CS 583: Approximation Algorithms

Chandra Chekuri

August 24, 2021

\[1\text{Dept. of Computer Science, University of Illinois, Urbana, IL 61820. Email: chekuri@illinois.edu.}\]
Contents

1 Introduction .......................................................... 3
  1.1 Formal Aspects .................................................. 6
    1.1.1 NP Optimization Problems ................................. 6
    1.1.2 Relative Approximation .................................. 7
    1.1.3 Additive Approximation .................................. 7
    1.1.4 Hardness of Approximation ............................... 8
  1.2 Designing Approximation Algorithms ........................... 10

2 Covering Problems .................................................. 12
  2.1 Greedy for Set Cover and Maximum Coverage .................. 13
    2.1.1 Greedy Algorithm ........................................ 13
    2.1.2 Analysis of Greedy Cover ................................. 14
    2.1.3 Dominating Set ........................................... 17
  2.2 Vertex Cover .................................................... 18
    2.2.1 A 2-approximation for Vertex Cover ..................... 18
    2.2.2 Set Cover with small frequencies ......................... 19
  2.3 Vertex Cover via LP ............................................. 19
  2.4 Set Cover via LP ............................................... 22
    2.4.1 Deterministic Rounding ................................... 22
    2.4.2 Randomized Rounding .................................... 23
    2.4.3 Dual-fitting ............................................. 27
    2.4.4 Greedy for implicit instances of Set Cover ............ 29
  2.5 Submodularity ................................................... 29
    2.5.1 Submodular Set Cover .................................... 30
    2.5.2 Submodular Maximum Coverage .............................. 31
  2.6 Covering Integer Programs (CIPs) ............................... 31
Chapter 1

Introduction

These are lecture notes for a course on approximation algorithms.

Course Objectives

1. To appreciate that not all intractable problems are the same. \textbf{NP} optimization problems, identical in terms of exact solvability, can appear very different from the approximation point of view. This sheds light on why, in practice, some optimization problems (such as \textsc{Knapsack}) are easy, while others (like \textsc{Clique}) are extremely difficult.

2. To learn techniques for design and analysis of approximation algorithms, via some fundamental problems.

3. To build a toolkit of broadly applicable algorithms/heuristics that can be used to solve a variety of problems.

4. To understand reductions between optimization problems, and to develop the ability to relate new problems to known ones.

The complexity class \textbf{P} contains the set of problems that can be solved in polynomial time. From a theoretical viewpoint, this describes the class of tractable problems, that is, problems that can be solved efficiently. The class \textbf{NP} is the set of problems that can be solved in non-deterministic polynomial time, or equivalently, problems for which a solution can be verified in polynomial time. \textbf{NP} contains many interesting problems that often arise in practice, but there is good reason to believe \textbf{P} \textless \textbf{NP}. That is, it is unlikely that there exist algorithms to solve \textbf{NP} optimization problems efficiently, and so we often resort to heuristic methods to solve these problems.

Heuristic approaches include backtrack search and its variants, mathematical programming methods, local search, genetic algorithms, tabu search, simulated
annealing etc. Some methods are guaranteed to find an optimal solution, though they may take exponential time; others are guaranteed to run in polynomial time, though they may not return a (optimal) solution. Approximation algorithms are (typically) polynomial time heuristics that do not always find an optimal solution but they are distinguished from general heuristics in providing guarantees on the quality of the solution they output.

Approximation Ratio: To give a guarantee on solution quality, one must first define what we mean by the quality of a solution. We discuss this more carefully later. For now, note that each instance of an optimization problem has a set of feasible solutions. The optimization problems we consider have an objective function which assigns a (real/rational) number/value to each feasible solution of each instance $I$. The goal is to find a feasible solution with minimum objective function value or maximum objective function value. The former problems are minimization problems and the latter are maximization problems.

For each instance $I$ of a problem, let $OPT(I)$ denote the value of an optimal solution to instance $I$. We say that an algorithm $A$ is an $\alpha$-approximation algorithm for a problem if, for every instance $I$, the value of the feasible solution returned by $A$ is within a (multiplicative) factor of $\alpha$ of $OPT(I)$. Equivalently, we say that $A$ is an approximation algorithm with approximation ratio $\alpha$. For a minimization problem we would have $\geq 1$ and for a maximization problem we would have $\leq 1$. However, it is not uncommon to find in the literature a different convention for maximization problems where one says that $A$ is an $\alpha$-approximation algorithm if the value of the feasible solution returned by $A$ is at least $\frac{1}{\alpha} \cdot OPT(I)$; the reason for using convention is so that approximation ratios for both minimization and maximization problems will be $\geq 1$. In this course we will for the most part use the convention that $\geq 1$ for minimization problems and $\leq 1$ for maximization problems.

Remarks:

1. The approximation ratio of an algorithm for a minimization problem is the maximum (or supremum), over all instances of the problem, of the ratio between the values of solution returned by the algorithm and the optimal solution. Thus, it is a bound on the worst-case performance of the algorithm.

2. The approximation ratio $\alpha$ can depend on the size of the instance $I$, so one should technically write $\alpha(|I|)$.

3. A natural question is whether the approximation ratio should be defined in an additive sense. For example, an algorithm has an $\alpha$-approximation for a minimization problem if it outputs a feasible solution of value at most
OPT(I) + \alpha for all I. This is a valid definition and is the more relevant one in some settings. However, for many NP problems it is easy to show that one cannot obtain any interesting additive approximation (unless of course \( P = NP \)) due to scaling issues. We will illustrate this via an example later.

**Pros and cons of the approximation approach:** Some advantages to the approximation approach include:

1. It explains why problems can vary considerably in difficulty.
2. The analysis of problems and problem instances distinguishes easy cases from difficult ones.
3. The worst-case ratio is *robust* in many ways. It allows *reductions* between problems.
4. Approximation algorithmic ideas/tools/relaxations are valuable in developing heuristics, including many that are practical and effective.
5. Quantification of performance via a concrete metric such as the approximation ratio allows for innovation in algorithm design and has led to many new ideas.

As a bonus, many of the ideas are beautiful and sophisticated, and involve connections to other areas of mathematics and computer science.

Disadvantages include:

1. The focus on *worst-case measures* risks ignoring algorithms or heuristics that are practical or perform well *on average*.
2. Unlike, for example, integer programming, there is often no incremental/continuous tradeoff between the running time and quality of solution.
3. Approximation algorithms are often limited to cleanly stated problems.
4. The framework does not (at least directly) apply to decision problems or those that are inapproximable.

**Approximation as a broad lens**

The use of approximation algorithms is not restricted solely to NP-Hard optimization problems. In general, ideas from approximation can be used to solve many problems where finding an exact solution would require too much of any resource.
A resource we are often concerned with is \textit{time}. Solving \textbf{NP}-Hard problems exactly would (to the best of our knowledge) require exponential time, and so we may want to use approximation algorithms. However, for large data sets, even polynomial running time is sometimes unacceptable. As an example, the best exact algorithm known for the \textsc{Matching} problem in general graphs requires $O(m \sqrt{n})$ time; on large graphs, this may be not be practical. In contrast, a simple greedy algorithm takes near-linear time and outputs a matching of cardinality at least $1/2$ that of the maximum matching; moreover there have been randomized sub-linear time algorithms as well.

Another often limited resource is \textit{space}. In the area of data streams/streaming algorithms, we are often only allowed to read the input in a single pass, and given a small amount of additional storage space. Consider a network switch that wishes to compute statistics about the packets that pass through it. It is easy to exactly compute the average packet length, but one cannot compute the median length exactly. Surprisingly, though, many statistics can be approximately computed.

Other resources include programmer time (as for the \textsc{Matching} problem, the exact algorithm may be significantly more complex than one that returns an approximate solution), or communication requirements (for instance, if the computation is occurring across multiple locations).

\section{1.1 Formal Aspects}

\subsection{1.1.1 \textbf{NP} Optimization Problems}

In this section, we cover some formal definitions related to approximation algorithms. We start from the definition of optimization problems. A problem is simply an infinite collection of \textit{instances}. Let $\Pi$ be an optimization problem. $\Pi$ can be either a minimization or maximization problem. Instances $I$ of $\Pi$ are a subset of $\Sigma^*$ where $\Sigma$ is a finite encoding alphabet. For each instance $I$ there is a set of feasible solutions $S(I)$. We restrict our attention to real/rational-valued optimization problems; in these problems each feasible solution $S \in S(I)$ has a value $val(S, I)$. For a minimization problem $\Pi$ the goal is, given $I$, find $\text{OPT}(I) = \min_{S \in S(I)} val(S, I)$.

