s_1 < s_2 < \dots < S_D$, with $\delta > 0$ being the smallest size. The instance is then defined by the number of items for each size. Define a ***configuration*** to be a collection of items that fits into a bin: there can be at most $1/s_1$ items in any bin, and each item has one of D sizes (or it can be the "null" item), so there are at most $N := (D+1)^{1/s_1}$ different configurations. Note that if D and s_1 are both constants, this is (large) constant. (In the next section, we use this result for the case where $s_1 \geq \varepsilon$.) Let $\mathcal{C}$ be the collection of all configurations.

We now use an integer LP due to Paul Gilmore and Ralph Gomory (from the 1950s). It has one variable x_C for every configuration $C \in \mathcal{C}$ that denotes the number of bins with configuration C in the solution. The LP is:

$$\begin{aligned} \min \sum_{C \in \mathcal{C}} x_C, & \\ s.t. \sum_{C} A_{Cs} x_C \geq n_s, & \qquad \forall \text{ sizes } s \\ x_C \in \mathbb{N}. & \end{aligned}$$

Here A_{Cs} is the number of items of type s being placed in the configuration C, and n_s is the total number items of size s in the instance. This is an exact formulation, and relaxing the integrality constraint to $x_C \geq 0$ gives us an LP that we can solve in time $\text{poly}(N, n)$. This is polynomial time when N is a constant. We use the optimal value of this LP as our surrogate.

In fact, we show in a homework problem that the LP can be solved in time polynomial in n even when N is not a constant.

How do we round the optimal solution for this LP? There are only D non-trivial constraints in the LP, and N non-negativity constraints. So if we pick an optimal vertex solution, it must have some N of these constraints at equality. This means at least $N - D$ of these tight constraints come from the latter set, and therefore $N - D$ variables are set to zero. In other words, at most D of the variables are non-zero. Rounding these variables up to the closest integer, we get a solution that uses at most $LP(I) + D \leq \text{Opt}(I) + D$ bins. Since D is a constant, we have approximated the solution up to a constant.

20.6.3 The Final Bin Packing Algorithm

Combining the two ideas, we get a solution that uses

$$\text{Opt}(I) + \lceil n/D \rceil + D$$

bins. Now if we could ensure that n/D were at most $\varepsilon\,\text{Opt}(I)$, when D was $f(\varepsilon)$, we would be done. Indeed, if all the items have size at least ε, the total volume (and therefore $\text{Opt}(I)$) is at least εn. If we now set $D = \lceil 1/\varepsilon^2 \rceil$, then $n/D \leq \varepsilon^2 n \leq \varepsilon\,\text{Opt}(I)$, and the number of bins is at most

$$(1+\varepsilon)\,\text{Opt}(I) + \left\lceil 1/\varepsilon^2 \right\rceil.$$

What if some of the items are smaller than ε? We now use the observation that First-Fit behaves very well when the item sizes are small. Indeed, we first hold back all the items smaller than ε, and solve the remaining instance as above. Then we add in the small items using First-Fit: if it does not open any new bins, we are fine. And if adding these small items results in opening some new bin, then each of the existing bins—and all the newly opened bins (except the last one)—must have at least $(1 - \varepsilon)$ total size in them. The number of bins is then at most

$$\frac{1}{1-\varepsilon}\,\text{Opt}(I) + 1 \approx (1 + O(\varepsilon))\,\text{Opt}(I) + 1,$$

as long as $\varepsilon \leq 1/2$.

20.7 Subsequent Results and Open Problems

21
Approximation Algorithms via SDPs

Just like the use of linear programming was a major advance in the design of approximation algorithms, specifically in the use of linear programs in the relax-and-round framework, another significant advantage was the use of semidefinite programs in the same framework. For instance, the approximation guaranteee for the MAX-CUT problem was improved from 1/2 to 0.878 using this technique. Moreover, subsequent results have shown that any improvements to this approximation guarantee in polynomial-time would disprove the Unique Games Conjecture.

21.1 Positive Semidefinite Matrices

The main objects of interest in semidefinite programming, not surprisingly, are positive semidefinite matrices.

Definition 21.1 (Positive Semidefinite Matrices)**.** Let $A \in \mathbb{R}^{n\times n}$ be a real-valued symmetric matrix and let $r = \text{rank}(A)$. We say that A is *positive semidefinite (PSD)* if any of the following equivalent conditions hold:

a. $x^\intercal A x \geq 0$ for all $x \in \mathbb{R}^n$.

b. All of A's eigenvalues are nonnegative (with r of them being strictly positive), and hence $A = \sum_{i=1}^{r} \lambda_i v_i v_i^\intercal$ for $\lambda_1, \ldots, \lambda_r > 0$, and v_i's being orthonormal.

c. There exists a matrix $B \in \mathbb{R}^{n\times r}$ such that $A = BB^\intercal$.

d. There exist vectors $v_1, \ldots, v_n \in \mathbb{R}^r$ such that $A_{i,j} = \langle v_i, v_j\rangle$ for all i, j.

e. There exist jointly distributed (real-valued) random variables $X_1, \ldots, X_n$ such that $A_{i,j} = \mathbb{E}[X_i X_j]$.

f. All principal minors have nonnegative determinants.

A principal minor is a submatrix of A obtained by taking the columns and rows indexed by some subset $I \subseteq [n]$.

The different definitions may be useful in different contexts. As an example, we see that the condition in Definition 21.1(f) gives a short proof of the following claim.

Lemma 21.2. *Let $A \succeq 0$. If $A_{i,i} = 0$ then $A_{j,i} = A_{i,j} = 0$ for all j.*

We will write $A \succeq 0$ to denote that A is PSD; more generally, we write $A \succeq B$ if $A - B$ is PSD: this partial order on symmetric matrices is called the *Löwner order*.

Proof. Let $j \neq i$. The determinant of the submatrix indexed by $\{i, j\}$ is

$$A_{i,i}A_{j,j} - A_{i,j}A_{j,i}$$

is nonnegative, by assumption. Since $A_{i,j} = A_{j,i}$ by symmetry, and $A_{i,i} = 0$, we get $A_{i,j}^2 = A_{j,i}^2 \leq 0$ and we conclude $A_{i,j} = A_{j,i} = 0$. □

Definition 21.3 (Frobenius Product)**.** Let $A, B \in \mathbb{R}^{n \times n}$. The Frobenius inner product $A \bullet B$, also written as $\langle A, B \rangle$ is defined as

$$\langle A, B \rangle := A \bullet B := \sum_{i,j} A_{i,j}B_{i,j} = \mathrm{Tr}(A^\intercal B).$$

We can think of this as being the usual vector inner product treating A and B as vectors of length $n \times n$. Note that by the cyclic property of the trace, $A \bullet xx^\intercal = \mathrm{Tr}(Axx^\intercal) = \mathrm{Tr}(x^\intercal Ax) = x^\intercal Ax$; we will use this fact to derive yet another of PSD matrices.

Lemma 21.4. *A is PSD if and only if $A \bullet X \geq 0$ for all $X \succeq 0$.*

Proof. Suppose $A \succeq 0$. Consider the spectral decomposition $X = \sum_i \lambda_i x_i x_i^\intercal$ where $\lambda_i \geq 0$ by Definition 21.1(b). Then

$$A \bullet X = \sum_i \lambda_i (A \bullet x_i x_i^\intercal) = \sum_i \lambda_i \, x_i^\intercal A x_i \geq 0.$$

On the other hand, if $A \not\succeq 0$, there exists v such that $v^\intercal A v < 0$, by 21.1(a). Let $X = vv^\intercal \succeq 0$. Then $A \bullet X = v^\intercal A v < 0$. □

Finally, let us mention a useful fact:

Fact 21.5 (PSD cone). Given two matrices $A, B \succeq 0$, and scalars $\alpha, \beta > 0$ then $\alpha A + \beta B \succeq 0$. Hence the set of PSD matrices forms a convex cone in $\mathbb{R}^{n(n+1)/2}$.

Here $n(n+1)/2$ is the number of entries on or above the diagonal in an $n \times n$ matrix, and completely specifies a symmetric matrix.

21.2 *Semidefinite Programs*

Loosely, a semidefinite program (SDP) is the problem of optimizing a linear function over the intersection of a convex polyhedron K (given by finitely many linear constraints, say $Ax \geq b$) with the PSD cone $\mathcal{K}$. Let us give two useful packagings for semidefinite programs.

21.2.1 *As Linear Programs with a PSD Constraint*

Consider a linear program where the variables are indexed by pairs $i, j \in [n]$, i.e., a typical variable is $x_{i,j}$. Let X be the $n \times n$ dimensional matrix whose $(i,j)^{th}$ entry is $x_{i,j}$. As the objective and constraints are linear, we can write them as $C \bullet X$ and $A_k \bullet X \leq b_k$ for some (not necessarily PSD) matrices $C, A_1, \ldots, A_m$ and scalars $b_1, \ldots, b_m$. An SDP is an LP of this form with the additional constraint $X \succeq 0$:

$$\begin{aligned} \underset{X \in \mathbb{R}^{n\times n}}{\text{maximize}} \quad & C \bullet X \\ \text{subject to} \quad & A_k \bullet X \leq b_k, \qquad \forall k \in [m] \\ & X \succeq 0. \end{aligned}$$

Observe that if each of the matrices A_i and C are diagonal matrices, say with diagonals a_i and c, this SDP becomes the linear program

$$\max\{c^\intercal x \mid a_k^\intercal x \leq b_k, x \geq 0\},$$

where x denotes the diagonal of the PSD matrix X.

21.2.2 *As Vector Programs*

We can use Definition 21.1(d) to rewrite the above program as a "vector program": where the linear objective and the linear constraints are on inner products of vector variables:

$$\begin{aligned} \underset{v_1,\ldots,v_n \in \mathbb{R}^n}{\text{maximize}} \quad & \sum_{i,j} c_{ij} \langle v_i, v_j \rangle \\ \text{subject to} \quad & \sum_{i,j} a_{ij}^{(k)} \langle v_i, v_j \rangle \leq b_k, \qquad \forall k \in [m]. \end{aligned}$$

In particular, we optimize over vectors in n-dimensional space; we cannot restrict the dimension of these vectors, much like we cannot restrict the rank of the matrices X in the previous representation.

21.2.3 *Examples of SDPs*

Let A a symmetric $n \times n$ real matrix. Here is an SDP to compute the maximum eigenvalue of A:

$$\begin{aligned} \underset{X \in \mathbb{R}^{n\times n}}{\text{maximize}} \quad & A \bullet X \\ \text{subject to} \quad & I \bullet X = 1 \\ & X \succeq 0 \end{aligned} \tag{21.1}$$

Lemma 21.6. *SDP (21.1) computes the maximum eigenvalue of A.*

Proof. Let X maximize SDP (21.1) (this exists as the objective is continuous and the feasible set is compact). Consider the spectral decomposition $X = \sum_{i=1}^n \lambda_i x_i x_i^\intercal$ where $\lambda_i \geq 0$ and $\|x_i\|_2 = 1$. The trace constraint $I \bullet X = 1$ implies $\sum_i \lambda_i = 1$. Thus the objective value $A \bullet X = \sum_i \lambda_i x_i^\intercal A x_i$ is a convex combination of $x_i^\intercal A x_i$. Hence without loss of generality, we can put all the weight into one of these terms, in which case $X = yy^\intercal$ is a rank-one matrix with $\|y\|_2 = 1$. By the Courant-Fischer theorem, $OPT \leq \max_{\|y\|_2=1} y^\intercal A y = \lambda_{\max}$.

On the other hand, letting v be a unit eigenvector of A corresponding to $\lambda_{\max}$, we have that $OPT \geq A \bullet vv^\intercal = v^\intercal A v = \lambda_{\max}$. ☐

Here is another SDP for the same problem:

$$\begin{aligned} \underset{t}{\text{minimize}} \quad & t \\ \text{subject to} \quad & tI - A \succeq 0. \end{aligned} \tag{21.2}$$

In fact, it turns out that this SDP is dual to the one in (21.1). Weak duality still holds for this case, but strong duality does not hold in general for SDPs. Indeed, there could be a duality gap for some cases, where both the primal and dual are finite, but the optimal solutions are not equal to each other. However, under some mild regularity conditions (e.g., the Slater conditions) we can show strong duality. More about SDP duality here.

Lemma 21.7. *SDP (21.2) computes the maximum eigenvalue of A.*

Proof. The matrix $tI - A$ has eigenvalues $t - \lambda_i$. And hence the constraint $tI - A \succeq 0$ is equivalent to the constraint $t - \lambda \geq 0$ for all its eigenvalues λ. In other words, $t \geq \lambda_{\max}$, and thus $OPT = \lambda_{\max}$. ☐

21.3 *SDPs in Approximation Algorithms*

We now consider designing approximation algorithms using SDPs. Recall that given a matrix A, we can check if it is PSD in (strongly) polynomial time, by performing its eigendecomposition. Moreover, if A is not PSD, we can return a hyperplane separating A from the PSD cone. Thus using the ellipsoid method, we can approximate SDPs when OPT is appropriately bounded. Informally,

We know that there is an optimal LP solution where the numbers are singly exponential, and hence can be written using a polynomial number of bits. But this is not true in SDPs, in fact, OPT in an SDP may be as large (or small) as doubly exponential in the size of the SDP. (See Section 2.6 of the Matoušek and Gärtner.)

Theorem 21.8 (Informal Theorem). *Assuming that the radius of the feasible set is at most* $\exp(\text{poly}(\langle SDP \rangle))$, *the ellipsoid algorithm can weakly solve SDP in time* $\text{poly}(\langle SDP \rangle, \log(1/\varepsilon))$ *up to an additive error of ε.*

For a formal statement, see Theorem 2.6.1 of Matoušek and Gärtner. However, we will ignore these technical issues in the remainder of this chapter, and instead suppose that we can solve our SDPs exactly.

21.4 *The* MAXCUT *Problem and Hyperplane Rounding*

Given a graph $G = (V, E)$, the MAXCUT problem asks us to find a partition of the vertices $(S, V \setminus S)$ maximizing the number of edges crossing the partition. This problem is **NP**-complete. In fact assuming **P** $\neq$ **NP**, a result of Johan Håstad shows that we cannot approximate MAXCUT better than $17/16 - \varepsilon$ for any $\varepsilon > 0$.

