24
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 $\mathrm{Alg}(I)$ is the output/solution produced by the algorithm, and $c(\mathrm{Alg}(I))$ its cost. Similarly, let $\mathrm{Opt}(I)$ denote the optimal output for input I, and let $c(\mathrm{Opt}(I))$ 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

$$\mathrm{Alg} \leq r \cdot \mathrm{Opt}$$

to denote that

$$c(\mathrm{Alg}(I)) \leq r \cdot c(\mathrm{Opt}(I)).$$

$$\rho = \rho_{\mathcal{A}} := \max_I \frac{c(\mathrm{Alg}(I))}{c(\mathrm{Opt}(I))}.$$

In this case, we say that Alg is an ρ-approximation algorithm. For *maximization* problems, we define *rho* to be

$$\rho = \rho_{\mathcal{A}} := \min_I \frac{c(\mathrm{Alg}(I))}{c(\mathrm{Opt}(I))},$$

and therefore a number in $[0, 1]$.

24.1 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.

24.2 *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:

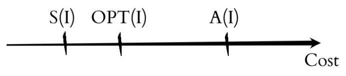

Figure 24.1: The cost diagram on instance I (costs increase from left to right).

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.

24.3 *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 $\mathbf{P} = \mathbf{NP}$).

24.3.1 *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 24.1. *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 $\ln n$.

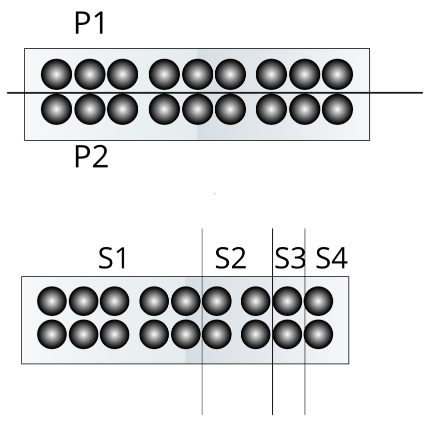

Figure 24.2: A Tight Example for the Greedy Algorithm

Proof of Theorem 24.1. Suppose Opt picks k sets from $\mathcal{S}$. Let n_i be the number of elements yet 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,

$$n_{i+1} \leq n_i - n_i/k = n_i(1 - 1/k).$$

As always, we use $1 + x \leq e^x$, and here we can use that the inequality is strict whenever $x \neq 0$.

Iterating, we get $n_t \leq n_0(1 - 1/k)^t < n \cdot e^{-t/k}$. 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. □

If the sets are of size at most B, we can show that the greedy algorithm is an H_B-approximation, where

$$H_B = 1 + 1/2 + 1/3 + \ldots + 1/B$$

is the B^{th} Harmonic number.

24.3.2 *Extending to the Weighted Case*

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 somewhat like the one above for this weighted case as well: let k now be the total cost of sets in the optimal set cover. After i sets have been picked, the remaining n_i elements can still be covered using a collection of cost k, so there must be a set whose cost-to-fresh-coverage ratio is at most k/n_i. If it covers $n_{i+1} - n_i$ previously uncovered elements, then we know that its cost most be at most

$$(n_{i+1} - n_i) \cdot k/n_i.$$

So if the algorithm picks ℓ sets, the total cost is

$$\sum_{i=1}^{\ell} (n_{i+1} - n_i) \cdot k/n_i \leq k\big(1/n + 1/(n-1) + \ldots + 1/2 + 1\big) = k \cdot H_n,$$

where we used that $n_0 = n$, since all elements are initially uncovered.

24.4 *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:

$$\begin{aligned} &\min \sum_{S \in \mathcal{S}} c_S x_S && \text{(ILP-SC)} \\ s.t. \quad &\sum_{S: e \in S} x_S \geq 1 && \forall e \in U \\ &x_S \in \{0,1\} && \forall S \in \mathcal{S}. \end{aligned}$$

The LP relaxation just drops the integrality constraints:

$$\begin{aligned} &\min \sum_{S \in \mathcal{S}} c_S x_S && \text{(LP-SC)} \\ s.t. \quad &\sum_{S: e \in S} x_S \geq 1 && \forall e \in U \\ &x_S \geq 0 && \forall S \in \mathcal{S}. \end{aligned}$$

If $\mathrm{LP}(I)$ is the optimal value for the linear program, then we get:

$$\mathrm{LP}(I) \leq \mathrm{Opt}(I).$$

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 24.2. *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.

24.5 *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.

24.5.1 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 24.3. $\mathrm{Alg}_{FF}(I) \leq 2 \cdot \mathrm{Opt}(I)$.

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,

$$\lfloor K/2 \rfloor < V(I) \implies K \leq 2 \lceil V(I) \rceil \leq 2\, \mathrm{Opt}(I). \qquad \square$$

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

$$\frac{1}{1-\varepsilon}\,\mathrm{Opt}(I) + 1 \approx (1+\varepsilon)\,\mathrm{Opt}(I) + 1.$$

24.6 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

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.

we will show that if we allow ourselves a small additive term, the hardness goes away. The main ideas here will be the following:

1. We can discretize the item sizes down to a constant number of values by losing at most $\varepsilon\,\mathrm{Opt}$ (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 $(1+\varepsilon)$.

24.6.1 The Linear Grouping Procedure

Lemma 24.4 (Linear Grouping)**.** *Given an instance $I = (s_1, s_2, \ldots, s_n)$ of* Bin Packing*, and a parameter $D \in \mathbb{N}$, we can efficiently produce another instance $I' = ({s_1}', {s_2}', \ldots, {s_n}')$ with increased sizes ${s_i}' > s_i$ and at most D distinct item sizes, such that*

$$\mathrm{Opt}(I') \leq \mathrm{Opt}(I) + \lceil n/D \rceil .$$

Proof. The instance I' is constructed as follows:

- Sort the sizes s_i in non-increasing order to get $s_1 \geq s_2 \geq \ldots \geq s_n$.
- 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 $\mathrm{Opt}(I) + \lceil n/D \rceil$ bins. Indeed, suppose $\mathrm{Opt}(I)$ 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. ☐

24.6.2 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 $s_1 < s_2 < \ldots < S_D$, 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.

24.6.3 *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

$$\frac{1}{1-\varepsilon}\,\mathrm{Opt}(I) + 1 \approx (1 + O(\varepsilon))\,\mathrm{Opt}(I) + 1,$$

as long as $\varepsilon \leq 1/2$.

24.7 Subsequent Results and Open Problems