Now let us formally define \textbf{NP} optimization (NPO) which is the class of optimization problems corresponding to \textbf{NP}.

\textbf{Definition 1.1.} $\Pi$ is in \textbf{NPO} if

- Given $x \in \Sigma^*$, there is a polynomial-time algorithm that decide if $x$ is a valid instance of $\Pi$. That is, we can efficiently check if the input string is well-formed. This is a basic requirement that is often not spelled out.
For each $I$, and $S \in S(I)$, $|S| \leq \text{poly}(|I|)$. That is, the solution are of size polynomial in the input size.

There exists a poly-time decision procedure that for each $I$ and $S \in \Sigma^*$, decides if $S \in S(I)$. This is the key property of NP; we should be able to verify solutions efficiently.

$\text{val}(I, S)$ is a polynomial-time computable function.

We observe that for a minimization NPO problem $\Pi$, there is a associated natural decision problem $L(\Pi) = \{(I, B) : \text{OPT}(I) \leq B\}$ which is the following: given instance $I$ of $\Pi$ and a number $B$, is the optimal value on $I$ at most $B$? For maximization problem $\Pi$ we reverse the inequality in the definition.

**Lemma 1.1.** $L(\Pi)$ is in NP if $\Pi$ is in NPO.

### 1.1.2 Relative Approximation

When $\Pi$ is a minimization problem, recall that we say an approximation algorithm $A$ is said to have approximation ratio $\alpha$ iff

- $A$ is a polynomial time algorithm
- for all instance $I$ of $\Pi$, $A$ produces a feasible solution $A(I)$ s.t. $\text{val}(A(I), I) \leq \alpha \text{ val } (\text{OPT}(I), I)$. (Note that $\alpha \geq 1$.)

Approximation algorithms for maximization problems are defined similarly. An approximation algorithm $A$ is said to have approximation ratio $\alpha$ iff

- $A$ is a polynomial time algorithm
- for all instance $I$ of $\Pi$, $A$ produces a feasible solution $A(I)$ s.t. $\text{val}(A(I), I) \geq \alpha \text{ val } (\text{OPT}(I), I)$. (Note that $\alpha \leq 1$.)

For maximization problems, it is also common to see use $1/\alpha$ (which must be $\geq 1$) as approximation ratio.

### 1.1.3 Additive Approximation

Note that all the definitions above are about relative approximations; one could also define additive approximations. $A$ is said to be an $\alpha$-additive approximation algorithm, if for all $I$, $\text{val}(A(I)) \leq \text{OPT}(I) + \alpha$. Most NPO problems, however, do not allow any additive approximation ratio because $\text{OPT}(I)$ has a scaling property.
To illustrate the scaling property, let us consider Metric-TSP. Given an instance $I$, let $I_\beta$ denote the instance obtained by increasing all edge costs by a factor of $\beta$. It is easy to observe that for each $S \in S(I) = S(I_\beta)$, $\text{val}(S, I_\beta) = \beta \text{val}(S, I)$ and $\text{OPT}(I_\beta) = \beta \text{OPT}(I)$. Intuitively, scaling edge by a factor of $\beta$ scales the value by the same factor $\beta$. Thus by choosing $\beta$ sufficiently large, we can essentially make the additive approximation (or error) negligible.

**Lemma 1.2.** Metric-TSP does not admit an $\alpha$ additive approximation algorithm for any polynomial-time computable $\alpha$ unless $P = NP$.

**Proof.** For simplicity, suppose every edge has integer cost. For the sake of contradiction, suppose there exists an additive $\alpha$ approximation $A$ for Metric-TSP. Given $I$, we run the algorithm on $I_\beta$ and let $S$ be the solution, where $\beta = 2\alpha$. We claim that $S$ is the optimal solution for $I$. We have $\text{val}(S, I) = \text{val}(S, I_\beta)/\beta \leq \text{OPT}(I_\beta)/\beta + \alpha/\beta = \text{OPT}(I) + 1/2$, as $A$ is $\alpha$-additive approximation. Thus we conclude that $\text{OPT}(I) = \text{val}(S, I)$, since $\text{OPT}(I) \leq \text{val}(S, I)$, and $\text{OPT}(I), \text{val}(S, I)$ are integers. This is impossible unless $P = NP$. ■

Now let us consider two problems which allow additive approximations. In the Planar Graph Coloring, we are given a planar graph $G = (V, E)$. We are asked to color all vertices of the given graph $G$ such that for any $vw \in E$, $v$ and $w$ have different colors. The goal is to minimize the number of different colors. It is known that to decide if a planar graph admits 3-coloring is NP-complete [22], while one can always color any planar graph $G$ with using 4 colors (this is the famous 4-color theorem) [2, 23]. Further, one can efficiently check whether a graph is 2-colorable (that is, if it is bipartite). Thus, the following algorithm is a 1-additive approximation for Planar Graph Coloring: If the graph is bipartite, color it with 2 colors; otherwise, color with 4 colors.

As a second example, consider the Edge Coloring Problem, in which we are asked to color edges of a given graph $G$ with the minimum number of different colors so that no two adjacent edges have different colors. By Vizing’s theorem [24], we know that one can color edges with either $\Delta(G)$ or $\Delta(G) + 1$ different colors, where $\Delta(G)$ is the maximum degree of $G$. Since $\Delta(G)$ is a trivial lower bound on the minimum number, we can say that the Edge Coloring Problem allows a 1-additive approximation. Note that the problem of deciding whether a given graph can be edge colored with $\Delta(G)$ colors is NP-complete [12].

### 1.1.4 Hardness of Approximation

Now we move to hardness of approximation.
Definition 1.2 (Approximability Threshold). Given a minimization optimization problem \( \Pi \), it is said that \( \Pi \) has an approximation threshold \( \alpha^*(\Pi) \), if for any \( \epsilon > 0 \), \( \Pi \) admits a \( \alpha^*(\Pi) + \epsilon \) approximation but if it admits a \( \alpha^*(\Pi) - \epsilon \) approximation then \( P = NP \).

If \( \alpha^*(\Pi) = 1 \), it implies that \( \Pi \) is solvable in polynomial time. Many NPO problems \( \Pi \) are known to have \( \alpha^*(\Pi) > 1 \) assuming that \( \% \neq \#\% \). We can say that approximation algorithms try to decrease the upper bound on \( \alpha^*(\Pi) \), while hardness of approximation attempts to increase lower bounds on \( \alpha^*(\Pi) \).

To prove hardness results on NPO problems in terms of approximation, there are largely two approaches; a direct way by reduction from NP-complete problems and an indirect way via gap reductions. Here let us take a quick look at an example using a reduction from an NP-complete problem.

In the (metric) \( k \)-center problem, we are given an undirected graph \( G = (V, E) \) and an integer \( k \). We are asked to choose a subset of \( k \) vertices from \( V \) called centers. The goal is to minimize the maximum distance to a center, i.e. \( \min_{S \subseteq V, |S| = k} \max_{v \in V} \text{dist}_G(v, S) \), where \( \text{dist}_G(v, S) = \min_{u \in S} \text{dist}_G(u, v) \).

The \( k \)-center problem has approximation threshold 2, since there are a few 2-approximation algorithms for \( k \)-center and there is no \( 2 - \epsilon \) approximation algorithm for any \( \epsilon > 0 \) unless \( P = NP \). We can prove the inapproximability using a reduction from the decision version of Dominating Set: Given an undirected graph \( G = (V, E) \) and an integer \( k \), does \( G \) have a dominating set of size at most \( k \)? A set \( S \subseteq V \) is said to be a dominating set in \( G \) if for all \( v \in V \), \( v \in S \) or \( v \) is adjacent to some \( u \) in \( S \). Dominating Set is known to be NP-complete.

Theorem 1.3 ([13]). Unless \( P = NP \), there is no \( 2 - \epsilon \) approximation for \( k \)-center for any fixed \( \epsilon > 0 \).