21.4.1 *Greedy and Randomized Algorithms*

We begin by considering a greedy algorithm: process the vertices $v_1, \ldots, v_n$ in some order, and place each vertex v_i in the part of the bipartition that maximizes the number of edges cut so far (breaking ties arbitrarily).

Lemma 21.9. *The greedy algorithm cuts at least $|E|/2$-many edges.*

Proof. Let δ_i be the number of edges from vertex i to vertices $j < i$: then the greedy algorithm cuts at least $\sum_i \delta_i/2 = |E|/2$ edges. □

This result shows two things: (a) every graph has a bipartition that cuts half the edges of the graph, so Opt $\geq |E|/2$. Moreover, (b) that since Opt $\leq |E|$ on any graph, this means that Alg $\geq |E|/2 \geq$ Opt $/2$.

We cannot hope to prove a better result than Lemma 21.9 in terms of $|E|$, since the complete graph K_n has $\binom{n}{2} \approx n^2/2$ edges and any partition can cut at most $n^2/4$ of them.

Here's a simple randomized algorithm: place each vertex in either S or in $\bar{S}$ independently and uniformly at random. Since each edge is cut with probability 1/2, the expected number of cut edges is $|E|/2$. Moreover, by the probabilistic method Opt $\geq |E|/2$.

21.4.2 *Relax-and-Round using LPs*

A natural direction would be to write an ILP formulation for MaxCut and to relax it: this approach does not give us anything beyond a factor of 1/2, say.

21.4.3 *A Semidefinite Relaxation*

We now see a well-known example of an SDP-based approximation algorithm due to Michel Goemans and David Williamson. Again, we will use the relax-and-round framework from the previous chapter. The difference is that we write a quadratic program to model the problem exactly, and then relax it to get an SDP.

Indeed, observe that the MaxCut problem can be written as the following quadratic program.

$$\begin{aligned} \underset{x_1,\ldots,x_n \in \mathbb{R}}{\text{maximize}} \quad & \sum_{(i,j)\in E} \frac{(x_i - x_j)^2}{4} \\ \text{subject to} \quad & x_i^2 = 1 \qquad \forall i. \end{aligned} \tag{21.3}$$

Since each x_i is real-valued, and $x_i^2 = 1$, each variable must be assigned one of two labels $\{-1, +1\}$. Since each term in the objective contributes 1 for an edge connecting two vertices in different partitions, and 0 otherwise, this IP precisely captures MaxCut.

We now relax this program by replacing the variables x_i with vector variables $v_i \in \mathbb{R}^n$, where $\|v_i\|^2 = 1$.

$$\begin{aligned} \underset{v_1,\ldots,v_n \in \mathbb{R}^n}{\text{maximize}} \quad & \sum_{(i,j)\in E} \frac{\|v_i - v_j\|^2}{4} \\ \text{subject to} \quad & \|v_i\|^2 = 1 \qquad \forall i. \end{aligned} \tag{21.4}$$

Noting that $\|v_i - v_j\|^2 = \|v_i\|^2 + \|v_j\|^2 - 2\langle v_i, v_j\rangle = 2 - 2\langle v_i, v_j\rangle$, we rewrite this vector program as

The SDP relaxation for the MaxCut problem was first introduced by Svata Poljak and Franz Rendl.

$$\begin{aligned} \underset{v_1,\ldots,v_n \in \mathbb{R}^n}{\text{maximize}} \quad & \sum_{(i,j)\in E} \frac{1-\langle v_i, v_j\rangle}{2} \\ \text{subject to} \quad & \langle v_i, v_i\rangle = 1 \qquad \forall i. \end{aligned} \tag{21.5}$$

This is a relaxation of the original quadratic program, because we can model any $\{-1,+1\}$-valued solution using vectors, say by a corresponding $\{-e_1, +e_1\}$-valued solution. Since this is a maximization problem, the SDP value is now at least the optimal value of the quadratic program.

21.4.4 The Hyperplane Rounding Technique

In order to round this vector solution $\{v_i\}$ to the MaxCut SDP into an integer scalar solution to MaxCut, we use the remarkably simple method of ***hyperplane rounding***. The idea is this: a term in the SDP objective incurs a tiny cost close to zero when v_i, v_j are very close to each other, and almost unit cost when v_i, v_j point in nearly opposite directions. So we would like to map close vectors to the same value.

To do this, we randomly sample a hyperplane through the origin and partition the vectors according to the side on which they land. Formally, this corresponds to picking a vector $g \in \mathbb{R}^n$ according to the standard n-dimensional Gaussian distribution, and setting

$$S := \{i \mid \langle v_i, g\rangle \geq 0\}.$$

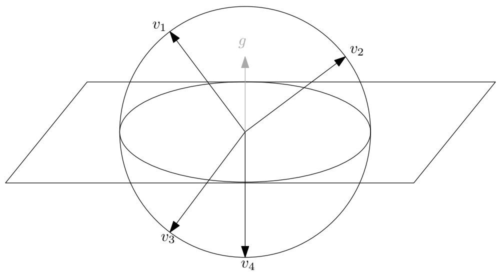


Figure 21.1: A geometric picture of Goemans-Williamson randomized rounding

We now argue that this procedure gives us a good cut in expectation; this procedure can be repeated to get an algorithm that succeeds with high probability.

Theorem 21.10. *The partition produced by the hyperplane rounding algorithm cuts at least* $\alpha_{GW} \cdot \mathrm{SDP}$ *edges in expectation, where* $\alpha_{GW} := 0.87856$.

Proof. By linearity of expectation, it suffices to bound the probability of an edge (i,j) being cut. Let

$$\theta_{ij} := \cos^{-1}(\langle v_i, v_j\rangle)$$

be the angle between the unit vectors v_i and v_j. Now consider the 2-dimensional plane P containing v_i, v_j and the origin, and let $\widetilde{g}$ be the projection of the Gaussian vector g onto this plane. Observe that the edge (i,j) is cut precisely when the hyperplane defined by g separates v_i, v_j. This is precisely when the vector perpendicular to $\widetilde{g}$ in the plane P lands between v_i and v_j. As the projection onto a subspace of the standard Gaussian is again a standard Guassian (by spherical symmetry),

$$\Pr[(i,j) \text{ cut}] = \frac{2\theta_{ij}}{2\pi} = \frac{\theta_{ij}}{\pi}.$$

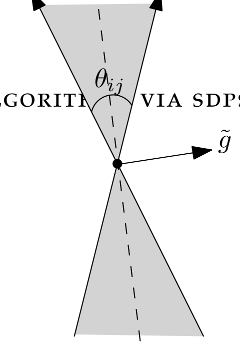


Figure 21.2: Angle between two vectors. We cut edge (i, j) when the vector perpendicular to $\tilde{g}$ lands in the grey area.

Since the SDP gets a contribution of

$$\frac{1 - \langle v_i, v_j \rangle}{2} = \frac{1 - \cos(\theta_{i,j})}{2}$$

for this edge, it suffices to show that

$$\frac{\theta}{\pi} \geq \alpha \frac{1 - \cos\theta}{2}.$$

Indeed, we can show (either by plotting, or analytically) that $\alpha = 0.87856\ldots$ suffices for the above inequality, and hence

$$\mathbb{E}[\#\text{ edges cut}] = \sum_{(i,j)\in E} \theta_{ij}/\pi \geq \alpha \sum_{(i,j)\in E} \frac{1-\cos(\theta_{ij})}{2} = \alpha \text{ SDP}.$$

This proves the theorem. □

Corollary 21.11. *For any $\varepsilon > 0$, repeating the hyperplane rounding algorithm $O(1/\varepsilon \log 1/\delta)$ times and returning the best solution ensures that we output a cut of value at least $(.87856 - \varepsilon)$ Opt with probability $1 - \delta$.*

We leave this proof as an exerise in using Markov's inequality: note that we want to show that the algorithm returns something not too far *below* the expecation, which seems to go the wrong way, and hence requires a moment's thought.

The above algorithm is randomized and the result only holds in expectation. However, it is possible to derandomize this result to obtain a polynomial-time deterministic algorithm with the same approximation ratio.

21.4.5 *Subsequent Work and Connections*

Can we get a better approximation factor, perhaps using a more sophisticated SDP? An influential result of Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O'Donnell says that a constant-better-than-α_{GW}-approximation would refute the Unique Games Conjecture.

Also, one can ask if similar rounding procedures exist for an linear-programming relaxation as opposed to the SDP relaxation here. Unfortunately the answer is again no: a result of Siu-On Chan, James Lee, Prasad Raghavendra, and David Steurer shows that no polynomial-sized LP relaxation of MaxCut can obtain a non-trivial approximation factor, that is, any polynomial sized LP of MaxCut has an integrality gap of 1/2.

21.5 *Coloring 3-Colorable Graphs*

Suppose we are given a graph $G = (V, E)$ and a promise that there is some 3-coloring of G. What is the minimum k such that we can find a k-coloring of G in polynomial time? It is well-known that 2-coloring a graph can be done in linear time, but 3-coloring a graph is **NP**-complete. Hence, even given a 3-colorable graph, it is **NP**-hard to color it using 3 colors. (In fact, a result of Venkat Guruswami and Sanjeev Khanna shows that it is **NP**-hard to color it using even 4 colors.) But what if we ask to color a 3-colorable graph using 5 colors? $O(\log n)$ colors? $O(n^\alpha)$ colors, for some fixed constant α? We will see an easy algorithm to achieve an $O(\sqrt{n})$-coloring, and then will use semidefinite programming to improve this to an $\widetilde{O}(n^{\log_6(2)})$ coloring. Before we describe these, let us recall the easy part of Brooks' theorem.

The harder part is to show that in fact Δ colors suffice unless the graph is either a complete graph, or an odd-length cycle.

Lemma 21.12. *Let Δ be the maximum degree of a graph G, then we can find a $(\Delta+1)$-coloring of G in linear time.*

Proof. Pick any vertex v, recursively color the remaining graph, and then assign v a color not among the colors of its Δ neighbors. □

We will now describe an algorithm that colors a 3-colorable graph G with $O(\sqrt{n})$ colors, originally due to Avi Wigderson: while there exists a vertex with degree at least $\sqrt{n}$, color it using a fresh color. Moreover, its neighborhood must be 2-colorable, so use two fresh colors to do so. This takes care of $\sqrt{n}$ vertices using 3 colors. Remove these, and repeat. Finally, use Lemma 21.12 to color the remaining vertices using $\sqrt{n}$ colors. This proves the following result.

Lemma 21.13. *There is an algorithm to color a 3-colorable graph with $O(\sqrt{n})$ colors.*

21.5.1 An Algorithm using SDPs

Let's consider an algorithm that uses SDPs to color a 3-colorable graph with maximum degree Δ using $\widetilde{O}(\Delta^{\log_3 2}) \approx \widetilde{O}(\Delta^{0.63})$ colors. In general Δ could be as large as n, so this could be worse than the algorithm in Lemma 21.13, but we will be able to combine the ideas together to get a better result.

For some parameter $\lambda \in \mathbb{R}$, consider the following feasibility SDP (where we are not optimizing any objective):

$$\begin{aligned} &\text{find} && v_1, \ldots, v_n \in \mathbb{R}^n \\ &\text{subject to} && \langle v_i, v_j \rangle \le \lambda && \forall (i,j) \in E \\ &&& \langle v_i, v_i \rangle = 1 && \forall i \in V. \end{aligned} \tag{21.6}$$

Why is this SDP relevant to our problem? The goal is to have vectors clustered together in groups, such that each cluster represents a color. Intuitively, we want to have vectors of adjacent vertices to be far apart, so we want their inner product to be close to -1 (recall we are dealing with unit vectors, due to the last constraint) and vectors of the same color to be close together.

Lemma 21.14. *For 3-colorable graphs, SDP (21.6) is feasible with $\lambda = -1/2$.*

Proof. Consider the vector placement shown in the figure to the right.

If the graph is 3-colorable, we can assign all vertices with color 1 the red vector, all vertices with color 2 the blue vector and all vertices with color 3 the green vector. Now for every edge $(i, j) \in E$, we have that

$$\langle v_i, v_j \rangle = \cos\left(\frac{2\pi}{3}\right) = -1/2. \qquad \square$$

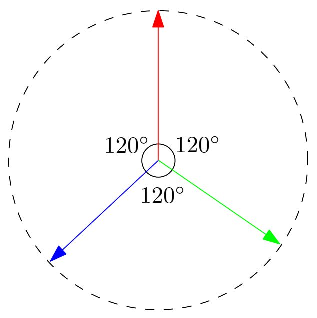


Figure 21.3: Optimal distribution of vectors for 3-coloring graph

At first sight, it may seem like we are done: if we solve the above SDP with $\lambda = -1/2$, don't all three vectors look like the figure above? No, that would only hold if all of them were to be co-planar. And in n-dimensions we can have an exponential number of cones of angle $\frac{2\pi}{3}$, like in the next figure, so we cannot cluster vectors as easily as in the above example.

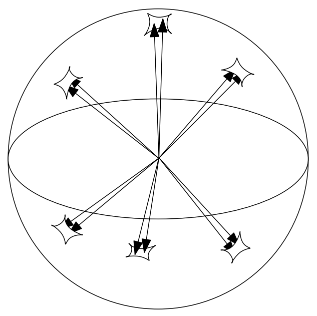


Figure 21.4: Dimensionality problem of $2\pi/3$ far vectors

To solve this issue, we apply a hyperplane rounding technique similar to that from the MaxCut algorithm. Indeed, for some parameter t we will pick later, pick t random hyperplanes. Formally, we pick $g_i \in \mathbb{R}^n$ from a standard n-dimensional Gaussian distribution, for $i \in [t]$. Each of these defines a normal hyperplane, and these split the $\mathbb{R}^n$ unit sphere into 2^t regions (except if two of them point in the same direction, which has zero probability). Now, each vectors $\{v_i\}$ that lie in the same region can be considered "close" to each other, and we can try to assign them a unique color. Formally, this means that if v_i and v_j are such that

$$\text{sign}(\langle v_i, g_k \rangle) = \text{sign}(\langle v_j, g_k \rangle)$$

for all $k \in [t]$, then i and j are given the same color. Each region is given a different color, of course.

However, this may color some neighbors with the same color, so we use the ***method of alterations***: while there exists an edge between vertices of the same color, we uncolor both endpoints. When this uncoloring stops, we remove the still-colored vertices from the graph, and then repeat the same procedure on the remaining graph, until we color every vertex. Note that since we use t hyperplanes, we add at most 2^t new colors per iteration. The goal is to now show that (a) the number of interations is small, and (b) the value of 2^t is also small.

