Approximation Algorithms

While we have good algorithms for many optimization problems, the unfortunate reality elucidated by theoretical computer science is that so very many important real-world optimization problems are **NP**-hard. What do we do? Suppose we are given an **NP**-hard problem to solve. Assuming $\mathbf{P} \neq \mathbf{NP}$, we can't hope for a polynomial-time algorithm for these problems. But can we get polynomial-time algorithms that always produce a "pretty good" solution? (a.k.a. *approximation algorithms*)

Objectives of this lecture

In this lecture, we will

- define and motivate **approximation algorithms**
- derive two **greedy algorithms** for scheduling jobs on multiple machines to minimize their makespan (a.k.a. stacking blocks to minimize the height of the tallest stack)
- see how **rounding linear programs** can give us an approximate vertex cover
- show how **scaling** can be used to turn pseudopolynomial-time algorithms into efficient approximation algorithms.

1 Introduction

Given an **NP**-hard problem, we don't hope for a fast algorithm that always gets the optimal solution — if we had such a polynomial algorithm, we would be able to use it to solve everything in NP, and that would imply that $\mathbf{P} = \mathbf{NP}$, something we expect is false. But when faced with an **NP**-hard problem, giving up is not the only reasonable solution! There are several ways that we might try to move forward

- First approach: find a polynomial-time algorithm that guarantees to get at least a "pretty good" solution? E.g., can we guarantee to find a solution that's within 10% of optimal? If not that, then how about within a factor of 2 of optimal? Or, anything non-trivial?
- Second approach: Find heuristics that speed up the algorithm for some cases, but still exponential time in the worst case.

Today's lecture focuses on the first idea, to derive polynomial-time *approximation algorithms*.

1.1 Formal definition

Definition: Approximation Algorithm

Given some optimization problem with optimal solution value OPT, and an algorithm which produces a feasible solution with value ALG, we say that the algorithm is a c-*approximation algorithm* if ALG is *always* within a factor of c of OPT. The convention differs depending on whether the optimization problem is a minimization or maximization problem.

Minimization An algorithm is a c-approximation ($c > 1$) if for all inputs, $\text{ALG} \le c \cdot \text{OPT}$.

Maximization An algorithm is a c-approximation ($0 < c < 1$) if for all inputs, $\text{ALG} \ge c \cdot \text{OPT}$.

2 Scheduling Jobs on Multiple Machines to Minimize Load

Problem: Scheduling jobs on multiple machines to minimize the makespan

You have m identical machines on which you want to schedule some n jobs. Each job $i \in \{1, 2, \ldots, n\}$ has a processing time $p_i > 0$. You want to partition the jobs among the machines to minimize the load of the most-loaded machine. In other words, if S_j is the set of jobs assigned to machine j, define the *makespan* of the solution to be

$$\max_{1 \le j \le m} \left(\sum_{i \in S_i} p_j \right)$$

You want to minimize the makespan of the solution you output.

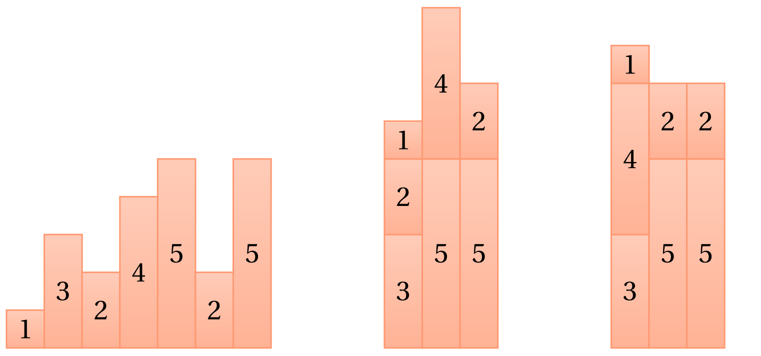
This is the formal definition of the problem that is usually used in textbooks, but here's a nicer and (subjectively) more intuitive way to describe the problem.

Problem: Stacking blocks to minimize the height

You have n blocks, the i^{th} of which has height p_i. You want to arrange the blocks into m stacks such that the height of the tallest stack is as short as possible.