Proof. Let \( I \) be an instance of Dominating Set Problem consisting of graph \( G = (V, E) \) and integer \( k \). We create an instance \( I' \) of \( k \)-center while keeping graph \( G \) and \( k \) the same. If \( I \) has a dominating set of size \( k \) then \( \text{OPT}(I') = 1 \), since every vertex can be reachable from the Dominating Set by at most one hop. Otherwise, we claim that \( \text{OPT}(I') \geq 2 \). This is because if \( \text{OPT}(I') < 2 \), then every vertex must be within distance 1, which implies the \( k \)-center that witnesses \( \text{OPT}(I') \) is a dominating set of \( I \). Therefore, the \( (2 - \epsilon) \) approximation for \( k \)-center can be used to solve the Dominating Set Problem. This is impossible, unless \( P = NP \).
1.2 Designing Approximation Algorithms

How does one design and more importantly analyze the performance of approximation algorithms? This is a non-trivial task and the main goal of the course is to expose you to basic and advanced techniques as well as central problems. The purpose of this section is to give some high-level insights. We start with how we design polynomial-time algorithms. Note that approximation makes sense mainly in the setting where one can find a feasible solution relatively easily but finding an optimum solution is hard. In some cases finding a feasible solution itself may involve some non-trivial algorithm, in which case it is useful to properly understand the structural properties that guarantee feasibility, and then build upon it.

Some of the standard techniques we learn in basic and advanced undergraduate algorithms courses are recursion based methods such as divide and conquer, dynamic programming, greedy, local search, combinatorial optimization via duality, and reductions to existing problems. How do we adapt these to the approximation setting? Note that intractability implies that there are no efficient characterizations of the optimum solution value.

Greedy and related techniques are often fairly natural for many problems and simple heuristic algorithms often suggest themselves for many problems. (Note that the algorithms may depend on being able to solve some existing problem efficiently. Thus, knowing a good collection of general poly-time solvable problems is often important.) The main difficulty is in analyzing their performance. The key challenge here is to identify appropriate lower bounds on the optimal value (assuming that the problem is a minimization problem) or upper bounds on the optimal value (assuming that the problem is a maximization problem). These bounds allow one to compare the output of the algorithm and prove an approximation bound. In designing poly-time algorithms we often prove that greedy algorithms do not work. We typically do this via examples. This skill is also useful in proving that some candidate algorithm does not give a good approximation. Often the bad examples lead one to a new algorithm.

How does one come up with lower or upper bounds on the optimum value? This depends on the problem at hand and knowing some background and related problems. However, one would like to find some automatic ways of obtaining bounds. This is often provided via linear programming relaxations and more advanced convex programming methods including semi-definite programming, lift-and-project hierarchies etc. The basic idea is quite simple. Since integer linear programming is NP-Complete one can formulate most discrete optimization problems easily and “naturally” as an integer program. Note that there may be many different ways of expressing a given problem as an integer program. Of course we cannot solve the integer program but we
can solve the linear-programming relaxation which is obtained by removing the integrality constraints on the variables. Thus, for each instance \(I\) of a given problem we can obtain an LP relaxation \(LP(I)\) which we typically can be solve in polynomial-time. This automatically gives a bound on the optimum value since it is a relaxation. How good is this bound? It depends on the problem, of course, and also the specific LP relaxation. How do we obtain a feasible solution that is close to the bound given by the LP relaxation. The main technique here is to round the fractional solution \(x\) to an integer feasible solution \(x'\) such that \(x'\)'s value is close to that of \(x\). There are several non-trivial rounding techniques that have been developed over the years that we will explore in the course. We should note that in several cases one can analyze combinatorial algorithms via LP relaxations even though the LP relaxation does not play any direct role in the algorithm itself. Finally, there is the question of which LP relaxation to use. Often it is required to “strengthen” an LP relaxation via addition of constraints to provide better bounds. There are some automatic ways to strengthen any LP and often one also needs problem specific ideas.

Local search is another powerful technique and the analysis here is not obvious. One needs to relate the value of a local optimum to the value of a global optimum via various exchange properties which define the local search heuristic. For a formal analysis it is necessary to have a good understanding of the problem structure.

Finally, dynamic programming plays a key role in the following way. Its main use is in solving to optimality a restricted version of the given problem or a subroutine that is useful as a building block. How does one obtain a restricted version? This is often done by some clever preprocessing of a given instance.

Reductions play a very important role in both designing approximation algorithms and in proving inapproximability results. Often reductions serve as a starting point in developing a simple and crude heuristic that allows one to understand the structure of a problem which then can lead to further improvements.

Discrete optimization problems are brittle — changing the problem a little can lead to substantial changes in the complexity and approximability. Nevertheless it is useful to understand problems and their structure in broad categories so that existing results can be leveraged quickly and robustly. Thus, some of the emphasis in the course will be on classifying problems and how various parameters influence the approximability.
Chapter 2

Covering Problems

Part of these notes were scribed by Abul Hassan Samee and Lewis Tseng.

Packing and Covering problems together capture many important problems in combinatorial optimization. We will discuss several covering problems in this chapter. Two canonical one problems are Minimum Vertex Cover and its generalization Minimum Set Cover. (Typically we will omit the use of the qualifiers minimum and maximum since this is often clear from the definition of the problem and the context.) They play an important role in the study of approximation algorithms.

A vertex cover in an undirected graph $G = (V, E)$ is a set $S \subseteq V$ of vertices such that for each edge $e \in E$, at least one of its end points is in $S$. It is also called a node cover. In the Vertex Cover problem, our goal is to find a smallest vertex cover of $G$. In the weighted version of the problem, a weight function $w : V \to \mathbb{R}^+$ is given, and our goal is to find a minimum weight vertex cover of $G$. The unweighted version of the problem is also known as Cardinality Vertex Cover. Note that we are picking vertices to cover the edges. Vertex Cover is NP-Hard and is on the list of problems in Karp’s list.

In the Set Cover problem the input is a set $U$ of $n$ elements, and a collection $S = \{S_1, S_2, \ldots, S_m\}$ of $m$ subsets of $U$ such that $\bigcup_i S_i = U$. Our goal in the Set Cover problem is to select as few subsets as possible from $S$ such that their union covers $U$. In the weighted version each set $S_i$ has a non-negative weight $w_i$ the goal is to find a set cover of minimim weight. Closely related to the Set Cover problem is the Maximum Coverage problem. In this problem the input is again $U$ and $S$ but we are also given an integer $k \leq m$. The goal is to select $k$ subsets from $S$ such that their union has the maximum cardinality. Note that Set Cover is a minimization problem while Maximum Coverage is a maximization problem. Set Cover is essentially equivalent to the Hitting Set problem. In Hitting Set the input is $U$ and $S$ but the goal is to pick the smallest number of
CHAPTER 2. COVERING PROBLEMS

Elements of $\mathcal{U}$ that cover the given sets in $S$. In other words we are seeking a set cover in the dual set system. It is easy to see Vertex Cover is a special case of Set Cover.

Set Cover is an important problem because in discrete optimization. In the standard definition the set system is given explicitly. In many applications the set system is implicit, and often exponential in the explicit part of the input; nevertheless such set systems are ubiquitous and one can often obtain exact or approximation algorithms. As an example consider the well known MST problem in graphs. One way to phrase MST is the following: given an edge-weighted graph $G = (V, E)$ find a minimum cost subset of the edges that cover all the cuts of $G$; by cover a cut $S \subseteq V$ we mean that at least one of the edges in $\delta(S)$ must be chosen. This may appear to be a strange way of looking at the MST problem but this view is useful as we will see later. Another implicit example is the following. Suppose we are given $n$ rectangles in the plane and the goal is to choose a minimum number of points in the plane such that each input rectangle contains one of the chosen points. This is perhaps more natural to view as a special case of the Hitting Set problem. In principle the set of points that we can choose from is infinite but it can be seen that we can confine our attention to vertices in the arrangement of the given rectangles and it is easy to see that there are only $O(n^2)$ vertices — however, explicitly computing them may be expensive and one may want to treat the problem as an implicit one for the sake of efficiency.

Covering problems have the feature that a superset of a feasible solution is also a feasible solution. More abstractly one can cast covering problems as the following. We are given a finite ground set $V$ (vertices in a graph or sets in a set system) and a family of feasible solutions $I \subseteq 2^V$ where $I$ is upward closed; by this we mean that if $A \in I$ and $A \subseteq B$ then $B \in I$. The goal is to find the smallest cardinality set $A$ in $I$. In the weighted case $V$ has weights and the goal is to find a minimum weight set in $I$. In some case one can also consider more complex non-additive objectives that assign a cost $c(S)$ for each $S \in I$.

2.1 Greedy for Set Cover and Maximum Coverage

In this section we consider the unweighted version of Set Cover.