Lemma 21.15. *If half of the vertices are colored in a single iteration in expectation, then the expected number of iterations to color the whole graph is $O(\log n)$.*

Proof. Since the expected number of uncolored vertices is at most half, Markov's inequality says that more than 3/4 of the vertices are uncolored in a single iteration, with probability at most 2/3. In other words, at least 1/4 of the vertices are colored with probability 1/3. Hence, the number of iterations to color the whole graph is dominated by the number of flips of a coin of bias 1/3 to get $\log_4 n$ heads. This is $4 \log_4 n$, which proves the result. □

Lemma 21.16. *The expected number of vertices that remain uncolored after a single iteration is at most* $n\Delta\,(1/3)^t$.

Proof. Fix an edge ij: for a single random hyperplane, the probability that v_i, v_j are not separated by it is

$$\frac{\pi - \theta_{ij}}{\pi} \leq \frac{1}{3},$$

using that $\theta_{ij} \geq \frac{2\pi}{3}$ which follows from the constraint in the SDP. Now if i is uncolored because of j, then v_i, v_j have the same color, which happens when all t hyperplanes fail to separate the two. By independence, this happens with probability at most $(1/3)^t$. Finally,

$$\begin{aligned} \mathbb{E}[\text{remaining}] &= \sum_{i \in V} \Pr[i \text{ uncolored}] \\ &\leq \sum_{i \in V} \sum_{(i,j) \in E} \Pr[i \text{ uncolored because of } j]. \end{aligned} \tag{21.7}$$

There are n vertices, and each vertex has degree at most Δ, which proves the result. □

Lemma 21.17. *There is an algorithm that colors a 3-colorable graph with maximum degree* Δ *with* $O(\Delta^{\log_3 2} \cdot \log n)$ *colors in expectation.*

Proof. Setting $t = \log_3(2\Delta)$ in Lemma 21.16, the expected number of uncolored vertices in any iteration is

$$n \cdot \Delta \cdot (1/3)^t \leq n/2. \tag{21.8}$$

Now Lemma 21.15 says we perform $O(\log n)$ iterations in expectation. Since we use most $2^{\log_3(2\Delta)} = (2\Delta)^{\log_3 2}$ colors in each iteration, we get the result. □

21.5.2 Improving the Algorithms Further

The expected number of colors used by the above algorithm is $\widetilde{O}(n^{\log_3 2}) \approx \widetilde{O}(n^{0.63})$, which is worse than our initial $O(\sqrt{n})$ algorithm. However we can combine the ideas together to get a better result:

Theorem 21.18. *There is an algorithm that colors a 3-colorable graph with $\widetilde{O}(n^{\log_6(2)})$ colors.*

Proof. For some value σ, repeatedly remove vertices with degree greater than σ and color them and their neighbors with 3 new colors, as in Lemma 21.13. This requires at most $3n/\sigma$ colors overall, and leaves us with a graph having maximum degree σ. Now use Lemma 21.17 to color the remaining graph with $O(\sigma^{\log_3 2} \cdot \log n)$ colors. Picking σ to be $n^{\log_6 3}$ to balance these terms, we get a procedure that uses $\widetilde{O}(n^{\log_6 2}) \approx \widetilde{O}(n^{0.38})$ colors. $\square$

21.5.3 *Final notes on coloring 3-colorable graphs*

This result us due to David Karger, Rajeev Motwani, and Madhu Sudan. They gave a better rounding algorithm that uses spherical caps instead of hyperplanes to achieve $\widetilde{O}(n^{1/4})$ colors. This result was then improved over a sequence of papers: the current best result by Ken-Ichi Kawarabayashi and Mikkel Thorup uses $O(n^{0.199})$ colors. It remains an outstanding open problem to either get a better algorithm, or to show hardness results, even under stronger complexity-theoretic hypotheses.

22
Online Algorithms

In this chapter we introduce ***online algorithms*** and study two classic online problems: the rent-or-buy problem, and the paging problem. While the models we consider are reminiscent of those in *regret minimization* in online learning and online convex optimization, there are some important differences, which lend a different flavor to the results that are obtained.

22.1 *The Competitive Analysis Framework*

In the standard setting of online algorithms there is a sequence of requests $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_t, \ldots)$ received online. An online algorithm does not know the input sequence up-front, but sees these requests one by one. It must serve request σ_i before it is allowed to see σ_{i+1}. Serving this request σ_i involves some choice of actions, and incurs some cost. We will measure the performance of an algorithm by considering the ratio of the total cost incurred on σ to the optimal cost of serving σ in hindsight. To make all this formal, let us xsee an example of an online problem.

22.1.1 *Example: The Paging Problem*

The paging problem arises in computer memory systems. Often, a memory system consists of a large but slow *main memory*, as well as a small but fast memory called a *cache*. The CPU typically communicates directly with the cache, so in order to access an item that is not contained in the cache, the memory system has to load the item from the slow memory into the cache. Moreover, if the cache is full, then some item contained in the cache has to be evicted to create space for the requested item.

We say that a *cache miss* occurs whenever there is a request to an item that is not currently in the cache. The goal is to come up with an eviction strategy that minimizes the number of cache misses.

Typically we do not know the future requests that the CPU will make so it is sensible to model this as an online problem.

If the entire sequence of requests is known, show that *Belády's rule* is optimal: evict the page in cache that is next requested furthest in the future.

We let U be a universe of n items or pages. The cache is a memory containing at most k pages. The requests are pages $\sigma_i \in U$ and the online algorithm is an eviction policy. Now we return back to defining the performance of an online algorithm.

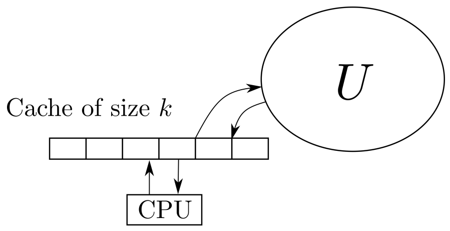


Figure 22.1: Illustration of the Paging Problem

22.1.2 *The Competitive Ratio*

As we said before, the online algorithm incurs some cost as it serves each request. If the complete request sequence is σ, then we let $\text{Alg}(\sigma)$ be the total cost incurred by the online algorithm in serving σ. Similarly, we let $\text{Opt}(\sigma)$ be the optimal cost in hindsight of serving σ. Note that $\text{Opt}(\sigma)$ represents the cost of an optimal *offline* algorithm that knows the full sequence of requests. Now we define the ***competitive ratio*** of an algorithm to be:

$$\max_{\sigma} \frac{\text{Alg}(\sigma)}{\text{Opt}(\sigma)}$$

In some sense this is an "apples to oranges" comparison, since the online algorithm does not know the full sequence of requests, whereas the optimal cost is aware of the full sequence and hence is an "offline" quantity.

Note two differences from regret minimization: there we made a prediction x_t before (or concurrently with) seeing the function f_t, whereas we now see the request σ_t before we produce our response at time t. In this sense, our problem is easier. However, the benchmark is different—we now have to compare with the best *dynamic* sequence of actions for the input sequence σ, whereas regret is typically measured with respect to a *static* response, i.e., to the cost of playing the same fixed action for each of the t steps. In this sense, we are now solving a harder problem. There are is a smaller, syntactic difference as well: regret is an additive guarantee whereas the competitive ratio is a multiplicative guarantee—but this is more a reflection on the kind of results that are possible, rather than fundamental difference between the two models.

22.1.3 *What About Randomized Algorithms?*

The above definitions generally hold for deterministic algorithms, so how should we characterize randomized algorithms. For the deterministic case we generally think about some adversary choosing the worst possible request sequence for our algorithm. For randomized algorithms we could consider either oblivious or adaptive adversaries. Oblivious adversaries fix the input sequence up front and then let the randomized algorithm process it. An adaptive adversary is allowed to see the results of the coin flips the online algorithm makes and thus adapt its request sequence. We focus on oblivious adversaries in these notes.

To define the performance of a randomized online algorithm we just consider the expected cost of the algorithm. Against an oblivious adversary, we say that a randomized online algorithm is α-competitive if for all request sequences σ,

$$\mathbb{E}[\mathrm{Alg}(\sigma)] \leq \alpha \cdot \mathrm{Opt}(\sigma).$$

22.2 *The Ski Rental Problem: Rent or Buy?*

Now that we have a concrete analytical framework, let's apply it to a simple problem. Suppose you are on a ski trip with your friends. On each day you can choose to either rent or buy skis. Renting skis costs \$1, whereas buying skis costs \$$B$ for $B > 1$. However, the benefit of buying skis is that on subsequent days you do not need to rent or buy again, just use the skis you already bought. The problem that arises is that for some mysterious reason we do not know how long the ski trip will last. On each morning we are simply told whether or not the trip will continue that day. The goal of the problem is to find a rent/buy strategy that is competitive with regards to minimizing the cost of the trip.

In the notation that we developed above, the request for the i'th day, σ_i, is either "Y" or "N" indicating whether or not the ski trip continues that day. We also now that once we see a "N" request that the request sequence has ended. For example a possible sequence might be $\sigma = (Y, Y, Y, N)$. This allows us to characterize all instances of the problem as follows. Let I_j be the sequence where the ski trip ends on day j. Suppose we knew ahead of time what instance we received, then we have that $\mathrm{Opt}(I_j) = \min\{j, B\}$ since we can choose to either buy skis on day 1 or rent skis every day depending on which is better.

22.2.1 *Deterministic Rent or Buy*

We can classify and analyze all possible deterministic algorithms since an algorithm for this problem is simply a rule deciding when to buy skis. Let Alg_i be the algorithm that rents skis until day i, then buys on day i if the trip lasts that long. The cost on instance I_j is then $\mathrm{Alg}_i(I_j) = (i - 1 + B) \cdot \mathbf{1}_{\{i \leq j\}} + j \cdot \mathbf{1}_{\{i > j\}}$. What is the best deterministic algorithm from the point of view of competitive analysis? The following claims answer this question.

Lemma 22.1. *The competitive ratio of algorithm* Alg_B *is* $2 - 1/B$ *and this is the best possible ratio for any deterministic algorithm.*

Proof. There are two cases to consider $j < B$ and $j \geq B$. For the first case, $\text{Alg}_B(I_j) = j$ and $\text{Opt}(I_j) = j$, so $\text{Alg}_B(I_j)/\text{Opt}(I_j) = 1$. In the second case, $\text{Alg}_B(I_j) = 2B - 1$ and $\text{Opt}(I_j) = B$, so $\text{Alg}_B(I_j)/\text{Opt}(I_j) = 2 - 1/B$. Thus the competitive ratio of Alg_B is

$$\max_{I_j} \frac{\text{Alg}_B(I_j)}{\text{Opt}(I_j)} = 2 - 1/B$$

Now to show that this is the best possible competitive ratio for any deterministic algorithm. Consider algorithm Alg_i. We find an instance I_j such that $\text{Alg}_i(I_j)/\text{Opt}(I_j) \geq 2 - 1/B$. If $i \geq B$ then we take $j = B$ so that $\text{Alg}_i(I_j) = (i - 1 + B)$ and $\text{Opt}(I_j) = B$ so that

$$\frac{\text{Alg}_i(I_j)}{\text{Opt}(I_j)} = \frac{i-1+B}{B} = \frac{i}{B} + 1 - \frac{1}{B} \geq 2 - \frac{1}{B}$$

Now if $i = 1$, we take $j = 1$ so that

$$\frac{\text{Alg}_i(I_j)}{\text{Opt}(I_j)} = \frac{B}{1} \geq 2$$

Since B is an integer > 1 by assumption. Now for $1 < i < B$, we take $j = \lfloor (i - 1 + B)/(2 - 1/B) \rfloor \geq 1$ so that

$$\frac{\text{Alg}_i(I_j)}{\text{Opt}(I_j)} \geq 2 - \frac{1}{B}.$$

□

22.2.2 *Randomized Rent or Buy*

Can randomization improve over deterministic algorithms in terms of expected cost? We will show that this is in fact the case. So how do we design a randomized algorithm for this problem? We use the following general insight about randomized algorithms, notably that *a randomized algorithm is a distribution over deterministic algorithms*.

To keep things simple let's consider the case when $B = 4$. We construct the following table of payoffs where the rows correspond to deterministic algorithms Alg_i and the columns correspond to instances I_j.

	I_1	I_2	I_3	I_∞
Alg_1	4/1	4/2	4/3	4/4
Alg_2	1/1	5/2	5/3	5/4
Alg_3	1/1	2/2	6/3	6/4
Alg_4	1/1	2/2	3/3	7/4

While the real game is infinite along with coordinates, we do not need to put columns for $I_4, I_5, \ldots$ because these strategies for the adversary are dominated by the column I_∞. Now given that the adversary would rather give us only these inputs, we do not need to put rows after $B = 4$ since buying after day B is worse than buying on day B for these inputs. (Please check these for yourself!) This means we can think of the above table as a 2-player zero-sum game with 4 strategies each. The row player chooses an algorithm and the column player chooses an instance, then the number in the corresponding entry indicates the loss of the row player. Thinking along the lines of the Von Neumann minimax theorem, we can consider mixed strategies for the row player to construct a randomized algorithm for the ski rental problem.

