APPROXIMATION ALGORITHMS FOR SET COVER AND RELATED PROBLEMS

By

Petr Slavík

April 1998

A DISSERTATION SUBMITTED TO THE

FACULTY OF THE GRADUATE SCHOOL OF STATE

UNIVERSITY OF NEW YORK AT BUFFALO

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

To the memory of my father

Abstract

In this thesis, we analyze several known and newly designed algorithms for approximating optimal solutions to NP-hard optimization problems. We give a new analysis of the greedy algorithm for approximating the SET COVER and PARTIAL SET COVER problems obtaining significantly improved performance bounds. We also give a first approximation algorithm with a non-trivial performance bound for the ERRAND SCHEDULING and TREE COVER problems, known also as the GENERALIZED TRAVELING SALESMAN and GROUP STEINER TREE problems.

The main results of this thesis first appeared in my papers [87], [89], [91], and [90]; and in my technical reports [86] and [88].

Acknowledgments

I would like to thank Eugene Kleinberg, my Ph.D. advisor, for many useful discussions and encouragement, and for making me revise this thesis so many times. His suggestions improved this thesis considerably, his support and optimism helped me to believe in myself at times when nothing went right. I only regret that we did not play tennis more often—I have still a lot to learn from him in both tennis and mathematics. I would also like to thank his son, Jon Kleinberg, for his comments, references, and suggestions. In particular, he pointed me towards establishing the result in Theorem 3.6.

Many thanks go to David Williamson. Despite his busy schedule, he was always willing to answer my professional and personal questions concerning research in approximation algorithms and pointed me to many useful references. He also agreed to be the outside reader of my thesis. His comments and suggestions improved the thesis considerably and the additional references he provided allowed me to cite the most recent results in the area of approximation algorithms.

I would like to thank R. Ravi for a useful discussion about the Group Steiner Tree problem, Ken Regan for his support and helpful comments on several drafts of this thesis, and Xin He for his help and support in the beginning of my career in theoretical computer science.

Finally, I am very grateful to my wife, Miho, for her patience during those days when I came home just to sleep. Without her ongoing support and love, this thesis would never have been written.

Contents

List of Figures

Chapter 1

Introduction

A large number of optimization problems that have to be solved in practice are NP-hard. And since it is believed that P $\neq$ NP, these problems are very unlikely to be solved exactly by polynomial-time algorithms. Thus despite the existence of high speed computers, obtaining an optimum solution to any of the NP-hard optimization problems would require years or even centuries of computing time for inputs of only moderate size. However, many of these problems are of practical interest, and the instances that need to be solved are often very large. Thus both researchers and practitioners became interested in designing polynomial-time *approximation algorithms*; that is, algorithms that in polynomial time output only slightly sub-optimal solutions. This thesis is devoted to the design of several approximation algorithms and establishing worst-case bounds on their performance.

Chapter 2 gives the background necessary for understanding the rest of the thesis. Its main purpose is to introduce a non-specialist with a general mathematical background to the basic concepts of graph theory, asymptotic notation, computational complexity, combinatorial optimization, and approximation algorithms. Since this chapter is aimed at non-specialists, experts in the field can safely skip directly to the next chapter.

In Chapter 3, we establish significantly improved bounds on the performance ratio of the greedy algorithm for approximating SET COVER. In particular, we provide the first substantial improvement of the 20 year old classical harmonic upper bound $H(m)$ of

Johnson, Lovász, and Chvátal, by showing that the performance ratio of the greedy algorithm is, in fact, *exactly* $\ln m - \ln\ln m + \Theta(1)$, where m is the size of the ground set and $H(m) = 1 + 1/2 + \cdots + 1/m = \ln m + \Theta(1)$. The additive difference between the upper and lower bounds turns out to be less than 1.1. This provides the first tight analysis of the greedy algorithm, as well as the first upper bound that lies below $H(m)$ by a function going to infinity with m.

In Chapter 4, we prove that the classical bounds on the performance of the greedy algorithm for approximating SET COVER with costs are valid for PARTIAL COVER as well, thus lowering, by more than a factor of two, the previously known estimate given by Kearns.