2.1.1 Greedy Algorithm

A natural greedy approximation algorithm for these problems is easy to come up with.
CHAPTER 2. COVERING PROBLEMS

Greedy Cover\((U, S)\)

1. repeat
   A. pick the set that covers the maximum number of uncovered elements
   B. mark elements in the chosen set as covered
2. until done

In case of Set Cover, the algorithm Greedy Cover is done when all the elements in set \(U\) have been covered. And in case of Maximum Coverage, the algorithm is done when exactly \(k\) subsets have been selected from \(S\).

We will prove the following theorem.

Theorem 2.1. Greedy Cover is a \(1 - (1 - 1/k)^k \geq (1 - 1/e) \approx 0.632\) approximation for Maximum Coverage, and a \((\ln n + 1)\) approximation for Set Cover.

The following theorem due to Feige [8] implies that Greedy Cover is essentially the best possible in terms of the approximation ratio that it guarantees.

Theorem 2.2. Unless \(NP \subseteq \text{DTIME}(n^{O(\log \log n)})\), there is no \((1 - o(1))\ln n\) approximation for Set Cover. Unless \(P = NP\), for any fixed \(\epsilon > 0\), there is no \((1 - 1/e - \epsilon)\) approximation for Maximum Coverage.

Recently the preceding theorem has been strengthened so that the hardness holds under the assumption that \(NP \neq P\) [20].

2.1.2 Analysis of Greedy Cover

We proceed towards the proof of Theorem 2.1 by providing analysis of Greedy Cover separately for Set Cover and Maximum Coverage.

Analysis for Maximum Coverage

Let \(OPT\) denote the value of an optimal solution to the Maximum Coverage problem; this is the maximum number of elements that are covered by \(k\) sets in the given set system. Let \(x_i\) denote the number of new elements covered by the \(i\)-th set chosen by Greedy Cover. Also, let \(y_i = \sum_{j=1}^{i} x_i\) be the total number of elements covered after \(i\) iterations, and \(z_i = OPT - y_i\). Note that, according to our notation, \(y_0 = 0\) and \(y_k\) is the number of elements chosen by Greedy Cover at the end of the algorithm, and \(z_0 = OPT\). The key to the analysis is the following simple claim.
Claim 2.1.1. For $0 \leq i < k$, $x_{i+1} \geq \frac{z_i}{k}$.

Proof. Let $F^* \subseteq U$ be the elements covered by some fixed optimum solution; we have $|F^*| = \text{OPT}$. Consider iteration $i + 1$. Greedy Cover selects the subset $S_j$ whose inclusion covers the maximum number of uncovered elements. Since $z_i$ is the total number of elements covered up to iteration $i$, at least $\text{OPT} - z_i$ elements from $F^*$ are uncovered. Let the set of uncovered elements from $F^*$ at the end of iteration $i$ be $F^*_i$. Since $k$ sets together cover $F^*$, and hence $F^*_i$ as well, there must be some set in that collection of $k$ sets that covers at least $|F^*_i|/k$ elements. This is a candidate set that can be chosen in iteration $i + 1$. Since the algorithm picks the set that covers the maximum number of uncovered elements, the chosen set in iteration $i + 1$ covers at least $|F^*_i|/k = \frac{z_i}{k}$ uncovered elements. Hence, $x_{i+1} \geq \frac{z_i}{k}$. ■

Remark 2.1. It is tempting to make a stronger claim that $x_{i+1} \geq \frac{z_i}{k}$. This is however false, and it is worthwhile to come up with an example.

By definition we have $y_k = x_1 + x_2 + \ldots + x_k$ is the total number of elements covered by Greedy Cover. To analyze the worst-case we want to make this sum as small as possible given the preceding claim. Heuristically (which one can formalize), one can argue that choosing $x_{i+1} = \frac{z_i}{k}$ minimizes the sum. Using this one can argue that the sum is at least $(1 - (1 - 1/k)^i) \cdot \text{OPT}$. We give a formal argument now.

Claim 2.1.2. For $i \geq 0$, $z_i \leq (1 - \frac{1}{k})^i \cdot \text{OPT}$.

Proof. By induction on $i$. The claim is trivially true for $i = 0$ since $z_0 = \text{OPT}$. We assume inductively that $z_i \leq (1 - \frac{1}{k})^i \cdot \text{OPT}$. Then

$$z_{i+1} = z_i - x_{i+1}$$

$$\leq z_i (1 - \frac{1}{k}) \quad [\text{using Claim 2.1.1}]$$

$$\leq (1 - \frac{1}{k})^{i+1} \cdot \text{OPT}. \quad \blacksquare$$

The preceding claims yield the following lemma for algorithm Greedy Cover when applied on Maximum Coverage.

Lemma 2.1. Greedy Cover is a $1 - (1 - 1/k)^k \geq 1 - \frac{1}{e}$ approximation for Maximum Coverage.

Proof. It follows from Claim 2.1.2 that $z_k \leq (1 - \frac{1}{k})^k \cdot \text{OPT} \leq \frac{\text{OPT}}{e}$. Hence, $y_k = \text{OPT} - z_k \geq (1 - \frac{1}{e}) \cdot \text{OPT}$. ■

We note that $(1 - 1/e) \approx 0.632$. 

Analysis for Set Cover

Let \( k^* \) denote the value of an optimal solution to the Set Cover problem. Then an optimal solution value to the Maximum Coverage problem with the same system and \( k = k^* \) would be \( n = |U| \) since it is possible to cover all the \( n \) elements in set \( U \) with \( k^* \) sets. From our previous analysis, \( z_k^* \leq \frac{n}{k} \). Therefore, at most \( \frac{n}{k} \) elements would remain uncovered after the first \( k^* \) steps of Greedy Cover. Similarly, after \( 2 \cdot k^* \) steps of Greedy Cover, at most \( \frac{n}{k} \) elements would remain uncovered. This easy intuition convinces us that Greedy Cover is a \( (\ln \frac{n}{k} + 1) \) approximation for the Set Cover problem. A more formal proof is given below.

**Lemma 2.2.** Greedy Cover is a \( (\ln \frac{n}{k} + 1) \) approximation for Set Cover.

**Proof.** Since \( z_l \leq (1 - \frac{1}{k^*})^l \cdot n \), after \( t = \lceil k^* \ln \frac{n}{k^*} \rceil \) steps,

\[
z_t \leq n(1 - 1/k^*)^{k^* \ln \frac{n}{k^*}} \leq ne^{-\ln \frac{n}{k^*}} \leq k^*.
\]

Thus, after \( t \) steps, at most \( k^* \) elements are left to be covered. Since Greedy Cover picks at least one element in each step, it covers all the elements after picking at most \( \lceil k^* \ln \frac{n}{k^*} \rceil + k^* \leq k^*(\ln n + 1) \) sets. \( \blacksquare \)

A useful special case of Set Cover is when all sets are “small”. Does the approximation bound for Greedy improve? We can prove the following corollary via Lemma 2.2.

**Corollary 2.3.** If each set in the set system has at most \( d \) elements, then Greedy Cover is a \( (\ln d + 1) \) approximation for Set Cover.

**Proof.** If each set has at most \( d \) elements then we have that \( k^* \geq \frac{n}{d} \) and hence \( \ln \frac{n}{k^*} \leq \ln d \). Then the claim follows from Lemma 2.2. \( \blacksquare \)

Theorem 2.1 follows directly from Lemma 2.1 and 2.2.

A near-tight example for Greedy Cover when applied on Set Cover: Let us consider a set \( U \) of \( n \) elements along with a collection \( S \) of \( k + 2 \) subsets \( \{R_1, R_2, C_1, C_2, \ldots, C_k\} \) of \( U \). Let us also assume that \( |C_i| = 2^i \) and \( |R_1 \cap C_i| = |R_2 \cap C_i| = 2^{i-1} \) (\( 1 \leq i \leq k \)), as illustrated in Fig. 2.1.

Clearly, the optimal solution consists of only two sets, i.e., \( R_1 \) and \( R_2 \). Hence, \( \text{OPT} = 2 \). However, Greedy Cover will pick each of the remaining \( k \) sets, namely \( C_k, C_{k-1}, \ldots, C_1 \). Since \( n = 2 \cdot \sum_{i=0}^{k-1} 2^i = 2 \cdot (2^k - 1) \), we get \( k = \Omega(\log n) \). One can construct tighter examples with more involved analysis.
Exercise 2.1. Consider the weighted version of the Set Cover problem where a weight function $w : S \to \mathcal{R}^+$ is given, and we want to select a collection $S'$ of subsets from $S$ such that $\bigcup_{X \in S'} X = \mathcal{U}$, and $\sum_{X \in S'} w(X)$ is the minimum. One can generalize the greedy heuristic in the natural fashion where in each iteration the algorithm picks the set that maximizes the ratio of the number of elements to its weight. Adapt the unweighted analysis to prove that the greedy algorithm yields an $O(\ln n)$ approximation for the weighted version (you can be sloppy with the constant in front of $\ln n$).