Let p_i be the probability of our randomized algorithm choosing row i. What is the expected cost of this algorithm? Suppose that the competitive ratio was at most c in expectation. The expected competitive ratio of our algorithm against each instance should be at most c, so this yields the following linear constraints.

$$\begin{aligned} 4p_1 + p_2 + p_3 + p_4 &\leq c \\ \frac{4p_2 + 5p_2 + 2p_3 + 2p_4}{2} &\leq c \\ \frac{4p_1 + 5p_3 + 6p_3 + 3p_4}{3} &\leq c \\ \frac{4p_1 + 5p_2 + 6p_3 + 7p_4}{4} &\leq c \end{aligned}$$

We would like to minimize c subject to $p_1 + p_2 + p_3 + p_4 = 1$ and $p_i \geq 0$. It turns out that one can do this by solving the following system of *equations*:

Why is it OK to set the inequalities to equalities? Simply because it works out: in essence, we are guessing that making these four constraints tight gives a basic feasible solution—and our guess turns out to right. It does not show optimality, but we can do that by giving a matching dual solution.

$$\begin{aligned} p_1 + p_2 + p_3 + p_4 &= 1 \\ 4p_1 + p_2 + p_3 + p_4 &= c \\ 4p_1 + 5p_2 + 2p_3 + 2p_4 &= 2c \\ 4p_1 + 5p_2 + 6p_3 + 3p_4 &= 3c \\ 4p_1 + 5p_2 + 6p_3 + 7p_4 &= 4c \end{aligned}$$

Subtracting each line from the previous one gives us

$$\begin{aligned} p_1 + p_2 + p_3 + p_4 &= 1 \\ 3p_1 &= c - 1 \\ 4p_2 + p_3 + p_4 &= c \\ 4p_3 + p_4 &= c \\ 4p_4 &= c. \end{aligned}$$

This gives us $p_4 = c/4$, $p_3 = (3/4)(c/4)$, etc., and indeed that

$$c = \frac{1}{1-(1-1/4)^4} \qquad \text{and} \qquad p_i = (3/4)^{i-1}(c/4)$$

for $i = 1, 2, 3, 4$. For general B, we get

$$c = c_B = \frac{1}{1-(1-1/B)^B} \leq \frac{e}{e-1}.$$

Moreover, this value of c_B is indeed the best possible competitive ratio for anyth randomized algorithm for the ski rental problem. How might one prove such a result? We instead consider playing a random instance against a deterministic algorithm. By Von Neumann's minimax theorem the value of this should be the same as what we considered above. We leave it as an exercise to verify this for the case when $B = 4$.

22.2.3 *(Optional) A Continuous Approach*

This section is quite informal right now, needs to be made formal. For simplicity, assume $B = 1$ by scaling, and that both the algorithm and the length of the season can be any real in $[0, 1]$. Now our randomized algorithm chooses a threshold $t \in [0, 1]$ from the probability distribution with density function $f(t)$. Let's say f is continuous. Then we get that for any season length ℓ,

$$\int_{t=0}^{\ell} (1+t)\, f(t)\, dt + \int_{t=\ell}^{1} \ell\, f(t)\, dt = c \cdot \ell.$$

(Again, we're setting an equality there without justification, except that it works out.) Now we can differentiate w.r.t. ℓ to get

$$(1+\ell)\, f(\ell) + \int_{t=\ell}^{1} f(t)\, dt - \ell f(\ell) = c.$$

(This differentiation is like taking differences of successive lines that we did above.) Simplifying,

$$f(\ell) + \int_{t=\ell}^{1} f(t)\, dt = c. \tag{22.1}$$

Taking derivatives again:

$$f'(\ell) - f(\ell) = 0$$

But this solves to $f(\ell) = Ce^{\ell}$ for some constant C. Since f is a probability density function, $\int_{\ell=0}^{1} f(\ell) = 1$, we get $C = \frac{1}{e-1}$. Substituting into (22.1), we get that the competitive ratio is $c = \frac{e}{e-1}$, as desired.

22.3 The Paging Problem

Now we return to the paging problem that was introduced earlier and start by presenting a disappointing fact.

Lemma 22.2. *No deterministic algorithm can be $< k$-competitive for the paging problem.*

Proof. Consider a universe with $k+1$ pages in all. In each step the adversary requests a page not in the cache (there is always at least 1 such page). Thus the algorithm's cost over n requests is n. The optimal offline algorithm can cut losses by always evicting the item that will be next requested furthest in the future, thus it suffers a cache miss every k steps so the optimal cost will be n/k. Thus the competitive ratio of any deterministic algorithm is at least $\frac{n}{n/k} = k$. □

It is also known that many popular eviction strategies are k-competitive such as *Least Recently Used (LRU)* and *First-In First-Out (FIFO)*. We will show that a 1-bit variant of LRU is k-competitive and also show that a randomized version of it achieves an $O(\log k)$-competitive randomized algorithm for paging.

22.3.1 The 1-bit LRU/Marking Algorithm

The 1-bit LRU/Marking algorithm works in phases. The algorithm maintains a single bit for each page in the universe. We say that a page is marked/unmarked if its bit is set to 1/0. At the beginning of each phase, all pages are unmarked. When a request for a page not in the cache comes, then we evict an arbitrary unmarked page and put the requested page in the cache, then mark the requested page. If there are no unmarked pages to evict, then we unmark all pages and start a new phase.

Lemma 22.3. *The Marking algorithm is k-competitive for the paging problem.*

Proof. Consider the i'th phase of the algorithm. By definition of the algorithm, Alg incurs a cost of at most k during the phase since we can mark at most k different pages and hence we will have at most k cache misses in this time. Now consider the first request after the i'th phase ends. We claim that Opt has incurred at least 1 cache miss by the time of this request. This follows since we have now seen $k+1$ different pages. Now summing over all phases we see that $\text{Alg} \le k\,\text{Opt}$ □

Now suppose that instead of evicting an arbitrary unmarked page, we instead evicted an unmarked page uniformly at random. For this randomized marking algorithm we can prove a much better result.

Lemma 22.4. *The Randomized Marking Algorithm is $O(\log k)$-competitive*

Proof. We break up the proof into an upper bound on Alg's cost and a lower bound on Opt's cost. Before doing this we set up some notation. For the i^{th} phase, let S_i be the set of pages in the algorithm's cache *at the beginning of the phase*. Now define

$$\Delta_i = |S_{i+1} \setminus S_i|.$$

We claim that the expected number of cache misses made by the algorithm in phase i is at most $\Delta_i(H_k + 1)$, where H_k is the k^{th} harmonic number. By summing over all phases we see that $\mathbb{E}[\text{Alg}] \leq \sum_i \Delta_i(H_k + 1)$.

Now let R_i be the set of distinct requests in phase i. For each request in R_i we say that it is *clean* if the requested page is not S_i, otherwise the request is called *stale*. Every cache miss in the i^{th} phase is caused by either a clean request or a stale request.

1. The number of cache misses due to clean requests is at most Δ_i since there can be at most Δ_i clean requests in phase i: each clean request brings in a page not belonging to S_i into the cache and marks it, so it will be in S_{i+1}.

2. To bound the cache misses due to stale requests, suppose there have been c clean requests and s stale requests so far, and consider the $s + 1^{st}$ stale request. The probability this request causes a cache miss is at most $\frac{c}{k-s}$ since we have evicted c random pages out of $k - s$ remaining stale requests. Now since $c \leq \Delta_i$, we have that the expected cost due to stale requests is at most

This is like the Airline seat problem, where we can imagine that c confused passengers get on at the beginning.

$$\sum_{s=0}^{k-1} \frac{c}{k-s} \leq \Delta_i \sum_{s=0}^{k-1} \frac{1}{k-s} = \Delta_i H_k.$$

Now the expected total cost in phase i is at most

$$\Delta_i H_k + \Delta_i = \Delta_i(H_k + 1).$$

Now we claim that $\text{Opt} \geq \frac{1}{2}\sum_i \Delta_i$. Let S_i^* be the pages in Opt's cache at the beginning of phase i. Let ϕ_i be the number of pages in S_i but not in Opt's cache at the beginning of phase i, i.e., $\phi_i = |S_i \setminus S_i^*|$. Now let Opt_i be the cost that Opt incurs in phase i. We have that $\text{Opt}_i \geq \Delta_i - \phi_i$ since this is the number of "clean" requests that Opt sees. Moreover, consider the end of phase i. Alg has the k most recent requests in cache, but Opt does not have ϕ_{i+1} of them by definition of ϕ_{i+1}. Hence $\mathrm{Opt}_i \geq \phi_{i+1}$. Now by averaging,

$$\mathrm{Opt}_i \geq \max\{\phi_{i+1}, \Delta_i - \phi_i\} \geq \frac{1}{2}(\phi_{i+1} + \Delta_i - \phi_i).$$

So summing over all phases we have

$$\mathrm{Opt} \geq \frac{1}{2}\sum_i \Delta_i + \phi_{final} - \phi_{initial} \geq \frac{1}{2}\sum_i \Delta_i,$$

since $\phi_{final} \geq 0$ and $\phi_{initial} = 0$. Combining the upper and lower bound yields

$$\mathbb{E}[\mathrm{Alg}] \leq 2(H_k + 1)\,\mathrm{Opt} = O(\log k)\,\mathrm{Opt}. \qquad \square$$

It can also be shown that no randomized algorithm can do better than $\Omega(\log k)$-competitive for the paging problem. For some intuition as to why this might be true, consider the ***coupon collector problem***: if you repeatedly sample a uniformly random number from $\{1, \ldots, k+1\}$ with replacement, show that the expected number of samples to see all $k+1$ coupons is H_{k+1}.

22.4 *Generalizing Paging: The k-Server Problem*

Another famous problem in online algorithms is the ***k-server*** problem. Consider a metric space $M = (V, d)$ with point set V and distance function $d : V \times V \to \mathbb{R}_+$ satisfying the triangle inequality. In the k-server problem there are k servers that are located at various points of M. At each timestep t we receive a request $\sigma_t \in V$. If there is a server at point σ_t already, then we can server that request for free. Otherwise we move some server from point x to point σ_t and pay a cost equal to $d(x, \sigma_t)$. The goal of the problem is to serve the requests while minimizing the total cost of serving them.

The paging problem can be modeled as a k-server problem as follows. We let U be the points of the metric space and take $d(x, y) = 1$ for all pages x, y where $x \neq y$. This special case shows that every deterministic algorithm is at least k-competitive and every randomized algorithm is $\Omega(\log k)$-competitive by the discussion in the previous section. It is conjectured that there is a k-competitive deterministic algorithm: the best known result is a $(2k-1)$-competitive algorithm of Elias Koutsoupias and Christos Papadimitriou.

For randomized algorithms, a poly-logarithmic competitive algorithm was given by Nikhil Bansal, Niv Buchbinder, Aleksander Madry, and Seffi Naor. This was recently improved via an approach based on Mirror descent by Sebastien Bubeck, Michael Cohen, and James Lee, Yin Tat Lee, Aleksander Madry; see this paper of Niv Buchbinder, Marco Molinaro, Seffi Naor, and myself for a discretization.

Part V

Additional Topics

23 Smoothed Analysis of Algorithms

Smoothed analysis originates from an influential paper of Spielman and Teng, and provides an alternative to worst-case analysis. Assume that we want to analyze an algorithm's cost, e.g., the running time of an algorithm. Let $\text{cost}(A(I))$ be the cost that the algorithm has for input instance I. We will usually group the possible input instances according to their "size" (depending on the problem, this might be the number of jobs, vertices, variables and so on). Let $\mathcal{I}_n$ be the set of all inputs of 'size' n. For a worst case analysis, we are now interested in

$$\max_{I \in \mathcal{I}_n} \text{cost}(A(I)),$$

the maximum cost for any input of size n. Consider Figure 23.1 and imagine that all instances of size n are lined up on the x-axis. The blue line could be the running time of an algorithm that is fast for most inputs but very slow for a small number of instances. The worst case is the height of the tallest peak. For the green curve, which could for example be the running time of a dynamic programming algorithm, the worst case is a tight bound since all instances induce the same cost.

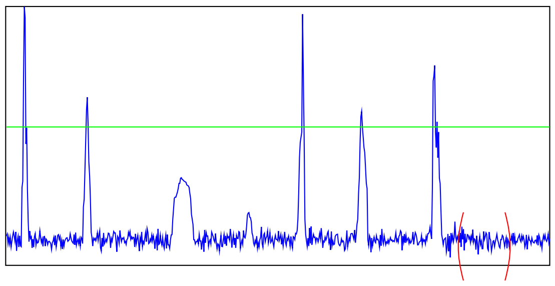


Figure 23.1: An example of a cost function with outliers.

Worst case analysis provides a bound that is true for all instances, but it might be very loose for some or most instances like it happens for the blue curve. When running the algorithm on real world data sets, the worst case instances might not occur, and there are different approaches to provide a more realistic analysis.

The idea behind smoothed analysis is to analyze how an algorithm acts on data that is jittered by a bit of random noise (notice that for real world data, we might have a bit of noise anyway). This is modeled by defining a notion of 'neighborhood' for instances and then analyzing the expected cost of an instance from the neighborhood of I instead of the running time of I.

The choice of the neighborhood is problem specific. We assume that the neighborhood of I is given in form of a probability distribution over all instances in $\mathcal{I}_n$. Thus, it is allowed that all instances in $\mathcal{I}_n$ are in the neighborhood but their influence will be different. The distribution depends on a parameter σ that says how much the input shall be jittered. We denote the neighborhood of I parameterized on σ by $\text{nbrhd}_\sigma(I)$. Then the smoothed complexity is given by

$$\max_{I \in \mathcal{I}_n} E_{I' \sim \text{nbrhd}_\sigma(I)}[\text{cost}(A(I'))]$$

In Figure 23.1, we indicate the neighborhood of one of the peak instances by the red brackets (imagine that the distribution is zero outside of the brackets – we will see one case of this in Section 23.3). The cost is smoothed within this area. For small σ, the bad instances get more isolated so that they dominate the expected value for their neighborhood, for larger σ, their influence decreases. We want the neighborhood to be big enough to smooth out the bad instances.

So far, we have mostly talked about the intuition behind smoothed analysis. The method has a lot of flexibility since the neighborhood can be defined individually for the analysis of each specific problem. We will see two examples in the following sections.

23.1 *The Traveling Salesman Problem*

The traveling salesman problem (TSP) is a classic example of an NP-complete problems with practical importance. In this problem, we are given an undirected weighted graph with (symmetric) edge weights $w_{ij} = w_{ji} \in [0, 1]$, find the minimum weighted cycle that contains all vertices.

On metric spaces there exists a polynomial 1.5-approximation algorithm (and now slightly better!) and on a d-dimensional Euclidean spaces (for fixed d) there are $1 + \varepsilon$ polynomial-time approximation schemes time (due to Sanjeev Arora, and others). In this chapter we consider the 2-OPT local-search heuristic.

23.1.1 *The 2-OPT Heuristic*

Start with an arbitrary cycle (i.e., a random permutation of vertices). In each iteration find two pairs of adjacent vertices (a, b) and (c, d) (i.e. $a \to b$ and $c \to d$ are edges in the cycle) and consider a new candidate cycle obtained by removing edges $a \to b$ and $c \to d$ as well as inserting edges $a \to c$ and $b \to d$. See Figure 23.2. If the new candidate cycle has smaller weight than the current cycle, replace the current cycle with the candidate one and repeat the heuristic. If no quadruplet (a, b, c, d) can decrease the weight, end the algorithm and report the final cycle.