In Chapter 5, we give the first algorithm with a non-trivial performance bound for approximating the GENERALIZED TRAVELING SALESMAN problem (GTSP) and significantly improve the known results for the GROUP STEINER TREE problem (GROUP-STEINER). We first consider both problems on weighted trees. Under the assumption that the number of cities from the same cluster in certain subtrees is bounded, we obtain an algorithm that approximates both GTSP and GROUP-STEINER within $O(\log m)$, where m is the total number of clusters. In the second part of Chapter 5, we discuss the GENERALIZED TRAVELING SALESMAN and GROUP STEINER TREE problems on general weighted graphs with clusters of size at most ρ. We show that in this case, GTSP can be approximated to within $3\rho/2$ and GROUP-STEINER to within 2ρ.

Chapter 2

Background

In this chapter, we present an overview of the background necessary for understanding the rest of the thesis. It can be safely skipped by readers familiar with computational complexity and approximation algorithms. In order to avoid details unnecessary for understanding the rest of the thesis, we will keep the discussion on a relatively informal level. The interested reader can find a formal introduction to computational complexity and theory of NP-completeness in books by Garey and Johnson [37] and more recently by Papadimitriou [74]. An introduction to combinatorial optimization can be found in books by Lawler [61], Papadimitriou and Steiglitz [75], and by Cook et al. [26]; a nice introduction to approximation algorithms is given by Motwani in his class notes [69]. A catalog of NP-hard optimization problems together with recent approximation and hardness results can be found in [27]. An advanced reader will find useful a survey [84] of recent advances in combinatorial optimization written by Shmoys.

2.1 Basic Concepts

Definition 2.1 (Asymptotic Notation) Let $f(n)$, $g(n)$ be two non-negative functions defined for all positive integers.

1. We say that

$$f(n) = O(g(n))$$

if there exist constants c and N such that $f(n) \leq c\, g(n)$ for all $n \geq N$.

2. We say that

$$f(n) = \Omega(g(n))$$

if there exist constants c and N such that $f(n) \geq c\, g(n)$ for all $n \geq N$.

3. We say that

$$f(n) = \Theta(g(n))$$

if there exist constants c_1, c_2, and N such that $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq N$, that is if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

4. We say that

$$f(n) = o(g(n))$$

if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

5. We say that

$$f(n) = \omega(g(n))$$

if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$

The asymptotic notation is frequently used to express upper and lower bounds on performance guarantees, running times of algorithms, etc. In general, O-notation denotes an upper bound that may or may not be asymptotically tight, o-notation denotes an upper bound that is not asymptotically tight, and similar statements apply to Ω and ω-notation used for lower bounds. Θ-notation is used for denoting upper or lower bounds that are asymptotically tight. For example, we would say that an algorithm that in the worst case needs $4n^2 + 5n$ steps to output a desired solution runs in time $O(n^2)$.

Definition 2.2 (Graph Theory Concepts) An (undirected) *graph* $G = (V, E)$ is a structure consisting of a finite set V whose elements are called *vertices* or *nodes*, and a set E of *unordered* pairs of vertices called *edges*. We say that two vertices $u, v \in V$ are *adjacent* if there is an edge between them; that is, if there exists an edge $e = (u, v) \in E$. We say that two edges are *adjacent* if they have a common end-point. A *directed graph* $G = (V, A)$ consists of a finite set V of vertices and a set A of *ordered* pairs of vertices called *arcs*. In what follows we will mostly deal with undirected graphs. For any (undirected) graph G, we denote by $V = V(G)$ the set of vertices of G, and by $E = E(G)$ the set of edges of G.

Given a graph $G = (V, E)$, a *path* from a vertex u to a vertex v is a sequence of vertices $\langle v_0, v_1, \ldots, v_k \rangle$ for some $k \geq 0$ such that $u = v_0$, $v = v_k$, and $(v_{i-1}, v_i) \in E$. We say that graph G is connected if for any two vertices $u, v \in V$ there exists a path from u to v. A path from v to v is called a *closed path.* A closed path from v to v containing at least one other vertex is called a *cycle* if it traverses each edge in E at most once. A cycle is *simple* if all the vertices $v_1, v_2, \ldots, v_k$ are distinct. A *Hamiltonian cycle* is a simple cycle that contains all vertices in V.