Observe that these two problems are exactly the same, just with a different story, but the latter (I think) is easier to visualize and think about.

Example Here's an example input to the job scheduling / block stacking problem. Say we have $p = \{1, 3, 2, 4, 5, 2, 5\}$ and $m = 3$. The blocks are shown on the left, and two possible ways to stack them are shown on the right. The makespan is the height of the tallest stack, which is 9 for the first example, and 8 for the second example.



Makespan = 9 Makespan = 8

The second example turns out to be optimal. Can you think of a proof of why?

Problem 1. Give a concise argument that a makespan of 8 is optimal for the block stacking example above.

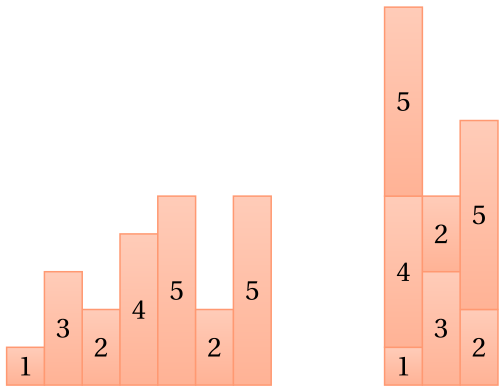
2.1 Algorithms for job scheduling / block stacking

Our first approach to the solve the block stacking problem is a *greedy algorithm*. Recall that greedy algorithms are those that just look for some locally best choice and make that at each step, rather than planning ahead in any way. Greedy algorithms are often very good for producing approximations.

Algorithm: Greedy job scheduling / block stacking

Start with m empty stacks, then, for each block, place it on the current shortest stack.

Applying this to the example above, we would get the following configuration, which has a makespan of 10, which is only 25% more than the optimal, so not too bad.



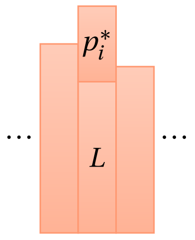
ALG = 10

But that was just one example, how bad can it get in general? We need to prove that this algorithm always gives us something that is pretty good, or near optimal.

Theorem 1: Quality of greedy job scheduling

The greedy approach outputs a solution with makespan at most 2 times the optimum, i.e., it is a 2-approximation algorithm.

Proof. Let's start by looking at the height of the tallest stack in our solution, since this is what defines the makespan (the answer). Call the last block added to the tallest stack i^*, so its height is p_{i^*}. Now call the remaining height of the tallest stack L. So we have by definition $\text{ALG} = L + p_{i^*}$. The hardest part of these greedy algorithm proofs is relating the value of ALG to the value of OPT.



$\text{ALG} = p_i^* + L$

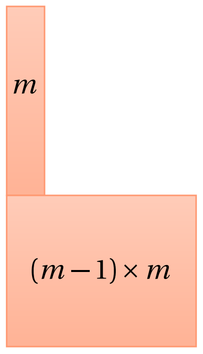
First, note that since every block must be placed somewhere, $\text{OPT} \geq p_i$ for all i and specifically, $\text{OPT} \geq p_{i^*}$. What can we say about L? Remember that the algorithm always chooses the *shortest stack* to place the next block, so when it decided to place i^* on the stack, it was because L was the height of the shortest stack at the time. This means that every stack has height at least L, which means that $\text{OPT} \geq L$.

So, combining these two inequalities, we get

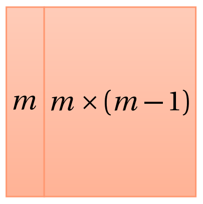
$$\text{ALG} = L + p_{i^*} \leq \text{OPT} + \text{OPT} = 2\text{OPT}$$

2.2 A worst-case example

Is this analysis tight? Sadly, yes. Suppose we have $m(m-1)$ jobs of size 1, and 1 job of size m, and we schedule the small jobs before the large jobs. The greedy algorithm will spread the small jobs equally over all the machines, and then the large job will stick out, giving a makespan of $2m-1$, whereas the right thing to do is to spread the small jobs over $m-1$ machines and get a makespan of m.