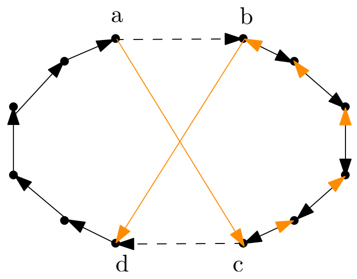


Figure 23.2: Main step of the 2-OPT heuristic: $a \to b, c \to d$ are replaced by $a \to c, b \to d$. The path $b \to \ldots \to c$ is also reversed.

There are two questions to be asked here: (a) how long does it take for us to reach the local optimum, and (b) what is the quality of the local optima?

It is useful to note that the 2-OPT always terminates since there at finite number of cycles (exactly $(n-1)!$ distinct ones) and each iteration strictly decreases the weight. However, there are examples on which the heuristic takes $\Omega(\exp(n))$ time as well as examples where it achieves value of $\Omega(\frac{\log n}{\log \log n}) \cdot OPT$. Despite this poor worst-case behavior, the heuristic performs well in practice, and hence it makes sense to see if smoothed analysis can explain its performance.

23.1.2 *The Smoothing Model and Result*

We set up the probability space by sampling each $w_{ij} \in [0, 1]$ from a independent distribution with probability density functions $f_{ij}(x)$ (note that the distributions can be different for different edges). The densities are bounded by $\frac{1}{\sigma}$, i.e. $f_{ij}(x) \leq \frac{1}{\sigma}$ for all i, j, x. The density can be otherwise completely arbitrary can chosen by the adversary.

We will prove that the expected number of iterations taken by 2-OPT is poly$(n, \frac{1}{\sigma})$. Let X be the the number of iterations; we write its expectation in terms of the probability that the algorithm takes more than t iterations via the formula

$$\begin{aligned} E[X] &= \sum_{t=1}^{\infty} Pr[X \geq t] \\ &= \sum_{t=1}^{(n-1)!} Pr[X \geq t]. \end{aligned} \tag{23.1}$$

Note that the $(n-1)!$ represents the maximum possible number of iterations the algorithm can take, since each swap is strictly improving and hence no permutation can repeat.

To bound this probability, we use the simple fact that a large number of iterations t means there is an iteration where the decrease in weight was very small, and this is an event with low probability.

Lemma 23.1. *The probability of an iteration with improvement (i.e., decrease in cycle weight) in the range* $(0, \varepsilon]$ *is at most* $\frac{n^4 \varepsilon}{\sigma}$.

Proof. Fix a quadruplet (a, b, c, d). We upper-bound the probability that the $(0, \varepsilon]$ improvement was obtained by replacing $a \to b, c \to d$ with $a \to c, b \to d$. The decrease in weight resulting from this replacement is $-w_{bd} - w_{ac} + w_{ab} + w_{cd} \in (0, \varepsilon]$. By conditioning on w_{bd}, w_{ac}, w_{ab}, the last unconditioned value w_{cd} must lie in some interval $(L, L + \varepsilon]$. By the hypothesis on the distribution density, this can happen with probability at most $\frac{\varepsilon}{\sigma}$, leading us to conclude that for a fixed (a, b, c, d) the probability of such an event is at most $\frac{\varepsilon}{\sigma}$.

By union-bounding over all n^4 quadruplets (a, b, c, d) we can bound the probability by $\frac{n^4 \varepsilon}{\sigma}$. □

Lemma 23.2. *The probability that the algorithm takes at least t iterations is at most* $\frac{n^5}{t\sigma}$.

Proof. Note that $w_{ij} \in [0, 1]$ implies that the weight of any cycle is in $[0, n]$. This implies by pigeonhole principle that there was an iteration where the decrease in weight was at most $\varepsilon := \frac{n}{t}$. By Lemma 23.1 the probability of this event is at most $\frac{n^5}{t\sigma}$, as advertised. □

Theorem 23.3. *The expected number of iterations that the algorithm takes is at most* $O\left(\frac{n^6 \log n}{\sigma}\right)$.

Proof. Using Equation 23.1 and Lemma 23.2 we have:

$$\begin{aligned} E[X] &= \sum_{t=1}^{(n-1)!} Pr[X \geq t] \\ &\leq \sum_{t=1}^{(n-1)!} \frac{n^5}{t\sigma} \\ &= \frac{n^5}{\sigma} \sum_{t=1}^{(n-1)!} \frac{1}{t} \\ &= O\left(\frac{n^6 \log n}{\sigma}\right) \end{aligned}$$

Here we used the fact that $\sum_{t=1}^{N} \frac{1}{t} = O(\log N)$ and $O(\log(n-1)!) = O(n \log n)$. □

An improved bound of blah was given by details.

23.2 *The Simplex Algorithm*

We now turn our attention to the simplex algorithm for solving general linear programs. Given a vector $c \in \mathbb{R}^d$ and a matrix $A \in \mathbb{R}^{n \times d}$, this algorithm finds a solution variables $x = (x_1, \ldots, x_d)^\intercal$ that optimizes

$$\begin{aligned} \max \quad & c^\intercal x \\ & Ax \leq \mathbb{1}. \end{aligned}$$

This is a local-search algorithm, which iterates through solutions corresponding to vertices of the feasible solution polyhedron until it finds an optimal solution, or detects that the linear program is unbounded. It always goes from a solution to one of at least the same value, and there are situations where we have to make swaps that do not improve the value. Give example? In case there are many possible next moves, a ***pivot rule*** decides the next solution for the algorithm. Many pivot rules have been proposed. For most of them, we now know inputs where the simplex method iterates through an super-polynomial or exponential number of vertices, and thus has poor worst-case performance. (One widely used bad example is the Klee–Minty cube.) Moreover, there is no pivot rule for which we can prove a polynomial number of steps.

In fact, we don't know the answer to a simpler problem: if we imagine the polyhedron defining the feasible region as a graph, with the extreme points being vertices, and the edges of the polyhedron connecting them, is the diameter of this polytope polynomially bounded in the dimension d and the number of constraints n? As far as we know, the diameter could even be $O(n + d)$.

Despite this, the simplex method is widely used to solve linear programs. In their seminal paper, Dan Spielman and Shang-Hua Teng show that the simplex algorithm has a polynomial smoothed complexity for a specific pivot rule, the ***"shadow vertex pivot rule"***.

23.2.1 The Smoothing Model and Result

More precisely, they have shown that the simplex method with this pivot rule provides a polynomial algorithm for the following problem:

They consider a slightly more general problem, where the right hand side could be some $b \in \mathbb{R}^n$ instead of $\mathbb{1}$.

Input: vector $c \in \mathbb{R}^d$, matrix $A \in \mathbb{R}^{n \times d}$

Problem: For a random matrix $G \in \mathbb{R}^{n \times d}$ and a random vector $g \in \mathbb{R}^n$ where all entries are chosen independently from a Gaussian distribution $\mathcal{N}(0, \max_i ||a_i||^2)$, solve the following LP:

$$\begin{aligned} \max \quad & c^\intercal x \\ & (A + G)x \leq \mathbb{1} + g. \end{aligned}$$

This is one specific neighborhood model. Notice that for any input $(A, c, \mathbb{1})$, all inputs $(A + G, c, \mathbb{1}, g)$ are potential neighbors, but the probability decreases exponentially when we go 'further away' from the original input. The vector c is not changed, only A and $\mathbb{1}$ are jittered. The variance of the Gaussian distributions scales with the smoothness parameter σ. For $\sigma = 0$, the problem reduces to the standard linear programming problem and the analysis becomes a worst case analysis.

Notice that randomly jittering A and $\mathbb{1}$ means that the feasibility of the linear program can be switched (from feasible to infeasible or vice versa). Thus, jittering the instance and then solving the LP does not necessarily give any solution for the original LP. However, assuming that the input comes from an appropriate noisy source, the following theorem gives a polynomial running time bound.

Theorem 23.4. *The 'smoothed' number of simplex steps executed by the simplex algorithm with the shadow vertex pivot rule is bounded by* $\operatorname{poly}(n, d, 1/\sigma)$ *for the smoothed analysis model described above.*

The original result bounded the number of steps by $\mathcal{O}((nd/\sigma)^{O(1)})$, but the exponents were somewhat large. Roman Vershynin proved an improved bound of $\mathcal{O}(\log^7 n(d^9 + d^3/\sigma^4))$. An improved and simplified analysis due to Daniel Dadush and Sophie Huiberts gives a bound of $\approx d^{3.5}\sigma^{-2} \operatorname{poly}\log n$. More on this later. See this survey chapter by Daniel Dadush and Sophie Huiberts for many details.

23.2.2 The Shadow Vertex Pivot Rule

We conclude with an informal description of the shadow vertex pivot rule. Consider Figure 23.3. The three-dimensional polytope stands for the polyhedron of all feasible solutions, which is in general d-dimensional. The vector c points in the direction in which we want to maximize (it is plotted with an offset to emphasize the optimal vertex). The shadow pivot rule projects the polyhedron to a two-dimensional space spanned by c and the starting vertex u.

Assume that the original LP has an optimal solution with finite objective value. Then the polyhedron must be bounded in the direction of c. It is also bounded in the direction of u since the start vertex u is optimal for the direction u.

After projecting the polyhedron we intuitively follow the vertices that lie on the convex hull of the projection (moving towards the direction of c)[1]. Notice that the extreme vertex on the c-axis is the projection of an optimal vertex of the original polyhedron.

Since the polyhedron of all feasible solutions is not known, the projection cannot be done upfront. Instead, in each step, the algorithm projects the vectors to the neighbor vertices onto $\operatorname{span}\{c, u\}$ and identifies the neighbor which is the next on the convex hull.

A first step towards proving the result is to show that the shadow has a small number of vertices if the polyhedron $(A+G)x \leq (\mathbb{1}+g)$ is projected onto two *fixed* vectors u and c. The real result for simplex, however, is complicated by the fact that the vector u depends

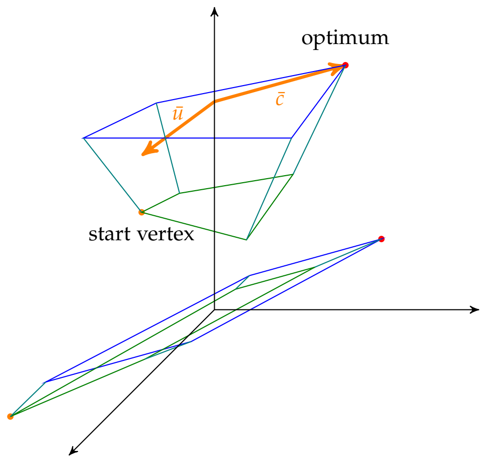


Figure 23.3: Illustration of the shadow vertex pivot rule.

[1] The two-dimensional projection is called the ***shadow*** of the original polyhedron, hence the name of the pivot rule.

on the polyhedron and on the starting vertex, and so the polyhedron is projected to a subspace that is correlated to the polyhedron itself. Another complication: finding a starting vertex is as hard as solving an LP. Spielman and Teng handle these and other issues; see the original publications.

23.3 *The Knapsack Problem*

Finally, we turn to a problem for which we will give (almost) all the details of the smoothed analysis. The input to the ***knapsack problem*** is a collection of n items, each with size/weight $w_i \in \mathbb{R}_{\geq 0}$ and profit $p_i \in \mathbb{R}_{\geq 0}$, and the goal is to pick a subset of items that maximizes the sum of profits, subject to the total weight of the subset being at most some bound B.

As always, we can write the inputs as vectors $p, w \in \mathbb{R}^n_{\geq 0}$, and hence the solution as $x \in \{0,1\}^n$. For example, with $p = (1,2,3)$, w = $(2,4,5)$ and $B = 10$, the optimal solution is $x = (0,1,1)$ with $p^\intercal x = 5$ and $w^\intercal x = 9$.

The knapsack problem is weakly NP-hard—e.g., if the sizes are integers, then it can be solved by dynamic programming in time $O(nB)$. Notice that perturbing the input by a real number prohibits the standard dynamic programming approach which assumes integral input. Therefore we show a smoothed analysis for a different algorithm, one due to George Nemhauser and Jeff Ullman. The smoothed analysis result is by René Beier, Heiko Röglin, and Berthold Vöcking.

23.3.1 *The Smoothing Model*

One natural neighborhood is to smooth each weight w_i uniformly over an interval of width σ centered at w_i. Figure 23.4 illustrates this. Within the interval, the density function is uniform, outside of the interval, it is zero. The profits are not perturbed.

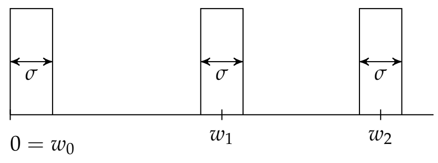


Figure 23.4: Distributions for three weights plotted into the same diagram.

We choose a slightly more general model already used earlier: The profits are fixed (chosen by the adversary), while the weights are chosen randomly and independently. The weight w_i is sampled from a distribution $f_i : [0,1] \to [0,1/\sigma]$, where the distribution can be different for different i and is chosen by the adversary. The important thing here is that the distribution is bounded by $\frac{1}{\sigma}$ and our complexity will polynomially depend on this value. Note that we made a simplifying assumption that weights are in $[0,1]$ to make our lives easier although it can be avoided. We also assume that it never happens that two solutions get exactly the same weight. This is justified since this is an event with zero probability, so it is almost surely not the case.

23.3.2 *The Nemhauser–Ullman algorithm*

The Nemhauser-Ullman algorithm for the knapsack problem computes the Pareto curve and returns the best solution from the curve. The ***Pareto curve*** for the knapsack problem can defined as follows.

Definition 23.5. Given two knapsack solutions $x_1, x_2 \in \{0,1\}^n$ we say that "*x_1 is dominated by x_2*" if $w^\intercal x_1 > w^\intercal x_2$ and $p^\intercal x_1 \leq p^\intercal x_2$. The *Pareto curve* is defined as the set of all non-dominated solutions (from a particular set of solutions). We also call the points on the curve *Pareto optimal*.