2.1.3 Dominating Set

A dominating set in a graph $G = (V, E)$ is a set $S \subseteq V$ such that for each $u \in V$, either $u \in S$, or some neighbor $v$ of $u$ is in $S$. In other words $S$ covers/dominates all the nodes in $V$. In the Dominating Set problem, the input is a graph $G$ and the goal is to find a smallest sized dominating set in $G$.

Exercise 2.2. 1. Show that Dominating Set is a special case of Set Cover.

2. What is the greedy heuristic when applied to Dominating Set. Prove that it yields an $(\ln (\Delta + 1) + 1)$ approximation where $\Delta$ is the maximum degree in the graph.

3. Show that Set Cover can be reduced in an approximation preserving fashion to Dominating Set. More formally, show that if Dominating Set has an $\alpha(n)$-approximation where $n$ is the number of vertices in the given instance then Set Cover has an $(1 - o(1))\alpha(n)$-approximation.
CHAPTER 2. COVERING PROBLEMS

2.2 VERTEX COVER

We have already seen that the VERTEX COVER problem is a special case of the SET COVER problem. The Greedy algorithm when specialized to VERTEX COVER picks a highest degree vertex, removes it and the covered edges from the graph, and recurses in the remaining graph. It follows that the Greedy algorithm gives an $O(\ln \Delta + 1)$ approximation for the unweighted versions of the VERTEX COVER problem. One can wonder whether the Greedy algorithm has a better worst-case for VERTEX COVER than the analysis suggests. Unfortunately the answer is negative and there are examples where the algorithm outputs a solution with $\Omega(\log n \cdot OPT)$ vertices.

We sketch the construction. Consider a bipartite graph $G = (U, V, E)$ where $U = \{u_1, u_2, \ldots, u_h\}$, $V$ is partitioned into $S_1, S_2, \ldots, S_h$ where $S_i$ has $\lfloor h/i \rfloor$ vertices. Each vertex $v$ in $S_i$ is connected to exactly $i$ distinct vertices of $U$; thus, any vertex $u_i$ is incident to at most one edge from $S_i$. It can be seen that the degree of each vertex $u_i \in U$ is roughly $h$. Clearly $U$ is a vertex cover of $G$ since the graph is bipartite. Convince yourself that the Greedy algorithm will pick all of $V$ starting with the lone vertex in $S_h$ (one may need to break ties to make this happen but the example can be easily perturbed to make this unnecessary). We have $n = \Theta(h \log h)$ and $OPT \leq h$ and Greedy outputs a solution of size $\Omega(h \log h)$.

2.2.1 A 2-approximation for VERTEX COVER

There is a very simple 2-approximation algorithm for the CARDINALITY VERTEX COVER problem.

\begin{algorithm}
\textbf{Matching-VC}(G)
\begin{enumerate}
\item $S \leftarrow \emptyset$
\item Compute a maximal matching $M$ in $G$
\item \textbf{for} each edge $(u, v) \in M$ \textbf{do}
\begin{enumerate}
\item add both $u$ and $v$ to $S$
\end{enumerate}
\item Output $S$
\end{enumerate}
\end{algorithm}

Theorem 2.4. Matching-VC is a 2-approximation algorithm.

The proof of Theorem 2.4 follows from two simple claims.
Claim 2.2.1. Let OPT be the size of the vertex cover in an optimal solution. Then $OPT \geq |M|$. 

Proof. Any vertex cover must contain at least one end point of each edge in $M$ since no two edges in $M$ intersect. Hence $OPT \geq |M|$. ■

Claim 2.2.2. Let $S(M) = \{u, v | (u, v) \in M\}$. Then $S(M)$ is a vertex cover.

Proof. If $S(M)$ is not a vertex cover, then there must be an edge $e \in E$ such that neither of its endpoints are in $M$. But then $e$ can be included in $M$, which contradicts the maximality of $M$. ■

We now finish the proof of Theorem 2.4. Since $S(M)$ is a vertex cover, Claim 2.2.1 implies that $|S(M)| = 2 \cdot |M| \leq 2 \cdot OPT$.

**Weighted Vertex Cover:** The matching based heuristic does not generalize in a straightforward fashion to the weighted case but 2-approximation algorithms for the Weighted Vertex Cover problem can be designed based on LP rounding.

### 2.2.2 Set Cover with small frequencies

Vertex Cover is an instance of Set Cover where each element in $U$ is in at most two sets (in fact, each element was in exactly two sets). This special case of the Set Cover problem admits a 2-approximation algorithm. What would be the case if every element is contained in at most three sets? More generally, given an instance of Set Cover, for each $e \in U$, let $f(e)$ denote the number of sets containing $e$. Let $f = \max_e f(e)$, which we call the maximum frequency.

**Exercise 2.3.** Give an $f$-approximation for Set Cover, where $f$ is the maximum frequency of an element. **Hint:** Follow the approach used for Vertex Cover.

### 2.3 Vertex Cover via LP

Let $G = (V, E)$ be an undirected graph with arc weights $w : V \rightarrow R^+$. We can formulate Vertex Cover as an integer linear programming problem as follows. For each vertex $v$ we have a variable $x_v$. We interpret the variable as follows: if $x_v = 1$ if $v$ is chosen to be included in a vertex cover, otherwise $x_v = 0$. With this interpretation we can easily see that the minimum weight vertex cover can be formulated as the following integer linear program.
\[
\min \sum_{v \in V} w_v x_v \\
\text{subject to} \\
x_u + x_v \geq 1 \quad \forall e = (u, v) \in E \\
x_v \in \{0, 1\} \quad \forall v \in V
\]

However, solving the preceding integer linear program is NP-Hard since it would solve Vertex Cover exactly. Therefore we use Linear Programming (LP) to approximate the optimal solution, \( \text{OPT}(I) \), for the integer program. First, we can relax the constraint \( x_v \in \{0, 1\} \) to \( x_v \in [0, 1] \). It can be further simplified to \( x_v \geq 0, \forall v \in V \).

Thus, a linear programming formulation for Vertex Cover is:

\[
\min \sum_{v \in V} w_v x_v \\
\text{subject to} \\
x_u + x_v \geq 1 \quad \forall e = (u, v) \in E \\
x_v \geq 0
\]

We now use the following algorithm:

**Vertex Cover via LP**

1. Solve LP to obtain an optimal fractional solution \( x^* \)
2. Let \( S = \{v \mid x_v^* \geq \frac{1}{2}\} \)
3. Output \( S \)

**Claim 2.3.1.** \( S \) is a vertex cover.

**Proof.** Consider any edge, \( e = (u, v) \). By feasibility of \( x^* \), \( x_u^* + x_v^* \geq 1 \), and thus \( x_u^* \geq \frac{1}{2} \) or \( x_v^* \geq \frac{1}{2} \). Therefore, at least one of \( u \) and \( v \) will be in \( S \). \( \blacksquare \)

**Claim 2.3.2.** \( w(S) \leq 2 \text{OPT}_{LP}(I) \).

**Proof.** \( \text{OPT}_{LP}(I) = \sum_u w_u x_u^* \geq \frac{1}{2} \sum_{v \in S} w_v = \frac{1}{2} w(S) \). \( \blacksquare \)

Therefore, \( \text{OPT}_{LP}(I) \geq \frac{\text{OPT}(I)}{2} \) for all instances \( I \).

**Remark 2.2.** For minimization problems: \( \text{OPT}_{LP}(I) \leq \text{OPT}(I) \), where \( \text{OPT}_{LP}(I) \) is the optimal solution found by LP; for maximization problems, \( \text{OPT}_{LP}(I) \geq \text{OPT}(I) \).
Integrality Gap: We introduce the notion of integrality gap to show the best approximation guarantee we can obtain if we only use the LP values as a lower bound.

**Definition 2.5.** For a minimization problem $\Pi$, the integrality gap for a linear programming relaxation/formulation $LP$ for $\Pi$ is $\sup_{I \in \text{OPT}} \frac{\text{OPT}(I)}{\text{OPT}_{LP}(I)}$.

That is, the integrality gap is the worst case ratio, over all instances $I$ of $\Pi$, of the integral optimal value and the fractional optimal value. Note that different linear programming formulations for the same problem may have different integrality gaps.

