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 \( \text{Alg} \) 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:

\[
\rho = \rho_\mathcal{A} := \max_I \frac{c(\text{Alg}(I))}{c(\text{Opt}(I))}.
\]

In this case, we say that \( \text{Alg} \) is an \( \rho \)-approximation algorithm.

\section{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-hard problems is still useful.

- **Fully Poly-Time Approximation Scheme (FPTAS):** For problems in this category, there exist approximation algorithms that take in a parameter \( \epsilon \), and output a solution with approximation ratio \( 1 + \epsilon \) in time \( \text{poly}(\langle I \rangle, 1/\epsilon) \). E.g., one such problem is Knapsack, where

\[
\text{Alg} \leq r \cdot \text{Opt}
\]

to denote that

\[
c(\text{Alg}(I)) \leq r \cdot c(\text{Opt}(I)).
\]
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.

- **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 \( \varepsilon \), with the doubly-exponential dependence on \( d \), is unavoidable.

- **Constant-Factor Approximation:** Examples in this class include the Traveling Salesman Problem on general metrics. In the late 1970s, Nicos Christofides and XXXX Serdyukov discovered the same 1.5-approximation algorithm for metric TSP. This was improved only in 2020, when Anna Karlin, Nathan Klein, and Shayan Oveis-Gharan found a way to get an \( (1.5 - \varepsilon) \)-approximation. Meanwhile, it has been shown that metric TSP can’t be approximated with a ratio better than \( \frac{123}{122} \) under the assumption of \( P \neq NP \), by Karpinski, Lampis, Schmied.

- **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 \( P = NP \). 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 \( P = NP \). 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 \( \text{Opt}(I) \). An important idea in proving the approximation guarantee involves the use of a *surrogate*, or a *lower bound*, as follows: Given an algorithm \( \text{Alg} \) 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:

1. We show that $S(I) \leq \text{Opt}(I)$ for all $I$, and
2. then show that $\text{Alg}(I) \leq aS(I)$ for all $I$.

This shows that $\text{Alg}(I) \leq a \text{Opt}(I)$. 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 $U = \bigcup_{S \in \mathcal{S}} S$. We want to find a subset $S' \subseteq \mathcal{S}$, such that $U = \bigcup_{S \in S'} S$ while minimizing the size $|S'|$.

In the weighted version of Set Cover, we have a cost $c_S$ for each set $S \in \mathcal{S}$, and want to minimize $c(S') = \sum_{S \in S'} c_S$. 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 $\text{NP}$-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 $\text{NP}$ 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 $\text{NP}$ has polynomial-time algorithm (i.e., that $\text{P} = \text{NP}$).

#### 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 \( \ln n \).

**Proof of Theorem 20.1.** Suppose \( \text{Opt} \) picks \( k \) sets from \( S \). 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 \( \text{Opt} \) cover all the elements of \( U \), they also cover the uncovered elements in \( n_i \). By averaging, there must exist a set in \( S \) that covers \( n_i/k \) of the yet-uncovered elements. Hence,

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

Iterating, we get \( n_T \leq n_0(1 - 1/k)^T < n \cdot e^{1/k} \). 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 \( (1 + \ln B) \)-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.

### 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 \textbf{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 \( \text{Opt} \)). This optimal value LP value is the \textbf{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

\[
\text{Minimize: } \sum_{S} c_S x_S
\]

\text{Subject to: } \forall e \in U, \sum_{S : e \in S} x_S \geq 1

\text{And: } x_S \in \{0, 1\} \text{ for all } S \subseteq U.
precisely models Set Cover:

\[
\begin{align*}
\text{min} & \quad \sum_{S \in \mathcal{S}} c_S x_S \\
\text{s.t.} & \quad \sum_{S \in \mathcal{S}} x_S \geq 1 \quad \forall e \in U \\
& \quad x_S \in \{0, 1\} \quad \forall S \in \mathcal{S}.
\end{align*}
\]

(ILP-SC)

The LP relaxation just drops the integrality constraints:

\[
\begin{align*}
\text{min} & \quad \sum_{S \in \mathcal{S}} c_S x_S \\
\text{s.t.} & \quad \sum_{S \in \mathcal{S}} x_S \geq 1 \quad \forall e \in U \\
& \quad x_S \geq 0 \quad \forall S \in \mathcal{S}.
\end{align*}
\]