A graph without any cycles is called a *forest.* A connected graph without any cycles is called a *tree.* Thus every connected component of a forest is a tree. A subgraph $G' = (V', E')$ of graph $G = (V, E)$ is a graph such that $V' \subseteq V$ and $E' \subseteq E$. Given a graph $G = (V, E)$, a *spanning tree* T is a subgraph of G which is a tree and $V(T) = V$.

We say that $G = (V, E)$ is a *complete* graph if all vertices of G are adjacent, that is if E contains *all* unordered pairs of vertices. A complete subgraph G' of G is called a *clique.*

An *edge-weighted* graph $G = (V, E, w)$ is a graph with weight w_e associated with each edge $e \in E$. The weight w_e is also referred to as the *length* of edge e. Given an edge-weighted graph $G = (V, E, w)$ the length of a path $\langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the lengths of all edges. Similarly, the length of a tree $T = (V', E') \subseteq G$ is the sum of the lengths of all edges in E'. If the weight function is clear from the context, we will often denote an edge-weighted graph by $G = (V, E)$.

2.2 Theory of NP-completeness

In order to appreciate the beauty and importance of approximation algorithms, it is essential to understand the basics of the theory of computation.

2.2.1 Turing Machine

Even though the concept of a Turing machine is not absolutely necessary for understanding the results in this thesis, we feel that anybody who is interested in the theory of algorithms should be familiar with this model of computation. We will try to keep the exposition brief and discuss only concepts relevant to the theory of NP-completeness. More details can be found in any introductory textbook on the theory of computation or in [37] and [74].

The Turing machine as a formal model of computation was introduced by Alan Turing in 1936. It consists of a *finite control*, an infinitely long *input tape* divided into cells, and a *tape head* that scans one tape cell at a time—see Figure 2.1. At any time, the finite control is in one of the finitely many states. The tape has a leftmost cell but it is infinite to the right. Each cell of the tape holds exactly one of a finite number of tape symbols.

Initially, the tape head scans the leftmost cell of the tape, the finite control is in a special starting state 'START', the n leftmost cells (for some $n \geq 0$) hold the input, and the remaining infinitely many cells each hold the blank symbol 'B'. The input is a string x of input symbols which form a subset of the tape symbols, the blank symbol 'B' is a special tape symbol that is not an input symbol. Thus the (finite) set of tape symbols consists of all the input symbols, the blank symbol, and possibly some other symbols.

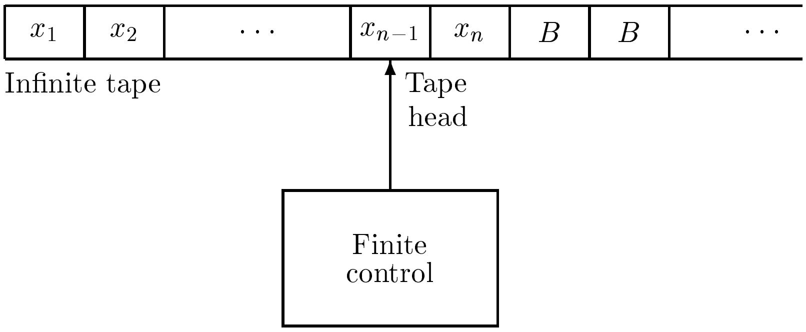


Figure 2.1: One-tape Turing machine

Each move of the Turing machine depends only on the state of the finite control and the symbol scanned by the tape head. The moves of the Turing machine are determined by the *transition function* that can be regarded as a "program" of the Turing machine. In one move, the Turing machine changes its state, prints a tape symbol on the tape cell currently scanned overwriting the previous symbol, and either moves the tape head one cell to the right or to the left, or keeps the head at its current position. Thus the transition function maps a pair $(state, symbol)$ onto a triple $(state, symbol, direction)$.

We denote by $M(x)$ the output of the Turing machine M on input x. Let us now describe what we mean by such output. The (finite) set of machine states contains some or all of the three possible final states—the 'ACCEPT' state, the 'REJECT' state, and the 'HALT' state. When the finite control reaches the 'ACCEPT' state, the machine halts and we say that the input string was accepted. In such case $M(x) =$ "*yes*". Similarly, when the finite control reaches the 'REJECT' state, the machine halts and we say that the string was not accepted. In this case $M(x) =$ "*no*". When the control reaches the 'HALT' state, the machine halts and the string currently written on the tape starting at the leftmost cell and ending one cell before the leftmost blank, is considered an output of the Turing machine. It is also possible that the Turing machine will never halt on a given input, in such case we write $M(x) = \nearrow$.