$\text{ALG} = 2m-1$



$\text{OPT} = m$

The approximation ratio in this case is $\frac{2m-1}{m} = 2(1-\frac{1}{m}) \approx 2$, so this looks almost tight. It can be made tight with a bit more analysis, but 2 is the tightest constant approximation ratio.

Problem 2. Improve the analysis slightly to show that the approximation ratio of the greedy algorithm is actually $2(1-\frac{1}{m})$, which makes the above example tight.

Can we get a better algorithm? The worst-case example helps us see the problem: when small jobs come before big jobs they can cause big problems! So lets prevent this...

2.3 An improved greedy algorithm

Algorithm: Sorted greedy job scheduling / block stacking

Sort the blocks from biggest to smallest, then do the greedy algorithm.

This algorithm prevents the worst-case example that we showed before, but what does it do in general? Let's prove that it is in fact an improvement.

Theorem 2: Quality of sorted greedy job scheduling

The sorted greedy approach outputs a solution with makespan at most 1.5 times the optimum, i.e., it is a 1.5-approximation algorithm.

Proof. Let's use the same setup as before and say that i^* is the last block added to the tallest stack, and L is the height of the rest of the stack underneath i^*, so the makespan (tallest stack) is $L + p_{i^*}$. We still have the facts that $\text{OPT} \geq L$ and $\text{OPT} \geq p_{i^*}$. We need to make some new observation in order to get the approximation ratio lower.

First, suppose $L > 0$, which means there are some blocks underneath i^*. Since the blocks were processed in sorted order (**important**), all of the blocks underneath are at least as large. Furthermore, since L was the height of the shortest stack at the time that i^* was added, every other stack is also non-empty and contains blocks that are at least as large as i^*. From this, we can deduce that there exists at least $m+1$ blocks of size at least p_{i^*} because there was at least one in each of the m stacks before we processed i^*. So, by the pigeonhole principle, since there are only m stacks, for any possible configuration, there must always be a stack that contains two blocks of size at least p_{i^*}. Therefore $\text{OPT} \geq 2p_{i^*}$, or equivalently $p_{i^*} \leq \frac{1}{2}\text{OPT}$.

Combining this with our original inequality, we get

$$\text{ALG} = L + p_{i^*} \leq \text{OPT} + \tfrac{1}{2}\text{OPT} = 1.5\text{OPT}.$$

Now we have an edge case to deal with. What if $L = 0$? Then we can't say that there are any blocks underneath i^* or make any argument about the number of large blocks. In this case, we just have $\text{ALG} = p_{i^*}$, but we know that $\text{OPT} \geq p_{i^*}$, so actually $\text{ALG} = \text{OPT}$, so we get the exact answer in this case. □

So again we ask if this analysis is tight. In fact, this time it isn't. We can further reason about the properties of the sorted algorithm to get a better approximation ratio.

Problem 3. It is possible to show that the makespan of Sorted Greedy is at most $\frac{4}{3}\text{OPT}$, and that this approximation ratio is indeed tight. Try it.

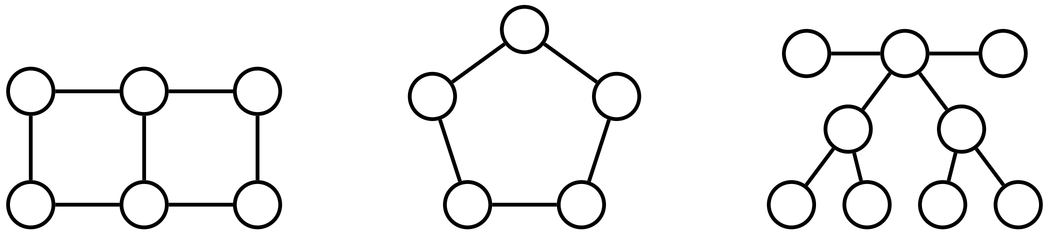
3 Vertex Cover via LP Rounding

Recall that a *vertex cover* in a graph is a set of vertices such that every edge is incident to (covers) at least one of them. The minimum vertex cover problem is to find the smallest such set of vertices.