(LP-SC)

If 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 expected number of elements not covered in Phase 1. To calculate this, consider an arbitrary element \(e\).

\[
\Pr[e \text{ not covered in phase 1}] = (\prod_{S \in \mathcal{S}} (1 - x^*_S))^t \\
\quad \leq (e^{-\sum S \in \mathcal{S} x^*_S})^t \\
\quad \leq (e^{-1})^t \\
\quad = \frac{1}{n^t},
\]

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! Details soon.
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 Bin Packing

**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 \( \frac{3}{2} \) 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 \( \text{Opt} + O(\log \text{Opt}) \) bins, and it is conceivable that we can get an algorithm that uses \( \text{Opt} + O(1) \) 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 bins in decreasing order of free space, and add the item to the first one that can take it.
All these algorithms are 2-approximations. Let us give a proof for First-Fit, the others have similar proofs.

**Theorem 20.3.** $\text{Alg}_{\text{FF}}(I) \leq 2 \cdot \text{Opt}(I)$.

**Proof.** The surrogate in this case is the total volume $V(I) = \sum_i s_i$ of items. Clearly, $[V(I)] \leq \text{OPT}(I)$. 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.

Hence, if we open $K$ bins, the total volume of items in these bins is strictly more than $\lceil K/2 \rceil$. Hence,

$$\lceil K/2 \rceil < V(I) \implies K \leq 2 \lceil V(I) \rceil \leq 2 \text{Opt}(I).$$

Exercise: if all the items were of size at most $\epsilon$, then each bin (except the last one) would have at least $1 - \epsilon$ total size, thereby giving an approximation of

$$\frac{1}{1-\epsilon} \text{Opt}(I) + 1 \approx (1+\epsilon) \text{Opt}(I) + 1.$$

### 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:

1. We can discretize the item sizes down to a constant number of values by losing at most $\epsilon \text{Opt}$ (where the constant depends on $\epsilon$).
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 $\epsilon$ can be added to any instance while maintaining an approximation factor of $(1+\epsilon)$.

#### 20.6.1 The Linear Grouping Procedure

**Lemma 20.4** (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

$$\text{Opt}(I') \leq \text{Opt}(I) + \lceil n/D \rceil.$$
Proof. The instance $I'$ is constructed as follows:

- Sort the sizes $s_i$ in non-increasing order to get $s_1 \geq s_2 \geq \ldots \geq s_n$.
- 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 $\text{Opt}(I) + \lceil n/D \rceil$ 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. \qed

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 $s_1 < s_2 < \ldots < S_D$, 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 $s_1 \geq \epsilon$.) 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:

$$\min \sum_{C \in \mathcal{C}} x_C,$$

$$\text{s.t. } \sum_{C} A_{Cs} x_C \geq n_s, \quad \forall \text{ sizes } s,$$

$$x_C \in \mathbb{N}.$$

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 \( \text{LP}(I) + D \leq \text{Opt}(I) + D \) 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

\[
\text{Opt}(I) + \left\lceil n/D \right\rceil + D
\]

bins. Now if we could ensure that \( n/D \) were at most \( \epsilon \text{Opt}(I) \), when \( D \) was \( f(\epsilon) \), we would be done. Indeed, if all the items have size at least \( \epsilon \), the total volume (and therefore \( \text{Opt}(I) \)) is at least \( \epsilon n \). If we now set \( D = \left\lceil 1/\epsilon^2 \right\rceil \), then \( n/D \leq \epsilon^2 n \leq \epsilon \text{Opt}(I) \), and the number of bins is at most

\[
(1 + \epsilon) \text{Opt}(I) + \left\lceil 1/\epsilon^2 \right\rceil.
\]

What if some of the items are smaller than \( \epsilon \)? We now use the observation that \textsc{First-Fit} behaves very well when the item sizes are small. Indeed, we first hold back all the items smaller than \( \epsilon \), and solve the remaining instance as above. Then we add in the small items using \textsc{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 \( (1 - \epsilon) \) total size in them. The number of bins is then at most

\[
\frac{1}{1 - \epsilon} \text{Opt}(I) + 1 \approx (1 + O(\epsilon)) \text{Opt}(I) + 1,
\]

as long as \( \epsilon \leq \frac{1}{2} \).

### 20.7 Subsequent Results and Open Problems

For both questions.