We can formally define Turing machine as follows:

Definition 2.3 A Turing machine is a quintuple $M = (Q, \Sigma, \Gamma, \delta, F)$, where Q is the finite set of states such that state 'START' belongs to Q, $F \subseteq Q$ is the set of final states, Σ is the set of input symbols not containing the blank symbol B, Γ is the set of tape symbols ($\Sigma \subseteq \Gamma$ and $B \in \Gamma$), and δ is the transition function, $\delta : (Q \setminus F) \times \Gamma \longrightarrow Q \times \Gamma \times D$, where $D = \{\leftarrow, \rightarrow, -\}$.

Note: Σ is often called the *input alphabet* of the Turing machine M, and Γ is called the *tape alphabet.*

Before continuing our description of Turing machines, let us give a simple example to illustrate how such a machine works.

Example 2.1 *Describe a Turing machine that decides whether a given nonempty string of 0's and 1's begins and starts with the same symbol.*

Solution: We give first an informal description. The Turing machine M could work as follows:

1. Read the first symbol of the input
2. If the input is empty—reject.
3. Else remember the symbol read in the final control; that is, go to state q_0 if 0 is the first symbol, or go to state q_1 if 1 was the first symbol.
4. Scan to the end of the string; that is, to the first blank.
5. The fact that we reached the end of the string has to again be "remembered" in the finite control, so if we are in state q_0 we change to state q_2 and from state q_1 we change to state q_3. In either case the head moves one symbol back so that it is now scanning the last symbol of the string.

6. If the last input symbol is 0 and we are in state q_2—accept, if the last input symbol is 1 and we are in state q_3—accept, else—reject.

Formally, we can specify the description given above by a transition function:

$(state, symbol)$	$\longmapsto$	$(state, symbol, direction)$
$(START, B)$	$\longmapsto$	$(REJECT, B, -)$
$(START, 0)$	$\longmapsto$	$(q_0, 0, \rightarrow)$
$(START, 1)$	$\longmapsto$	$(q_1, 1, \rightarrow)$
(q_0, B)	$\longmapsto$	$(q_2, B, \leftarrow)$
$(q_0, 0)$	$\longmapsto$	$(q_0, 0, \rightarrow)$
$(q_0, 1)$	$\longmapsto$	$(q_0, 1, \rightarrow)$
(q_1, B)	$\longmapsto$	$(q_3, B, \leftarrow)$
$(q_1, 0)$	$\longmapsto$	$(q_1, 0, \rightarrow)$
$(q_1, 1)$	$\longmapsto$	$(q_1, 1, \rightarrow)$
(q_2, B)	$\longmapsto$	$(REJECT, B, -)$
$(q_2, 0)$	$\longmapsto$	$(ACCEPT, 0, -)$
$(q_2, 1)$	$\longmapsto$	$(REJECT, 1, -)$
(q_3, B)	$\longmapsto$	$(REJECT, B, -)$
$(q_3, 0)$	$\longmapsto$	$(REJECT, 0, -)$
$(q_3, 1)$	$\longmapsto$	$(ACCEPT, 1, -)$

$\square$

The example illustrates the power (and weaknesses) of the Turing machine. Using its states, the Turing machine can remember a finite amount of information, and as such is more powerful than certain other models of computation such as finite automata or pushdown automata. On the other hand, the Turing machine is an extremely weak and primitive programming language. It has a single primitive data structure—a string of symbols. The only operations available to the machine are moving right or left on the string, rewriting the symbol at the current head position, and branching depending on the symbol currently scanned. Despite its weak and clumsy appearance, it can be shown that the Turing machine is capable of expressing any algorithm and simulating any programming language, with only a small loss of efficiency.

Definition 2.4 Let Σ be some alphabet (that is some finite set of symbols). Formally, a *string* over Σ is a finite sequence of symbols of Σ. We denote by $|x|$ the length of string x and by ε the empty string, that is a string consisting of zero symbols. The *Kleene closure* Σ^* of Σ is the set of all finite strings over Σ. A *(formal) language* is any subset of Σ^*.