Definition: Minimum Vertex Cover

Given a graph G, find a smallest set of vertices such that every edge is incident to at least one of them.

For instance, this problem is like asking: what is the fewest number of guards we need to place in a museum in order to cover all the corridors.



Problem 4. Find a vertex cover in the graphs above of size 3. Argue that there is no vertex cover of size 2.

3.1 A linear-programming-based algorithm

We don't expect to find the optimal solution in polynomial-time, but we will show in this section that we can use *linear programming* to obtain a 2-approximate solution. That is, if the graph G has a vertex cover of size k^*, we can return a vertex cover of size at most $2k^*$. Let's remind ourselves of the LP *relaxation* of the vertex cover problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} w_v & \\ \text{s.t.} \quad & w_u + w_v \geq 1 & \forall \{u, v\} \in E \\ & w_v \geq 0 & \forall v \in V \end{aligned}$$

Remember that in the integral version of the problem, the variables denote that a vertex v is in the cover if $x_v = 1$ and not in the cover if $x_v = 0$. Solving the integral version is NP-hard, so we settle for a relaxation, where we allow fractional values of x_v, so a vertex can be "half" in the cover. This is called an "LP relaxation" because any true vertex cover is a feasible solution, but we've made the problem easier by allowing fractional solutions too. So, the value of the optimal solution now will be at least as good as the smallest vertex cover, maybe even better (i.e., smaller), but it just might not be legal any more.

Since, in an actual vertex cover we can not take half of a vertex, our goal is to convert this fractional solution into an actual vertex cover. Here's a natural idea.

Algorithm: Relax-and-round for vertex cover

Solve the LP relaxation for x_v for each $v \in V$, then pick each vertex for which $x_v \geq \frac{1}{2}$

This is called *rounding* the linear program (which literally is what we are doing here by rounding the fraction to the nearest integer — for other problems, the "rounding" step might not be so simple).

Theorem: Relax-and-round is a 2-approximation

The above algorithm is a 2-approximation to VERTEX-COVER.

Proof. We need to prove two things. First, that we actually obtain a valid vertex cover (every edge must be incident to a vertex that we pick) and that the size of the resulting cover is not too large.

Feasibility Suppose for the sake of contradiction that there exists an edge (u, v) that is not covered. Then that means we did not pick either u or v, which means that $x_u < \frac{1}{2}$ and $x_v < \frac{1}{2}$. Therefore, $x_u + x_v < 1$, but this contradicts the LP constraint that $x_u + x_v \geq 1$. So the algorithm always outputs a vertex cover.

Approximation ratio Let LP denote the objective value of the relaxation. Since it is a relaxation, its solution can not be worse than the optimal integral solution (the actual minimum vertex cover), so LP $\leq$ OPT. Furthermore, when we round the solution, in the worst case, we increase variables from $\frac{1}{2}$ to 1, so we double their value. Since the objective is $\sum x_v$, this at most doubles the objective value, so ALG $\leq$ 2LP. Combining these inequalities, we get ALG $\leq$ 2LP $\leq$ 2OPT. □

Again, we ask if this is analysis is tight. It in fact is, and a simple example to show that is the graph with two vertices u and v connected by an edge. If the LP relaxation assigns $x_u = x_v = \frac{1}{2}$, we will pick both vertices for the cover, giving ALG = 2 when OPT = 1.

3.2 Hardness of Approximation

Interesting fact: nobody knows any approximation algorithm for vertex cover with approximation ratio 1.99. Best known is $2 - O(1/\sqrt{\log n})$, which is $2 - o(1)$.

There are results showing that a good-enough approximation algorithm will end up showing that **P**=**NP**. Clearly, a 1-approximation would find the exact vertex cover, and show this. H**r**astad showed that if you get a 7/6-approximation, you would prove **P**=**NP**. This 7/6 was improved to 1.361 by Dinur and Safra. Beating $2 - \varepsilon$ has been related to some other problems (it is "unique games hard"), but is not known to be **NP**-hard.

4 Scaling algorithms

Recall the famous Knapsack problem from the Dynamic Programming lecture.