Claims 2.3.1 and 2.3.2 show that the integrality gap of the Vertex Cover LP formulation above is at most 2.

**Question:** Is this bound tight for the Vertex Cover LP?

Consider the following example: Take a complete graph, $K_n$, with $n$ vertices, and each vertex has $w_v = 1$. It is clear that we have to choose $n - 1$ vertices to cover all the edges. Thus, $\text{OPT}(K_n) = n - 1$. However, $x_v = \frac{1}{2}$ for each $v$ is a feasible solution to the LP, which has a total weight of $\frac{n}{2}$. So gap is $2 - \frac{1}{n}$, which tends to 2 as $n \to \infty$. One can also prove that the integrality gap is essentially 2 even in a class of sparse graphs.

**Exercise 2.4.** The vertex cover problem can be solved optimally in polynomial time in bipartite graphs. In fact the LP is integral. Prove this via the maxflow-mincut theorem and the integrality of flows when capacities are integral.

**Other Results on Vertex Cover**

1. The current best approximation ratio for Vertex Cover is $2 - \Theta(\frac{1}{\sqrt{\log n}})$ [14].

2. It is known that unless $P = NP$ there is $\alpha$-approximation for Vertex Cover for $\alpha < 1.36$ [7]. Under a stronger hypothesis called the Unique Games Conjecture it is known that there is no $2 - \epsilon$ approximation for any fixed $\epsilon > 0$ [16].

3. There is a polynomial time approximation scheme (PTAS), that is a $(1 + \epsilon)$-approximation for any fixed $\epsilon > 0$, for planar graphs. This follows from a general approach due to Baker [3]. The theorem extends to more general classes of graphs.
2.4 Set Cover via LP

The input to the Set Cover problem consists of a finite set \( U = \{1, 2, ..., n\} \), and \( m \) subsets of \( U \), \( S_1, S_2, ..., S_n \). Each set \( S_j \) has a non-negative weight \( w_j \) and the goal is to find the minimum weight collection of sets which cover all elements in \( U \) (in other words their union is \( U \)).

A linear programming relaxation for Set Cover is:

\[
\begin{align*}
\text{min} & \quad \sum_j w_j x_j \\
\text{subject to} & \quad \sum_{j: i \in S_j} x_j \geq 1 \quad \forall i \in \{1, 2, ..., n\} \\
& \quad x_j \geq 0 \quad 1 \leq j \leq m
\end{align*}
\]

And its dual is:

\[
\begin{align*}
\text{max} & \quad \sum_{i=1}^n y_i \\
\text{subject to} & \quad \sum_{i \in S_j} y_i \leq w_j \quad \forall i \in \{1, 2, ..., n\} \\
& \quad y_i \geq 0 \quad \forall i \in 1, 2, ..., n
\end{align*}
\]

We give several algorithms for Set Cover based on this primal/dual pair LPs.

2.4.1 Deterministic Rounding

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Set Cover via LP} \\
1. Solve LP to obtain an optimal solution \( x^* \), which contains fractional numbers. \\
2. Let \( P = \{i \mid x_i^* > 0\} \) \\
3. Output \( \{S_j \mid j \in P\} \) \\
\hline
\end{tabular}
\end{center}

Note that the above algorithm, even when specialized to Vertex Cover is different from the one we saw earlier. It includes all sets which have a strictly positive value in an optimum solution to the LP.
Let $x^*$ be an optimal solution to the primal LP, $y^*$ be an optimum solution to the dual, and let $P = \{ j \mid x_j^* > 0 \}$. First, note that by strong duality, $\sum_j w_j x_j^* = \sum_i y_i^*$. Second, by complementary slackness if $x_j^* > 0$ then the corresponding dual constraint is tight, that is $\sum_{i \in S_j} y_i^* = w_j$.

**Claim 2.4.1.** The output of the algorithm is a feasible set cover for the given instance.

*Proof.* Exercise.

**Claim 2.4.2.** $\sum_{j \in P} w_j \leq f \sum_j w_j x_j^* = \text{OPT}_{LP}$.  

*Proof.*

\[
\sum_{j \in P} w_j = \sum_{j : x_j^* > 0} w_j = \sum_{j : x_j^* > 0} \left( \sum_{i \in S_j} y_i^* \right) = \sum_i y_i^\top \left( \sum_{j : i \in S_j, x_j^* > 0} 1 \right) \leq f \sum_i y_i^* = f \text{OPT}_{LP}(I).
\]

Notice that the second equality is due to complementary slackness conditions (if $x_j^* > 0$, the corresponding dual constraint is tight), the penultimate inequality uses the definition of $f$, and the last inequality follows from weak duality (a feasible solution for the dual problem is a lower bound on the optimal primal solution).

Therefore we have that the algorithm outputs a cover of weight at most $f \text{OPT}_{LP}$. We note that $f$ can be as large as $n$ in which case the bound given by the algorithm is quite weak. In fact, it is not hard to construct examples that demonstrate the tightness of the analysis.

**Remark 2.3.** The analysis crucially uses the fact that $x^*$ is an optimal solution. On the other hand the algorithm for **Vertex Cover** is more robust and works with any feasible solution $x$. It is easy to generalize the earlier rounding for **Vertex Cover** to obtain an $f$-approximation. The point of the above rounding is to illustrate the utility of complementary slackness.

### 2.4.2 Randomized Rounding

Now we describe a different rounding that yields an approximation bound that does not depend on $f$. 
### Set Cover via Randomized Rounding

1. Let \( A = \emptyset \), and let \( x^* \) be an optimal solution to the LP.
2. For \( k = 1 \) to \( 2 \ln n \) do
   - A. pick each \( S_j \) independently with probability \( x^*_j \)
   - B. if \( S_j \) is picked, \( A = A \cup \{j\} \)
3. Output the sets with indices in \( A \)

---

**Claim 2.4.3.** \( \mathbb{P}[^{i \text{ is not covered in an iteration}}] = \prod_{j \in S_j} (1 - x^*_j) \leq \frac{1}{e} \).

Intuition: We know that \( \sum_{j \in S_j} x^*_j \geq 1 \). Subject to this constraint, if we want to minimize the probability that element \( i \) is covered, one can see that the minimum is achieved with \( x^*_j = 1/\ell \) for each set \( S_j \) that covers \( i \); here \( \ell \) is the number of sets that cover \( i \). Then the probability is \( (1 - 1/\ell)^{\ell} \).

**Proof.** We use the inequality \( (1 - x) \leq e^{-x} \) for all \( x \in [0, 1] \).

\[
\mathbb{P}[^{i \text{ is not covered in an iteration}}] = \prod_{j \in S_j} (1 - x^*_j) \leq \prod_{j \in S_j} e^{-x^*_j} \leq e^{-\sum_{j \in S_j} x^*_j} \leq \frac{1}{e}.
\]

---

We then obtain the following corollaries:

**Corollary 2.6.** \( \mathbb{P}[^{i \text{ is not covered at the end of the algorithm}}] \leq e^{-2\ln n} \leq \frac{1}{n^2} \).

**Corollary 2.7.** \( \mathbb{P}[^{all elements are covered, after the algorithm stops}] \geq 1 - \frac{1}{n} \).

**Proof.** Via the union bound. The probability that \( i \) is not covered is at most \( 1/n^2 \), hence the probability that there is some \( i \) that is not covered is at most \( n \cdot 1/n^2 \leq 1/n \).

Now we bound the expected cost of the algorithm. Let \( C_t = \text{cost of sets picked in iteration } t \), then \( \mathbb{E}[C_t] = \sum_{j=1}^{w_j} j x^*_j \), where \( \mathbb{E}[X] \) denotes the expectation of a random variable \( X \). Then, let \( C = \sum_{t=1}^{2\ln n} C_t \); we have \( \mathbb{E}[C] = \sum_{t=1}^{2\ln n} \mathbb{E}[C_t] \leq 2\ln n \mathbb{O}T_{LP} \). By Markov’s inequality, \( \mathbb{P}[C > 2\mathbb{E}[C]] \leq \frac{1}{2} \), hence \( \mathbb{P}[C \leq 4\ln n \mathbb{O}T_{LP}] \geq \frac{1}{2} \). Therefore, \( \mathbb{P}[C \leq 4\ln n \mathbb{O}T_{LP} \text{ and all items are covered}] \geq \frac{1}{2} - \frac{1}{n} \). Thus, the randomized rounding algorithm, with probability close to 1/2.
succeeds in giving a feasible solution of cost $O(\log n) \OPT_{LP}$. Note that we can check whether the solution satisfies the desired properties (feasibility and cost) and repeat the algorithm if it does not.