The definition is visualized in Figure 23.5. Note that the optimal solution is always on the Pareto curve. Moreover, if $P(j)$ is the collection of Pareto optimal solutions among the subinstance consisting of just the first j items, then

$$P(j+1) \subseteq P(j) \cup \underbrace{\{S \cup \{j+1\} \mid S \in P(j)\}}_{=:A_j}.$$

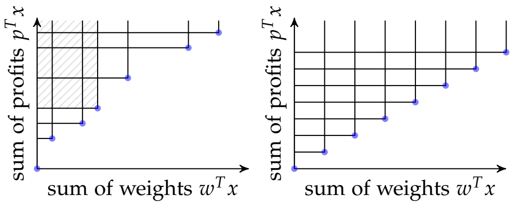


Figure 23.5: Left side: A point set and its Pareto curve. The blue points form the Pareto curve. They belong to the curve because the areas indicated by the black lines does not contain points. For the third point on the curve, this area is specifically highlighted. Right side: Worst Case.

The above containment implies that $P(j+1)$ can be computed from $P(j)$. In other words, a dominated solution by $P(j)$ will still be dominated in $P(j+1)$, so it can be safely forgotten.

If we keep the elements of $P(j)$ in sorted order with regard to the weights, then $P(j+1)$ can be computed in linear time by merging them together. Note that $P(j)$ and A_j are naturally constructed in increasing-weight order. Then we merge them in parallel (using the technique from merge-sort) and eliminate dominated points on the fly. The result is $P(j+1)$ with points stored in increasing-weight order. This technique leads to the following Lemma.

Lemma 23.6. *The Nemhauser-Ullman algorithm can be implemented to have a running time of* $O(\sum_{i=1}^{n} |P(j)|) = O(n \cdot \max_{j\in[n]} \mathbb{E}|P(j)|)$ *in expectation.*

Note that for this analysis we no longer care about the size of the knapsack g. The remainder of the section will be focused on bounding $\mathbb{E}[|P(j)|]$.

23.3.3 *Bounding Expected Size of Pareto Curve*

In the worst-case it is possible that all 2^n solutions are Pareto optimal. (Exercise: put $w_i = 2^i, p_i = 2^i$) In this case, the Nemhauser-Ullman algorithm will take exponential time. However, we will see that the expected size of $P(j)$ is polynomial for all $j \in [n]$ if the instance is σ-smoothed.

Fix any $j \in [n]$, and let $P := P(j)$. The proof idea is to partition the interval $[0, \infty]$ of **weights** into stripes of decreasing width $\varepsilon := 1/k$

for an integer k and then to count the number of Pareto optimal solutions in each stripe. See Figure 23.6. There is always an $\varepsilon > 0$ that is small enough such that each stripe contains at most one point from P (since we assumed no two solutions have the same weight). Denote by $P \cap [a,b]$ the set of all Pareto optimal solutions that have weight in the range $[a,b]$. Thus,

$$|P| = 1 + \lim_{k\to\infty} \sum_{\ell=0}^{\infty} \mathbb{1}\left(P \cap (\frac{\ell}{k}, \frac{\ell+1}{k}] \neq \emptyset\right).$$

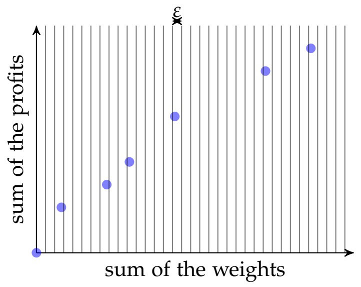


Figure 23.6: Dividing into stripes of width $\varepsilon = 1/k$.

We want to restrict the number of terms in the sum. Since the knapsack has size g, we can ignore all solutions that weight more than g. However, g might still be large. By our simplification, we know that all weights are at most 1. Thus, no solution can weight more than n and it is thus sufficient to consider the interval $[0,n]$. The kn stripes at $(0,1/k], (1/k,2/k], \dots, (n(k-1)/k, nk/k]$ fit into this interval. We thus get that

$$\mathbb{E}[|P|] \le 1 + \lim_{k\to\infty} \sum_{\ell=0}^{nk} \Pr\left(P \cap (\frac{\ell}{k}, \frac{\ell+1}{k}] \neq \emptyset\right) \tag{23.2}$$

Now we have to bound the probability that there is a point from P in the interval $(\frac{\ell}{k}, \frac{\ell+1}{k}]$. The following Lemma establishes this.

Lemma 23.7. *For any $t \ge 0$ it holds that* $\Pr(P \cap (t, t+\varepsilon] \neq \emptyset) \le \frac{n\varepsilon}{\sigma}$.

Proof. Define x^R to be the leftmost point right of t that is on the Pareto curve.

$$x^R := \begin{cases} \arg\min_{x\in P}\{p^\intercal x \mid w^\intercal x > t\} & \text{if the set is nonempty} \\ \perp & \text{else} \end{cases}$$

We define Δ to be the distance between t and x^R:

$$\Delta := \begin{cases} w^\intercal x^R - t & x^R \neq \perp \\ \infty & \text{otherwise} \end{cases}$$

(See Figure 23.7 for a visualization.) Clearly:

$$P \cap (t, t+\varepsilon] \neq \emptyset \quad \iff \quad \Delta \in (0, \varepsilon]$$

The rest of the proof shows that $\Pr(\Delta \in (0,\varepsilon])$ is small.

It is difficult to directly argue about Δ, so we use a set of auxiliary random variables which make the proof clean. For any item $i \in [n]$ we define several random variables: let $x^{UL,-i}$ be the most profitable (highest) point left of t without the item i; and let $x^{*,+i}$ be the leftmost (least weight) point that is higher than $x^{UL,-i}$ and contains i.

Formally:

$$x^{UL,-i} := \arg\max_{x \in 2^{[n]}} \{p^\intercal x \mid w^\intercal x \le t,\ x_i = 0\}$$
$$x^{*,+i} := \arg\min_{x \in 2^{[n]}} \{w^\intercal x \mid p^\intercal x \ge p^\intercal x^{UL,-i},\ x_i = 1\}$$

The reason for defining these variables is that (i) it is easy to bound the probability that $x^{*,+i}$ has weight within an interval $(t, t+\varepsilon]$ and (ii) we will later show that $x^R = x^{*,+i}$ for some i. With this reason in mind we define $\Delta^i = x^{*,+i} - t$ (∞ if undefined).

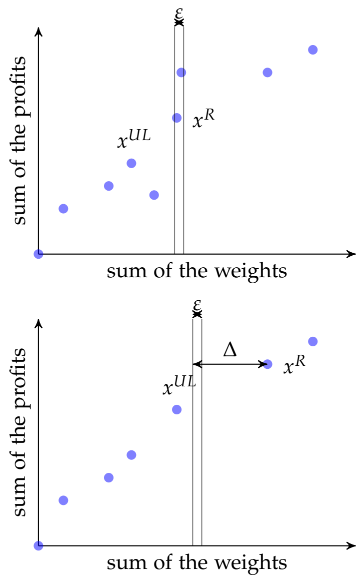


Figure 23.7: Illustration of the definition of x^{UL} and x^R. Δ is only plotted in the right figure.

Subclaim For any item $i \in [n]$ it holds that $\Pr\left[w^\intercal x^{*,+i} \in (t, t+\varepsilon]\right] \le \frac{\varepsilon}{\sigma}$. In particular, $\Pr\left[\Delta^i \in (0, \varepsilon]\right] \le \frac{\varepsilon}{\sigma}$.

Proof. Assume we fixed all weights except for i. Then $x^{UL,-i}$ is completely determined. We remind the reader that $x^{*,+i}$ is defined as the leftmost point that contains item i and is higher than $x^{UL,-i}$.

Now we turn our attention to the "precursor" of $x^{*,+i}$ without the item i, namely the item $x^{*,+i} - e_i$ where e_i is the i^{th} basis vector. The claim is that this point is completely determined when we fixed all weights except i. Name that point y (formal definition of this point will be given later). The point y is exactly the one that is leftmost with the condition that $y_i = 0$ and $p^\intercal y + p_i \ge p^\intercal x^{UL,-i}$ (by definition of $x^{*,+i}$). Note that the order of y's does not change when adding w_i. In particular, if y_1 was left of y_2 (had smaller weight), then adding the (currently undetermined) w_i will not change their ordering.

More formally, let $y := \arg\min_{y \in 2^{[n]}} \{w^\intercal y \mid p^\intercal y + p_i \ge p^\intercal x^{UL,-i},\ y_i = 0\}$ (we drop the index i from y for clarity). In other words, it is the leftmost solution without i higher than $x^{UL,-i}$ when we add the profit of i to it. It holds that $w^\intercal x^{*,+i} = w^\intercal y + w_i$. Therefore,

$$\begin{aligned}
\Pr\left[w^\intercal x^{*,+i} \in (t, t+\varepsilon]\right] &= \Pr\left[w^\intercal y + w_i \in (t, t+\varepsilon]\right] \\
&= \Pr\left[w_i \in (t - w^\intercal y, t + \varepsilon - w^\intercal y]\right] \\
&\le \frac{\varepsilon}{\sigma}
\end{aligned}$$

$\square$

Subclaim There exists $i \in [n]$ s.t. $x^R = x^{*,+i}$. In particular, $\exists i$ s.t. $\Delta = \Delta^i$.

Proof. Let x^{UL} be the most profitable (highest) point left of t. Formally:

$$x^{UL} := \arg\max_{x \in P} \{p^\intercal x \mid w^\intercal x \le t\}$$

Since the zero vector is always Pareto optimal there is always at least one point left of t, so x^{UL} is well-defined. There must be an item i s.t. that is contained in x^R, but is not in x^{UL}. Formally, pick and fix any $i \in [n]$ s.t. $x_i^R = 1, x_i^{UL} = 0$. It is clear that such i must exist since otherwise x^{UL} would have higher weight than x^R. Also, the height of x^R higher than x^{UL} since they are both on the Pareto curve.

Clearly (for this i) it holds that $x^{UL} = x^{UL,-i}$. Assume for the sake of contradiction that $x^{*,+i}$ is not x^R. Then:

- $x^{*,+i}$ must be left of x^R, otherwise we would have $x^{*,+i} = x^R$ and be done.
- $x^{*,+i}$ must be higher than x^{UL} by definition.
- $x^{*,+i}$ cannot be higher than x^R, otherwise x^R would not be on the Pareto curve (since it's left of it). So assume it's below x^R.
- $x^{*,+i}$ cannot be left of t, otherwise we would pick that point to be x^{UL} (since it's higher than x^{UL}).
- The only remaining spot for $x^{*,+i}$ is right of t and left of x^R, but that contradicts the choice of x^R as the leftmost point right of t.

Hence we conclude that $x^R = x^{*,+i}$ for our choice of i, which concludes the proof of the subclaim. □

Combining the above Subclaims, we get

$$\begin{aligned}\Pr[\Delta \in (0,\varepsilon]] &\le \sum_{i=1}^{n} \Pr\left[\Delta^i \in (0,\varepsilon]\right] \\ &= n\frac{\varepsilon}{\sigma}.\end{aligned}$$

This is equivalent to the statement of the Lemma, hence we are done. □

Using the above analysis, we conclude with a smoothness theorem.

Theorem 23.8. *For σ-smoothed instances, the expected size of P is bounded by n^2/σ for all $j \in [n]$. In particular, the Nemhauser-Ullman algorithm for the knapsack problem has a smoothed complexity of $O(n^3/\sigma)$ for the smoothness model described in Subsection 23.3.1.*

Proof. By Inequality (23.2), Lemma 23.6 and Lemma 23.7, we conclude that

$$\begin{aligned}\mathbb{E}[|P|] &\le 1 + \lim_{k\to\infty} \textstyle\sum_{\ell=0}^{nk} \Pr\left(P \cap \left(\frac{\ell}{k}, \frac{\ell+1}{k}\right] \neq \emptyset\right) \\ &\le 1 + \lim_{k\to\infty} nk \cdot \frac{n\frac{1}{k}}{\sigma} = \frac{n^2}{\sigma}.\end{aligned}$$

□

23.3.4 *More general result*

The result can be extended beyond the knapsack problem. Let Π be a "combinatorial optimization" problem given as

$$\begin{aligned} \max\ & p^\intercal n x \\ s.t.\ & Ax \leq b \\ & x \in \mathcal{S} \end{aligned}$$

where $\mathcal{S} \subseteq \{0,1\}^n$. Observe: the knapsack problem is one such problem. Beier and Vöcking prove the following theorem.

Theorem 23.9. *Problem Π has polynomial smoothed complexity if solving Π on unitarily encoded instances can be done in polynomial time.*

For the knapsack problem, the dynamic programming algorithm has a running time of $O(ng)$, where g is the size of the knapsack. If the instance is encoded in binary, then we need $O(n \log g)$ bits for the input and hence this algorithm is not polynomial-time; however if input is encoded in unary then we use $O(ng)$ to encode the instance, and the dynamic programming algorithm becomes polynomial-time.

24
Prophets and Secretaries

The *prophet* and *secretary* problems are two classes of problems where online decision-making meets stochasticity: in the first set of problems the inputs are random variables, whereas in the second one the inputs are worst-case but revealed to the algorithm (a.k.a. the decision-maker) in random order. Here we survey some results, proofs, and techniques, and give some pointers to the rich body of work developing around them.

24.1 The Prophet Problem

The problem setting: there are n random variables $X_1, X_2, \dots, X_n$. We know their distributions up-front, but not their realizations. These realizations are revealed one-by-one (say in the order $1, 2, \dots, n$). We want to give a strategy (which is a stopping rule) that, upon seeing the value X_i (and all the values before it) decides either to *choose* i, in which case we get reward X_i and the process stops. Or we can *pass*, in which case we move on to the next items, and are not allowed to come back to i ever again. We want to maximize our expected reward. If

$$X_{\max} := \max\{X_1, X_2, \dots, X_n\},$$

it is clear that our expected reward cannot exceed $\mathbb{E}[X_{\max}]$. But how close can we get?

If we want to find the best strategy, and we know the order in which we are shown these r.v.s, there is dynamic programming algorithm. (Exercise!)

In fact, we may be off by a factor of almost two against this yardstick in some cases: suppose $X_1 = 1$ surely, whereas $X_2 = 1/\varepsilon$ with probability ε, and 0 otherwise. Any strategy either picks 1 or passes on it, and hence gets expected value 1, whereas $\mathbb{E}[X_{\max}] = (2 - \varepsilon)$. Surprisingly, this is the worst case.

Theorem 24.1 (Krengel, Sucheston, and Garling)**.** *There is a strategy with expected reward* $1/2\,\mathbb{E}[X_{\max}]$.