Definition: The Knapsack Problem

We are given a set of n items, where each item i is specified by a size s_i and a value v_i. We are also given a size bound S (the size of our knapsack). The goal is to find the subset of items of maximum total value such that sum of their sizes is at most S (they all fit into the knapsack).

In that lecture, we gave a dynamic programming algorithm whose running time was $O(nS)$, which is not polynomial time but rather **pseudopolynomial time** since it depends linearly on S, which could be a very large number (exponential in the number of bits required to specify the problem). We can derive an alternate dynamic programming algorithm that does not depend on S but will instead depend on the magnitude of the values (this may seem equally useless but the reason will become clear momentarily).

Another pseudopolynomial-time solution Let us define the following subproblems:

$$G(k,P) = \text{Minimum weight of a subset of tiems } \{1,\dots,k\} \text{ with value} \geq P$$

This is kind of like the "dual" of the original dynamic program where we were trying to maximize the value with a fixed amount of weight, now we minimize the weight needed to obtain a fixed amount of value. We can solve this with the following recurrence:

$$G(k,P) = \begin{cases} 0 & \text{if } k=0 \text{ and } P \leq 0, \\ \infty & \text{if } k=0 \text{ and } P>0, \\ \min(G(k-1,P), G(k-1,P-v_k)+s_k) & \text{otherwise} \end{cases}$$

This new solution has runtime proportional to the largest possible value of any subset of items. If we let V denote the value of the most valuable item, then the total value of any subset is at most nV, so the runtime of this algorithm is $O(n^2V)$.

Now, what is so interesting compared to our previous algorithm. The different is that our runtime is now proportional to the values rather than the weights, but either of those quantities could be large (exponential-size) numbers, so how does this help? The difference is that today we are okay with approximate solutions, so we are allowed to edit the values if it makes the problem faster to solve! Provided that we don't change the optimal objective too much, we can try to reduce the values which will consequently reduce the runtime of the algorithm!

4.1 The scaling algorithm

The scaling algorithm for Knapsack works as follows. We set

$$k = \frac{V}{10n},$$

and then we scale down the value of every item in the input by a factor of k, i.e., we set

$$v_i' = \left\lfloor \frac{v_i}{k} \right\rfloor.$$

Now we simply solve the scaled down instance and return the optimal set of items for the scaled down instance! The claims are that this algorithm is now polynomial time and that is returns a pretty close solution to the true optimal solution to the original (unscaled) problem.

Theorem: Runtime

The scaling algorithm runs in $O(n^3)$ time.

Proof. By scaling every value by k, the largest value in the input is now $10n$, and therefore the total value of every item is at most $10n^2$. The dynamic program above therefore runs in $O(n^3)$ time. □

Theorem: Approximation ratio

The scaling algorithm is a 0.9 approximation.

Proof. The key to the proof is understanding how much value we "lose" when scaling. Notice that when we "unscale" an optimal solution to the scaled problem, for each item, we lose at most

$$v_i - v_i' \cdot k = v_i - \left\lfloor \frac{v_i}{k} \right\rfloor \cdot k \le k$$

value. Therefore the entire solution loses at most $nk = \frac{V}{10}$ value. However, notice that $\text{OPT} \ge V$ since we can always take that single item, so therefore we have

$$\text{ALG} \ge \text{OPT} - \frac{V}{10} \ge \text{OPT} - \frac{\text{OPT}}{10} = 0.9\text{OPT}.$$

Therefore the solution has value at least 0.9 OPT, so the algorithm is a 0.9 approximation. □

Remark: Polynomial-time approximation schemes

The constant factor of 10 in the algorithm above was arbitrary, but it got us a 0.9 approximation. It turns out that this algorithm works for any ε you want! You scale by

$$k = \varepsilon \frac{V}{n},$$

and you get a $(1-\varepsilon)$ approximation! This is called a polynomial-time approximation scheme. Actually in this case its a **fully** polynomial-time approximation scheme, because the runtime scales polynomially in ε as well (you can have polynomial-time approximation schemes where the dependence on ε is exponential.)