1. We can check if solution after rounding satisfies the desired properties, such as all elements are covered, or cost at most $2c \log n \OPT_{LP}$. If not, repeat rounding. Expected number of iterations to succeed is a constant.

2. We can also use Chernoff bounds (large deviation bounds) to show that a single rounding succeeds with high probability (probability at least $1 - \frac{1}{\text{poly}(n)}$).

3. The algorithm can be derandomized. Derandomization is a technique of removing randomness or using as little randomness as possible. There are many derandomization techniques, such as the method of conditional expectation, discrepancy theory, and expander graphs.

4. After a few rounds, select the cheapest set that covers each uncovered element. This has low expected cost. This algorithm ensures feasibility but guarantees cost only in the expected sense. We will see a variant on the homework.

**Randomized Rounding with Alteration:** In the preceding analysis we had to worry about the probability of covering all the elements and the expected cost of the solution. Here we illustrate a simple yet powerful technique of alteration in randomized algorithms and analysis. Let $d$ be the maximum set size.

**Set Cover: Randomized Rounding with Alteration**

1. $A = \emptyset$, and let $x^*$ be an optimal solution to the LP
2. Add to $A$ each $S_j$ independently with probability $\min\{1, \ln d \cdot x^*_j\}$
3. Let $U'$ be the elements uncovered by the chosen sets in $A$
4. For each uncovered element $i \in U'$ do
   A. Add to $A$ the cheapest set that covers $i$
5. Output the sets with indices in $A$

The algorithm has two phases. A randomized phase and a fixing/altering phase. In the second phase we apply a naive algorithm that may have a high cost in the worst case but we will bound its expected cost appropriately. The
algorithm deterministically guarantees that all elements will be covered, and hence we only need to focus on the expected cost of the chosen sets. Let $C_1$ be the random cost of the sets chosen in the first phase and let $C_2$ be the random cost of the sets chosen in the second phase. It is easy to see that $E[C_1] = \ln d \sum_j w_j x_j^* = \ln d \text{OPT}_{LP}$. Let $E_i$ be the event that element $i$ is not covered after the first randomized phase.

Exercise 2.5. $P[E_i] \leq e^{-\ln d} \leq 1/d$.

The worst case second phase cost can be upper bounded via the next lemma.

**Lemma 2.3.** Let $\beta_i$ be the cost of the cheapest set covering $i$. Then $\sum_i \beta_i \leq d \text{OPT}_{LP}$.

**Proof.** Consider an element $i$. We have the constraint that $\sum_{j : i \in S_j} x_j^* \geq 1$. Since each set covering $i$ has cost at least $\beta_i$, we have $\sum_{j : i \in S_j} c_j x_j^* \geq \beta_i \sum_{j : i \in S_j} x_j^* \geq \beta_i$. Thus,

$$\sum_i \beta_i \leq \sum_i \sum_{j : i \in S_j} c_j x_j^* \leq \sum_j c_j x_j^* |S_j| \leq d \sum_j c_j x_j^* = d \text{OPT}_{LP}.$$ 

Now we bound the expected second phase cost.

**Lemma 2.4.** $E[C_2] \leq \text{OPT}_{LP}$.

**Proof.** We pay for a set to cover element $i$ in the second phase only if it is not covered in the first phase. Hence $C_2 = \sum_i E_i \beta_i$. Note that the events $E_i$ for different elements $i$ are not necessarily independent, however, we can apply linearity of expectation.

$$E[C_2] = \sum_i E[E_i] \beta_i = \sum_i P[E_i] \beta_i \leq 1/d \sum_i \beta_i \leq \text{OPT}_{LP}.$$ 

Combining the expected costs of the two phases we obtain the following theorem.

**Theorem 2.8.** Randomized rounding with alteration outputs a feasible solution of expected cost $(1 + \ln d) \text{OPT}_{LP}$.

Note that the simplicity of the algorithm and tightness of the bound.
2.4.3 Dual-fitting

In this section, we introduce the technique of dual-fitting for the analysis of approximation algorithms. At a high-level the approach is the following:

1. Consider an algorithm that one wants to analyze.
2. Construct a feasible solution to the dual LP based on the structure of the algorithm.
3. Show that the cost of the solution returned by the algorithm can be bounded in terms of the value of the dual solution.

Note that the algorithm itself need not be LP based. Here, we use Set Cover as an example. See the previous section for the primal and dual LP formulations for Set Cover.

We can interpret the dual as follows: Think of $y_i$ as how much element $i$ is willing to pay to be covered; the dual maximizes the total payment, subject to the constraint that for each set, the total payment of elements in that set is at most the cost of the set.

We rewrite the Greedy algorithm for Weighted Set Cover.

<table>
<thead>
<tr>
<th>Greedy Set Cover</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $Covered = \emptyset$</td>
</tr>
<tr>
<td>2. $A = \emptyset$;</td>
</tr>
<tr>
<td>3. While $Covered \neq U$ do</td>
</tr>
<tr>
<td>A. $j \leftarrow \arg \min_k \left( \frac{w_k}{</td>
</tr>
<tr>
<td>B. $Covered = Covered \cup S_j$;</td>
</tr>
<tr>
<td>C. $A = A \cup {j}$.</td>
</tr>
<tr>
<td>4. end while;</td>
</tr>
<tr>
<td>5. Output sets in $A$ as cover</td>
</tr>
</tbody>
</table>

Let $H_k = 1 + 1/2 + \ldots + 1/k$ be the $k$th Harmonic number. It is well known that $H_k \leq 1 + \ln k$.

**Theorem 2.9.** Greedy Set Cover picks a solution of cost $\leq H_d \cdot \text{OPT}_{LP}$, where $d$ is the maximum set size, i.e., $d = \max_j |S_j|$. 

To prove this, we augment the algorithm to keep track of some additional information.

**Augmented Greedy Algorithm of Weighted Set Cover**

1. `Covered = ∅`
2. While `Covered ≠ U` do
   a. `j ← arg min_k w_k / |S_k ∩ Uncovered|`
   b. if `i` is uncovered and `i ∈ S_j`, set `p_i = w_i / |S_j ∩ Uncovered|`;
   c. `Covered = Covered ∪ S_j`
   d. `A = A ∪ {j}`.
3. Output sets in `A` as cover

It is easy to see that the algorithm outputs a feasible cover.

**Claim 2.4.4.** \( \sum_{j ∈ A} w_j = \sum_{i} p_i. \)

**Proof.** Consider when `j` is added to `A`. Let \( S'_j \subseteq S_j \) be the elements that are uncovered before `j` is added. For each `i ∈ S'_j` the algorithm sets `p_i = w_i / |S'_j|`. Hence, \( \sum_{i ∈ S'_j} p_i = w_j. \) Moreover, it is easy to see that the sets \( S'_j, j ∈ A \) are disjoint and together partition `U`. Therefore,

\[
\sum_{j ∈ A} w_j = \sum_{j ∈ A} \sum_{i ∈ S'_j} p_i = \sum_{i ∈ U} p_i.
\]

For each `i`, let \( y'_i = \frac{1}{p_i} p_i. \)

**Claim 2.4.5.** \( y' \) is a feasible solution for the dual LP.

Suppose the claim is true, then the cost of Greedy Set Cover’s solution = \( \sum_i p_i = H_d \sum_i y'_i ≤ H_d \text{OPT}_{L.P.} \). The last step is because any feasible solution for the dual problem is a lower bound on the value of the primal LP (weak duality).

Now, we prove the claim. Let \( S_j \) be an arbitrary set, and let \( |S_j| = t ≤ d \). Let \( S_j = \{i_1, i_2, ..., i_t\} \), where we the elements are ordered such that \( i_1 \) is covered by Greedy no-later than \( i_2 \), and \( i_2 \) is covered no later than \( i_3 \) and so on.

**Claim 2.4.6.** For \( 1 ≤ h ≤ t \), \( p_{i_h} ≤ \frac{w_j}{t-h+1}. \)
Proof. Let \( S' \) be the set that covers \( i_h \) in Greedy. When Greedy picked \( S' \) the elements \( i_h, i_{h+1}, \ldots, i_t \) from \( S_j \) were uncovered and hence Greedy could have picked \( S_j \) as well. This implies that the density of \( S' \) when it was picked was no more than \( \frac{w_j}{t-h+1} \). Therefore \( p_{i_h} \) which is set to the density of \( S' \) is at most \( \frac{w_j}{t-h+1} \).

\[
\sum_{1 \leq h \leq t} p_{i_h} \leq \sum_{1 \leq h \leq t} \frac{w_j}{t-h+1} = w_j H_t \leq w_j H_d.
\]

Thus, the setting of \( y'_h \) to be \( p_i \) scaled down by a factor of \( H_d \) gives a feasible solution.