Such a result, that gives a stopping rule whose value is comparable to the $\mathbb{E}[X_{\max}]$ is called a *prophet inequality*, the idea being that one can come close to the performance of a prophet who is clairvoyant, can see the future. The result in Theorem 24.1 was proved by Krengel, Sucheston, and Garling; several proofs have been given since. Apart from being a useful algorithmic construct, the prophet inequality naturally fits into work on algorithmic auction design: suppose you know that n potential are interested in an item with valuations $X_1, \ldots, X_n$, and you want to sell to one person: how do you make sure your revenue is close to $\mathbb{E}[X_{\max}]$?

We now give three proofs of this theorem. For the moment, let us ignore computational considerations, and just talk about the information theoretic issues.

24.1.1 *The Median Algorithm*

This proof is due to Ester Samuel-Cahn.

Let τ be the median of the distribution of $X_{\max}$: i.e.,

$$\Pr[X_{\max} \geq \tau] = 1/2.$$

(For simplicity we assume that there is no point mass at τ, the proof is easily extended to discrete distributions too.) Now the strategy is simple: *pick the first X_i which exceeds τ*. We claim this prove Theorem 24.1.

Proof. Observe that we pick an item with probability exactly 1/2, but how does the expected reward compare with $\mathbb{E}[X_{\max}]$?

$$\begin{aligned} \mathbb{E}[X_{\max}] &\leq \tau + \mathbb{E}[(X_{\max} - \tau)^+]) \\ &\leq \tau + \mathbb{E}\left[\sum_{i=1}^{n} (X_i - \tau)^+\right]. \end{aligned}$$

And what does the algorithm get?

$$\begin{aligned} ALG &\geq \tau \cdot \Pr[X_{\max} \geq \tau] + \sum_{i=1}^{n} \mathbb{E}[(X_i - \tau)^+] \cdot \Pr[\bigwedge_{j \leq i} (X_j < \tau)] \\ &\geq \tau \cdot \Pr[X_{\max} \geq \tau] + \sum_{i=1}^{n} \mathbb{E}[(X_i - \tau)^+] \cdot \Pr[X_{\max} < \tau] \end{aligned}$$

But both these probability values equal half, and hence $ALG \geq$ 1/2 $\mathbb{E}[X_{\max}]$. □

While a beautiful proof, it is somewhat mysterious, and difficult to generalize. Indeed, suppose we are allowed to choose up to k variables to maximize the sum of their realizations? The above proof seems difficult to generalize, but the following LP-based one will.

However, a recent paper of Shuchi Chawla, Nikhil Devanur, and Thodoris Lykouris gives an extension of Samuel-Cahn's proof to the multiple item setting.

24.1.2 *The LP-based Algorithm*

The second proof is due to Shuchi Chawla, Jason Hartline, David Malec, and Balu Sivan, and to Saeed Alaei. Define p_i as the probability that element X_i takes on the highest value. Hence $\sum_i p_i = 1$. Moreover, suppose τ_i is such that $\Pr[X_i \geq \tau_i] = p_i$, i.e., the p_i^{th} percentile for X_i. Define

If you know of an earlier reference for this proof, please let me know.

$$v_i(p_i) := \mathbb{E}[X_i \mid X_i \geq \tau_i]$$

as the value of X_i conditioned on it lying in the top p_i^{th} percentile. Clearly, $\mathbb{E}[X_{\max}] \leq \sum_i v_i(p_i) \cdot p_i$. Here's an algorithm that gets value $1/4 \sum_i v_i(p_i) \cdot p_i \geq 1/4\, \mathbb{E}[X_{\max}]$:

If we have not chosen an item among $1, \dots, i-1$, when looking at item i, pass with probability half without even looking at X_i, else accept it if $X_i \geq \tau_i$.

Lemma 24.2. *The algorithm achieves a value of* $1/4\mathbb{E}[X_{\max}]$.

Proof. Say we "reach" item i if we've not picked an item before i. The expected value of the algorithm is

$$\begin{aligned} ALG &\geq \sum_{i=1}^{n} \Pr[\text{reach item } i] \cdot 1/2 \cdot \Pr[X_i \geq \tau_i] \cdot \mathbb{E}[X_i \mid X_i \geq \tau_i] \\ &= \sum_{i=1}^{n} \Pr[\text{reach item } i] \cdot 1/2 \cdot p_i \cdot v_i(p_i). \end{aligned} \tag{24.1}$$

Since we pick each item with probability $p_i/2$, the expected number of items we choose is half. So, by Markov's inequality, the probability we pick no item at all is at least half. Hence, the probability we reach item i is at least one half too, the above expression is $1/4 \sum_i v_i(p_i) \cdot p_i$ as claimed. □

Now to give a better bound, we refine the algorithm above: suppose we denote the probability of reaching item i by r_i, and suppose we reject item i outright with probability $1 - q_i$. Then (24.1) really shows that

$$ALG \geq \sum_{i=1}^{n} r_i \cdot q_i \cdot p_i \cdot v_i(p_i).$$

Above, we ensured that $q_i = r_i = 1/2$, and hence lost a factor of $1/4$. But if we could ensure that $r_i \cdot q_i = 1/2$, we'd get the desired result of $1/2\mathbb{E}[X_{\max}]$. For the first item $r_1 = 1$ and hence we can set $q_1 = 1/2$. What about future items? Note that since that

$$r_{i+1} = r_i(1 - q_i \cdot p_i) \tag{24.2}$$

we can just set $q_{i+1} = \frac{1}{2r_{i+1}}$. A simple induction shows that $r_{i+1} \geq 1/2$—indeed, sum up (24.2) to get $r_{i+1} = r_1 - \sum_{j \leq i} p_i/2$—so that $q_{i+1} \in [0, 1]$ and is well-defined.

24.1.3 *The Computational Aspect*

If the distribution of the r.v.s X_i is explicitly given, the proofs above immediately give us algorithms. However, what if the variables X_i are given via a black-box that we can only access via samples?

The first proof merely relies on finding the median: in fact, finding an "approximate median" $\widehat{\tau}$ such that $\Pr[X_{\max} < \widehat{\tau}] \in (1/2 - \varepsilon, 1/2 + \varepsilon)$ gives us expected reward $1/2{+}\varepsilon/2\mathbb{E}[X_{\max}]$. To do this, sample from $X_{\max}$ $O(\varepsilon^{-2} \log \delta^{-1})$ times (each sample to $X_{\max}$ requires one sample to each of the X_is) and take $\widehat{\tau}$ to be the sample median. A Hoeffding bound shows that $\widehat{\tau}$ is an "approximate median" with probability at least $1 - \delta$.

For the second proof, there are two ways of making it algorithmic. Firstly, if the quantities are polynomially bounded, estimate p_i and v_i by sampling. Alternatively, solve the convex program

$$\max \left\{ \sum_i y_i \cdot v_i(y_i) \mid \sum_i y_i = 1 \right\}$$

and use the y_i's from its solution in lieu of p_i's in the algorithm above.

Do we need such good approximations? Indeed, getting samples from the distributions may be expensive, so how few samples can we get away with? A paper of Pablo Azar, Bobby Kleinberg, and Matt Weinberg shows how to get a constant fraction of $\mathbb{E}[X_{\max}]$ via taking just one sample from each of the X_is. Let us a give a different algorithm, by Aviad Rubinstein, Jack Wang, and Matt Weinberg.

24.1.4 *The One-Sample Algorithm*

For the preprocessing, take one sample from the distributions for each of $X_1, X_2, \ldots, X_n$. (Call these samples $S_1, S_2, \ldots, S_n$.) Set the threshold τ to be the largest of these sample values. Now when seeing the actual items $X_1, X_2, \ldots, X_n$, pick the first item higher than τ. We claim this one-sample algorithm gives an expected value at least $1/2\, \mathbb{E}[X_{\max}]$.

Proof. As a thought experiment, consider taking two independent samples from each distribution, then flipping a coin C_i to decide which is X_i and which is S_i. This has the same distribution as original process, so we consider this perspective.

Now consider all these $2n$ values together, in sorted order: call these $W_1 \geq W_2 \geq \ldots \geq W_{2n}$. We say W_j has *index i* if it is drawn from the i^{th} distribution, and hence equal to X_i or S_i. Let j^* be the smallest position where W_i, W_{j^*+1} have the same index for some $i \leq j^*$. Observe: the coins C_i for the indices corresponding to the first

j^* positions are independent, and the coin for the position j^*+1 is the same as one of the previous ones. We claim that

$$\mathbb{E}[X_{\max}] = \sum_{i \leq j^*} \frac{W_i}{2^i} + \frac{W_{j^*+1}}{2^{j^*}}.$$

Indeed, $X_{\max} = W_i$ if all the previous $W_{i'}$s belong to the sample (i.e., they are S's and not X's), but W_i belongs to the actual values (it is an X). Moreover, if all the previous values are Ss, then W_{j^*+1} would be an X and hence the maximum, by our choice of j^*.

What about the algorithm? If W_1 is a sample (i.e., an S-value) then we don't get any value. Else if $W_1, \ldots, W_i$ are all X values, and W_{i+1} is a sample (S value) then we get value at least W_i. If $i < j^*$, this happens with probability $\frac{1}{2^{i+1}}$ since all the $i+1$ coins are independent. Else if $i = j^*$, the probability is $\frac{1}{2^i} = \frac{1}{2^{j^*}}$. Hence

$$\text{Alg} \geq \sum_{i<j^*} \frac{W_i}{2^{i+1}} + \frac{W_{j^*}}{2^{j^*}} \geq \sum_{i<j^*} \frac{W_i}{2^{i+1}} + \frac{W_{j^*}}{2^{j^*+1}} + \frac{W_{j^*+1}}{2^{j^*+1}}.$$

But this is at least half of $\mathbb{E}[X_{\max}]$, which proves the theorem. □

24.1.5 *Extensions: Picking Multiple Items*

What about the case where we are allowed to choose k variables from among the n? Proof #2 generalizes quite seamlessly. If p_i is the probability that X_i is among the top k values, we now have:

$$\textstyle\sum_i p_i = k. \tag{24.3}$$

The "upper bound" on our quantity of interest remains essentially unchanged:

$$\mathbb{E}[\text{sum of top } k \text{ r.v.s}] \leq \textstyle\sum_i v_i(p_i) \cdot p_i. \tag{24.4}$$

What about an algorithm to get value $1/4$ of the value in (24.4)? The same as above: reject each item outright with probability $1/2$, else pick i if $X_i \geq \tau_i$. Proof #2 goes through unchanged.

For this case of picking multiple items, we can do much better: a result of Alaei shows that one can get within $(1 - 1/\sqrt{k+3})$ of the value in (24.4)—for $k = 1$, this matches the factor $1/2$ we showed above. One can, however, get a factor of $(1 - O(\sqrt{\frac{\log k}{k}}))$ using a simple concentration bound.

Proof. Suppose we reduce the rejection probability to δ. Then the probability that we reach some item i without having picked k items already is lower-bounded by the probability that we pick at most k elements in the entire process. Since we reject items with probability

δ, the expected number of elements we pick is $(1-\delta)k$. Hence, the probability that we pick less than k items is at least $1 - e^{-\Theta(\delta^2 k)}$, by a Hoeffding bound for sums of independent random variables. Now setting $\delta = O(\sqrt{\frac{\log k}{k}})$ ensures that the probability of reaching each item is at least $(1-\frac{1}{k})$, and an argument similar to that in Proof #2 shows that

$$ALG \geq \sum_{i=1}^{n} \Pr[\text{reach item } i] \cdot \Pr[\text{not reject item } i] \cdot \Pr[X_i \geq \tau_i] \cdot \mathbb{E}[X_i \mid X_i \geq \tau_i]$$
$$= \sum_{i=1}^{n} (1 - 1/k) \cdot (1 - O(\sqrt{\tfrac{\log k}{k}})) \cdot p_i \cdot v_i(p_i),$$

which gives the claimed bound of $(1 - O(\sqrt{\frac{\log k}{k}}))$. □

24.1.6 *Extensions: Matroid Constraints*

Suppose there is a matroid structure $\mathcal{M}$ with ground set $[n]$, and the set of random variables we choose must be independent in this matroid $\mathcal{M}$. The value of the set is the sum of the values of items within it. (Hence, the case of at most k items above corresponds to the *uniform* matroid of rank k.) The goal is to make the expected value of the set picked by the algorithm close to the expected value of the max-weight independent set.

Recall that a ***matroid*** $\mathcal{M} = (U, \mathcal{F})$ is a set U is a collection of subsets $\mathcal{F} \subseteq 2^U$ that is closed under taking subsets, such that if $A, B \in \mathcal{F}$ and $|A| < |B|$ then there exists $b \in B \setminus A$ such that $A \cup \{b\} \in \mathcal{F}$. Sets in $\mathcal{F}$ are called *independent sets*.

Bobby Kleinberg and Matt Weinberg give an algorithm to picks an independent set whose expected value is at least half the value of the max-weight independent set, thereby extending the original single-item prophet inequality seamlessly to all matroids. While their original proof uses a combinatorial idea, a LP-based proof was subsequently given by Moran Feldman, Ola Svensson, and Rico Zenklusen. The idea is again clever and conceptually clean: find a solution y to the convex program

$$\sum_i v_i(y_i) \cdot y_i.$$
$$y \in \text{the matroid polytope for } \mathcal{M}$$

Now given a fractional point y in the matroid polytope, how to get an integer point (i.e., an independent set). For this they give an approach called an "online contention resolution" scheme that ensures that any item i is picked with probability at least $\Omega(y_i)$, much like in the single-item and k-item cases.

There are many other extensions to prophet inequalities: people have studied more general constraint sets, submodular valuations instead of just additive valuations, what if the order of items is not known, what if we are allowed to choose the order, etc. See papers on arXiv, or in the latest conferences for much more.

24.1.7 Exercises

1. Give a dynamic programming algorithm for the best strategy when we know the order in which r.v.s are revealed to us. (Footnote 1). Extend this to the case where you can pick k items.

   Open problem: is this "best strategy" problem computationally hard when we are given a general matroid constraint? Even a laminar matroid or graphical matroid?

2. If we can choose the order in which we see the items, show that we can get expected value $\geq (1 - 1/e)\mathbb{E}[X_{\max}]$. (Hint: use proof #2, but consider the elements in decreasing order of $v_i(p_i)$.)

   Open problem: can you beat $(1 - 1/e)\mathbb{E}[X_{\max}]$? A recent paper of Abolhassani et al. does so for i.i.d. X_is.

24.2 Secretary Problems

The problem setting: there are n items, each having some intrinsic non-negative value. For simplicity, assume the values are distinct, but we know nothing about their ranges. We know n, and nothing else. The items are presented to us one-by-one. Upon seeing an item, we can either pick it (in which case the process ends) or we can pass (but then this item is rejected and we cannot ever pick it again). The goal is to maximize the probability of picking the item with the largest value $v_{\max}$.

If an adversary chooses the order in which the items are presented, every deterministic strategy must fail. Suppose there are just two items, the first one with value 1. If the algorithm picks it, the adversary can send a second item with value 2, else it sends one with value $1/2$. Randomizing our algorithm can help, but we cannot do much better than $1/n$.

So the secretary problem asks: *what if the items are presented in uniformly random order?* For this setting, it seems somewhat surprising at first glance that one can pick the best item with probability at least a constant (knowing nothing other than n, and the promise of a uniformly random order). Indeed, here a simple algorithm and proof showing a probability of $1/4$:

> Ignore the first $n/2$ items, and then pick the next item that is better than all the ones seen so far.

Note that this algorithm succeeds if the best item is in the second half of the items (which happens w.p. $1/2$) and the second-best item is in the first half (which, conditioned on the above event, happens w.p. $\geq 1/2$). Hence $1/4$. It turns out that rejecting the first half of the items is not optimal, and there are other cases where the algorithm succeeds that this simple analysis does not account for, so let's be more careful. Consider the following *37%-algorithm*:

> Ignore the first n/e items, and then pick the next item that is better than all the ones seen so far.

Theorem 24.3. *As $n \to \infty$, the 37%-algorithm picks the highest number with probability at least $1/e$. Hence, it gets expected value at least $v_{\max}/e$. Moreover, n/e is the optimal choice of m among all wait-and-pick algorithms.*

Proof. Call a number a *prefix-maximum* if it is the largest among the numbers revealed before it. Notice being the maximum is a property of just the set of numbers, whereas being a prefix-maximum is a property of the random sequence and the current position. If we pick the first prefix-maximum after rejecting the first m numbers, the probability we pick the maximum is

$$\sum_{t=m+1}^{n} \Pr[v_t \text{ is max}] \cdot \Pr[\text{max among first } t-1 \text{ numbers falls in first } m \text{ positions}]$$

$$\overset{(\star)}{=} \sum_{t=m+1}^{n} \frac{1}{n} \cdot \frac{m}{t-1} \quad = \quad \frac{m}{n}\big(H_{n-1} - H_{m-1}\big),$$

where $H_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}$ is the k^{th} harmonic number. The equality $(\star)$ uses the uniform random order. Now using the approximation $H_k \approx \ln k + 0.57$ for large k, we get the probability of picking the maximum is about $\frac{m}{n} \ln \frac{n-1}{m-1}$ when m, n are large. This quantity has a maximum value of $1/e$ if we choose $m = n/e$. □

Next we show we can replace any strategy (in a comparison-based model) with a wait-and-pick strategy without decreasing the probability of picking the maximum.

Theorem 24.4. *The strategy that maximizes the probability of picking the highest number can be assumed to be a wait-and-pick strategy.*

Proof. Think of yourself as a player trying to maximize the probability of picking the maximum number. Clearly, you should reject the next number v_i if it is not prefix-maximum. Otherwise, you should pick v_i only if it is prefix-maximum and the probability of v_i being the maximum is more than the probability of you picking the maximum in the remaining sequence. Let us calculate these probabilities.

We use Pmax to abbreviate "prefix-maximum". For position $i \in \{1, \ldots, n\}$, define

$$f(i) = \Pr[v_i \text{ is max} \mid v_i \text{ is Pmax}] \overset{(\star)}{=} \frac{\Pr[v_i \text{ is max}]}{\Pr[v_i \text{ is Pmax}]} \overset{(\star\star)}{=} \frac{1/n}{1/i} = \frac{i}{n},$$

where equality $(\star)$ uses that the maximum is also a prefix-maximum, and $(\star\star)$ uses the uniform random ordering. Note that $f(i)$ increases with i.

Now consider a problem where the numbers are again being revealed in a random order but we must reject the first i numbers. The goal is to still maximize the probability of picking the highest of the n numbers. Let $g(i)$ denote the probability that the optimal strategy for this problem picks the global maximum.

The function $g(i)$ must be a non-increasing function of i, else we could just ignore the $(i+1)^{st}$ number and set $g(i)$ to mimic the strategy for $g(i+1)$. Moreover, $f(i)$ is increasing. So from the discussion above, you should not pick a prefix-maximum number at any position i where $f(i) < g(i)$ since you can do better on the suffix. Moreover, when $f(i) \geq g(i)$, you should pick v_i if it is prefix-maximum, since it is worse to wait. Therefore, the approach of waiting until f becomes greater than g and thereafter picking the first prefix-maximum is an optimal strategy. □

In keeping with the theme of this chapter, we now give an alternate proof that uses a convex-programming view of the process. We will write down an LP that captures some properties of any feasible solution, optimize this LP and show a strategy whose success probability is comparable to the objective of this LP! The advantage of this approach is that it then extends to adding other constraints to the problem.

Proof. (Due to Niv Buchbinder, Kamal Jain, and Mohit Singh.) Let us fix an optimal strategy. By the first proof above, we know what it is, but let us ignore that for the time being. Let us just assume w.l.o.g. that it does not pick any item that is not the best so far (since such an item cannot be the global best).

Let p_i be the probability that this strategy picks an item at position i. Let q_i be the probability that we pick an item at position i, *conditioned on it being the best so far*. So $q_i = \frac{p_i}{1/i} = i \cdot p_i$.

Now, the probability of picking the best item is

$$\begin{aligned}
&\sum_i \Pr[i^{th} \text{ position is global best and we pick it }] \\
&= \sum_i \Pr[i^{th} \text{ position is global best }] \cdot q_i = \sum_i \frac{1}{n} q_i = \sum_i \frac{i}{n} p_i. \qquad (24.5)
\end{aligned}$$

What are the constraints? Clearly $p_i \in [0,1]$. But also

$$\begin{aligned}
p_i &= \Pr[\text{ pick item } i \mid i \text{ best so far}] \cdot \Pr[i \text{ best so far}] \\
&\leq \Pr[\text{ did not pick } 1, \ldots, i-1 \mid i \text{ best so far}] \cdot (1/i) \qquad (24.6)
\end{aligned}$$

But not picking the first $i-1$ items is independent of i being the best so far, so we get

$$p_i \leq \frac{1}{i}\Big(1 - \sum_{j<i} p_j\Big).$$

Hence, the success probability of any strategy (and hence of the optimal strategy) is upper-bounded by the following LP in variables p_i:

$$\max \sum_i \frac{i}{n} \cdot p_i$$
$$i \cdot p_i \leq 1 - \sum_{j<i} p_j$$
$$p_i \in [0,1].$$

Now it can be checked that the solution $p_i = 0$ for $i \leq \tau$ and $p_i \frac{\tau}{n}(\frac{1}{i-1} - \frac{1}{i})$ for $\tau \leq i \leq n$ is a feasible solution, where τ is defined by the smallest value such that $H_{n-1} - H_{\tau-1} \leq 1$. (By duality, we can also show it is optimal!)

Finally we can get a stopping strategy whose success probability matches that of the LP. Indeed, solve the LP. Now, for the i^{th} position if we've not picked an item already and if this item is the best so far, pick it with probability $\frac{ip_i}{1-\sum_{j<i} p_j}$. By the LP constraint, this probability $\in [0,1]$. Moreover, removing the conditioning shows we pick an item at location i with probability p_i, and a calculation similar to the one above shows that our algorithm's success probability is $\sum_i ip_i/n$, the same as the LP. □

24.2.1 *Extension: Game-Theoretic Issues*

Note that in the optimal strategy, we don't pick any items in the first n/e timesteps, and then we pick items with quite varying probabilities. If the items are people interviewing for a job, this gives them an incentive to not come early in the order. Suppose we insist that for each position i, the probability of picking the item at position i is the same. What can we do then?

Let's fix any such strategy, and write an LP capturing the success probabilities of this strategy with uniformity condition as a constraint. Suppose $p \leq 1/n$ is this uniform probability (over the randomness of the input sequence). Again, let q_i be the probability of picking an item at position i, conditioned on it being the best so far. Note that we may pick items even if they are not the best so far, just to satisfy the uniformity condition; hence instead of $q_i = i \cdot p$ as before, we have

$$q_i \leq ip.$$

Moreover, by the same argument as (24.6), we know that

$$q_i \leq 1 - (i-1)p.$$

And the strategy's success probability is again $\sum q_i/n$ using (24.5). So we can now solve the LP

$$\max \sum_i \frac{1}{n} \cdot q_i$$
$$q_i \leq 1 - (i-1) \cdot p$$
$$q_i \leq i \cdot p$$
$$q_i \in [0,1], p \geq 0$$

Now the Buchbinder, Jain, and Singh paper shows the optimal value of this LP is at least $1 - 1/\sqrt{2} \approx 0.29$; they also give a slightly more involved algorithm that achieves this success probability.

24.2.2 *Extension: Multiple Items*

Now back to having no restrictions on the item values. Suppose we want to pick k items, and want to maximize the expected sum of these k values. Suppose the set of the k largest values is $S^\star \subseteq [n]$, and their total value is $V^\star = \sum_{i \in S} v_i$. It is easy to get an algorithm with expected value $\Omega(V^\star)$. E.g., split the n items into k groups of n/k items, and run the single-item algorithm separately on each of these. (Why?) Or ignore the first half of the elements, look at the value $\hat{v}$ of the $(1-\varepsilon)k/2^{th}$ highest value item in this set, and pick all items in the second half with values greater than $\hat{v}$. And indeed, ignoring half the items must lose a constant factor in expectation.

But here's an algorithm that gives value $V^\star(1-\delta)$ where $\delta \to 0$ as $k \to \infty$. We will set $\delta = O(k^{-1/3} \log k)$ and $\varepsilon = \delta/2$. Ignore the first δn items. (We expect $\delta k \approx k^{2/3}$ items from $S^\star$ fall in this ignored set.) Now look at the value $\hat{v}$ of the $(1-\varepsilon)\delta k^{th}$-highest valued item in this ignored set, and pick the first (at most) k elements with values greater than $\hat{v}$ along the remaining $(1-\delta)n$ elements.

Why is this algorithm good? There are two failure modes: (i) if $v' = \min_{i \in S^\star} v_i$ be the lowest value item we care about, then we don't want $\hat{v} \leq v'$ else we may pick low valued items, and (ii) we want the number of items from $S^\star$ in the last $(1-\delta)n$ and greater than $\hat{v}$ to be close to k.

Let's sketch why both bad events happen rarely. For event (i) to happen, fewer than $(1-\varepsilon)\delta k$ items from $S^\star$ fall in the first δn locations: i.e., their number is less than $(1-\varepsilon)$ times its expectation, which has probability $\exp(-\varepsilon^2 \delta k) = 1/\operatorname{poly}(k)$ by a Hoeffding bound. For event (ii) to happen, more than $(1-\varepsilon)\delta k$ of the top $(1-\delta)k$ items from $S^\star$ fall among the ignored items. This means their number exceeds $(1+O(\varepsilon))$ times its expectation, which again has probability $\exp(-\varepsilon^2 \delta k) = 1/\operatorname{poly}(k)$.

An aside: the standard concentration bounds we know are for sums of i.i.d. r.v.s whereas the random order model causes correlations. The easiest way to handle that is to ignore not the first δn items but a random number of items $\sim \operatorname{Bin}(n, \delta)$. Then each item has probability δ of being ignored, independent of others.

Is this tradeoff optimal? No. Kleinberg showed that one can get expected value $V^\star(1 - O(k^{-1/2}))$, and this is asymptotically optimal.

In fact, one can extend this even further: a set of vectors $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n \in [0,1]^m$ is fixed, along with values $v_1, v_2, \dots, v_n$. These are initially unknown. Now they are revealed one-by-one to the algorithm in a uniformly random order. The algorithm, on seeing a vector and its value must decide to pick it or irrevocably reject it. It can pick as many vectors as it wants, subject to their sum being at most k in each coordinate; the goal is to maximize the expected value of the picked vectors. The k-secretary case is the 1-dimensional case when each $\mathbf{a}_i = (1)$. Indeed, this is the problem of solving a packing linear program online, where the columns arrive in random order. A series of works have extended the k-secretary case to this online packing LP problem, getting values which are $(1 - O(\sqrt{(\log m)/k}))$ times the optimal value of the LP.

24.2.3 *Extension: Matroids*

One of the most tantalizing generalizations of the secretary problem is to matroids. Suppose the n elements form the ground set of a matroid, and the elements we pick must form an independent set in this matroid. Babioff, Immorlica, and Kleinberg asked: if the max-weight independent set has value V^*, can we get $\Omega(V^*)$ using an online algorithm? The current best algorithms, due to Lachish, and to Feldman, Svensson, and Zenklusen, achieve expected value $\Omega(V^* / \log\log k)$, where k is the rank of the matroid. Can we improve this further, say to a constant? A constant factor is known for many classes of matroids, like graphical matroids, laminar matroids, transversal matroids, and gammoids.

24.2.4 *Other Random Arrival Models*

One can consider other models for items arriving online: say a set of n items (and their values) is fixed by an adversary, and each timestep we see one of these items sampled uniformly *with replacement*. (The random order model is same, but without replacement.) This model, called the *i.i.d. model*, has been studied extensively—results in this model are often easier than in the random order model (due to lack of correlations). See, e.g., references in a monograph by Aranyak Mehta.

Do we need the order of items to be uniformly random, or would weaker assumptions suffice? Kesselhiem, Kleinberg, and Niazadeh consider this question in a very nice paper and show that much less independence is enough for many of these results to hold [1].

[1]

In general the random-order model is a clean way of modeling the fact that an online stream of data may not be adversarially ordered. Many papers in online algorithms have used this model to give better results than in the worst-case model: some of my favorite ones are paper of Meyerson on facility location, and this paper of Bahmani, Chowdhury, and Goel on computing PageRank incrementally.

Again, see online for many many papers related to the secretary problem: numerous models, different constraints on what sets of items you can pick, and how you measure the quality of the picked set. It's a very clean model, and can be used in many different settings.

Exercises

1. Give an algorithm for general matroids that finds an independent set with expected value at least an $O(1/(\log k))$-fraction of the max-value independent set.
2. Improve the above result to $O(1)$-fraction for graphic matroids.