2.4.4 Greedy for implicit instances of Set Cover

Set Cover and the Greedy heuristic are quite useful in applications because many instances are implicit, nevertheless, the algorithm and the analysis applies. That is, the universe \( U \) of elements and the collection \( S \) of subsets of \( U \) need not be restricted to be finite or explicitly enumerated in the Set Cover problem. For instance, a problem could require covering a finite set of points in the plane using disks of unit radius. There is an infinite set of such disks, but the Greedy approximation algorithm can still be applied. For such implicit instances, the Greedy algorithm can be used if we have access to an oracle, which, at each iteration, selects a set having the optimal density. However, an oracle may not always be capable of selecting an optimal set. In some cases it may have to make the selections approximately. We call an oracle an \( \alpha \)-approximate oracle for some \( \alpha \geq 1 \) if, at each iteration, it selects a set \( S \) such that \( \frac{w(S)}{S} \leq \alpha \min_A \frac{A}{w(A)} \).

Exercise 2.6. Prove that the approximation guarantee of Greedy with an \( \alpha \)-approximate oracle would be \( \alpha (\ln n + 1) \) for Set Cover, and \( (1 - \frac{1}{e^\alpha}) \) for Maximum Coverage.

We will see several examples of implicit use of the greedy analysis in the course.

2.5 Submodularity

Set Cover turns out to be a special case of a more general problem called Submodular Set Cover. The Greedy algorithm and analysis applies in this more generality. Submodularity is a fundamental notion with many applications in
combinatorial optimization and else where. Here we take the opportunity to provide some definitions and a few results.

**Definition 2.10.** Given a finite set $E$, a real-valued set function $f : 2^E \to \mathbb{R}$ is submodular iff

$$f(A) + f(B) \geq f(A \cup B) + f(A \cap B) \quad \forall A, B \subseteq E.$$ 

Alternatively, $f$ is a submodular function iff

$$f(A \cup \{i\}) - f(A) \geq f(B \cup \{i\}) - f(B) \quad \forall A \subset B, i \in E \setminus B.$$ 

The second characterization shows that submodularity is based on *decreasing marginal utility* property in the discrete setting. Adding element $i$ to a set $A$ will help at least as much as adding it to to a (larger) set $B \supseteq A$. It is common to use $A + i$ to denote $A \cup \{i\}$ and $A - i$ to denote $A \setminus \{i\}$.

**Exercise 2.7.** Prove that the two characterizations of submodular functions are equivalent.

Many application of submodular functions are when $f$ is a non-negative function though there are several important applications when $f$ can be negative. A submodular function $f(\cdot)$ is *monotone* if $f(A + i) \geq f(A)$ for all $i \in E$ and $A \subseteq E$. Typically one assumes that $f$ is normalized by which we mean that $f(\emptyset) = 0$; this can always be done by shifting the function by $f(\emptyset)$. $f$ is *symmetric* if $f(A) = f(E \setminus A)$ for all $A \subseteq E$. Submodular set functions arise in a large number of fields including combinatorial optimization, probability, and geometry. Examples include rank function of a matroid, the sizes of cutsets in a directed or undirected graph, the probability that a subset of events do not occur simultaneously, entropy of random variables, etc. In the following we show that the Set Cover and Maximum Coverage problems can be easily formulated in terms of submodular set functions.

**Exercise 2.8.** Let $\mathcal{U}$ be a set and let $S = \{S_1, S_2, \ldots, S_m\}$ be a finite collection of subsets of $\mathcal{U}$. Let $N = \{1, 2, \ldots, m\}$, and define $f : 2^N \to \mathbb{R}$ as: $f(A) = |\bigcup_{i \in A} S_i|$ for $A \subseteq E$. Show that $f$ is a monotone non-negative submodular set function.

**Exercise 2.9.** Let $G = (V, E)$ be a directed graph and let $f : 2^V \to \mathbb{R}$ where $f(S) = |\delta^+(S)|$ is the number of arcs leaving $S$. Prove that $f$ is submodular. Is the function monotone?

### 2.5.1 Submodular Set Cover

When formulated in terms of submodular set functions, the Set Cover problem is the following. Given a monotone submodular function $f$ (whose value would
be computed by an oracle) on \( N = \{1, 2, \ldots, m\} \), find the smallest set \( S \subseteq N \) such that \( f(S) = f(N) \). Our previous greedy approximation can be applied to this formulation as follows.

**Greedy Submodular \((f, N)\)**

1. \( S \leftarrow \emptyset \)
2. **While** \( f(S) \neq f(N) \) do
   
   A. find \( i \) to maximize \( f(S + i) - f(S) \)
   
   B. \( S \leftarrow S \cup \{i\} \)
3. Output \( S \)

Not so easy exercise.

**Exercise 2.10.**

1. Prove that the greedy algorithm is a \( 1 + \ln(f(N)) \) approximation for **Submodular Set Cover**.

2. Prove that the greedy algorithm is a \( 1 + \ln(\max_i f(i)) \) approximation for **Submodular Set Cover**.

The above results were first obtained by Wolsey [25].

### 2.5.2 Submodular Maximum Coverage

By formulating the **Maximum Coverage** problem in terms of submodular functions, we seek to maximize \( f(S) \) such that \( |S| \leq k \). We can apply algorithm **Greedy Submodular** for this problem by changing the condition in line 2 to be: **while** \( |S| \leq k \).

**Exercise 2.11.** Prove that greedy gives a \( (1 - 1/e) \)-approximation for **Submodular Maximum Coverage** problem when \( f \) is monotone and non-negative. **Hint:** Generalize the main claim that we used for **Maximum Coverage**.

The above and many related results were shown in the influential papers of Fisher, Nemhauser and Wolsey [10, 21].

### 2.6 Covering Integer Programs (CIPs)

There are several extensions of **Set Cover** that are interesting and useful. **Submodular Set Cover** is a very general problem while there are intermediate
problems of interest such as Set Mmulticover. We refer to the reader to the relevant chapters in the two reference books. Here we refer to a general problem called Covering Integer Programs (CIPs for short). The goal is to solve the following integer program where $A \in \mathbb{R}^{n \times m}$ is a non-negative matrix. We can assume without loss of generality that $w$ and $b$ are also non-negative.

$$
\begin{align*}
\text{min} & \quad \sum_{j=1}^{n} w_j x_j \\
\text{subject to} & \quad Ax \geq b \\
& \quad x_j \leq d_j \quad 1 \leq j \leq m \\
& \quad x_j \geq 0 \quad 1 \leq j \leq m \\
& \quad x_j \in \mathbb{Z} \quad 1 \leq j \leq m
\end{align*}
$$

$Ax \geq b$ model covering constraints and $x_j \leq d_j$ models multiplicity constraints. Note that Set Cover is a special case where $A$ is simply the incidence matrix of the sets and elements (the columns correspond to sets and the rows to elements) and $d_j = 1$ for all $j$. What are CIPs modeling? It is a generalization of Set Cover. To see this, assume, without loss of generality, that $A$, $b$ are integer matrices. For each element corresponding to row $i$ the quantity $b_i$ corresponds to the requirement of how many times $i$ needs to be covered. $A_{ij}$ corresponds to the number of times set $S_j$ covers element $i$. $d_j$ is an upper bound on the number of copies of set $S_j$ that are allowed to be picked.

**Exercise 2.12.** Prove that CIPs are a special case of Submodular Set Cover.

One can apply the Greedy algorithm to the above problem and the standard analysis shows that the approximation ratio obtained is $O(\log B)$ where $B = \sum_i b_i$ (assuming that they are integers). Even though this is reasonable we would prefer a strongly polynomial bound. In fact there are instances where $B$ is exponential in $n$ and the worst-case approximation ratio can be poor. The natural LP relaxation of the above integer program has a large integrality gap in constrast to the case of Set Cover. One needs to strengthen the LP relaxation via what are known as knapsack cover inequalities. We refer the reader to the paper of Kolliopoulos and Young [17] and recent one by Chekuri and Quanrud [5] for more on this problem.
Bibliography


[16] Subhash Khot and Oded Regev. “Vertex cover might be hard to approximate to within $2-\epsilon$”. In: Journal of Computer and System Sciences 74.3 (2008), pp. 335–349.