Even though Turing machines can be used to implement any algorithm, it should be clear by now that they are especially well-suited for solving certain specialized kinds of problems on strings, namely *computing string functions* and *accepting* and *deciding languages*. Let us now define these tasks precisely.

Definition 2.5 Let L be a language over Σ, and let $M = (Q, \Sigma, \Gamma, \delta, F)$ be a Turing machine such that for any $x \in L$, $M(x) =$ "*yes*", and for any $x \in \Sigma^* \setminus L$, $M(x) =$ "*no*". Then we say that M *decides* L.

Similarly, we say that M *accepts* L if for any $x \in L$, $M(x) =$ "*yes*", and, for any $x \notin L$, either $M(x) =$ "*no*" or $M(x) = \nearrow$.

If a language L is decided by some Turing machine, then L is called *recursive*. If a language L is accepted by some Turing machine, then L is called *recursively enumerable*.

Notice that the concept of acceptance is not a true algorithmic concept, only a useful way of categorizing problems. This is because we can never tell whether the Turing machine is just computing for a long time but will halt eventually, or it will never halt.

We will also need Turing machines to compute functions on strings.

Definition 2.6 Suppose that f is function from Σ^* to Σ^* and let M be a Turing machine with input alphabet Σ. We say that M *computes* f if, for any string $x \in \Sigma^*$, $M(x) = f(x)$. If such an M exists, f is called a *recursive function*.

It can be shown that the set of recursive languages is a proper subset of the set of recursively enumerable languages. There are many other interesting relations between the above defined concepts. Further details can be found in [37] and [74].

Note: There is an entire area of mathematics called the *theory of recursive functions* that deals with recursive and recursively enumerable languages, recursive functions, and related concepts. The interested reader can find more about it in [81].

Let us now turn our attention to formal definitions of the time and space needed by Turing machines for their computation. In order to do so, we need to define a k-tape Turing machine, $k \geq 1$. Such machine has k tapes, the first tape usually holds the input and the last tape holds the output. The machine has one finite control but k tape heads, one for each tape. One move of the k-tape Turing machine now depends on the current state of the machine and on all the symbols scanned by the k tape heads. In one move, the Turing machine changes its state, writes a new symbol on each of the k tapes, and moves each tape head to the left or to the right or keeps it at the same position. Thus the transition function is now a function from $Q \times \Gamma^k$ to $Q \times \Gamma^k \times D^k$. Other aspects of the k-tape Turing machine remain the same as for the 1-tape Turing machine.

It can be shown that a k-tape Turing machine can be simulated by a 1-tape Turing machine with at most quadratic slow-down and by a 2-tape Turing machine with at most linear slow-down. To make this more precise, if M is a k-tape Turing machine that needs $h(x)$ steps to output $M(x)$, then there exists a 1-tape Turing machine M' and 2-tape Turing machine M'' that for every $x \in \Sigma^*$ output $M(x)$ in $O((h(x))^2)$ and $O(h(x))$ steps respectively.

It is convenient to use k-tape Turing machines to define the concept of *time*.

Definition 2.7 Let $f : \mathbf{N} \longrightarrow \mathbf{N}$. We say that a Turing machine operates within time $f(n)$ if for every input $x \in \Sigma^*$ the number of steps needed to reach a final state is at most $f(|x|)$.

We define the *complexity class* $DTIME(f(n))$ as the set of all languages $L \subseteq \Sigma^*$ that are *decided* by Turing machines with multiple tapes operating within the time bound $f(n)$.

It is reasonable to assume that $f(n) \geq n + 1$ since the Turing machine should be at least allowed to read its input and that takes $n + 1$ steps. For simplicity, we will assume that $f(n) = \omega(n)$; that is, $\lim f(n)/n = \infty$. For such running times, a result known as the *linear speed-up theorem* [49, pp.289–291] shows that any k-tape Turing machine, $k > 1$, that runs in time $O(f(n))$ can be simulated by a k-tape Turing machine that runs within time bound $f(n)$. Thus we say that $L \in DTIME(f(n))$ if there exists a multiple-tape Turing machine M that decides L and runs in time $O(f(n))$. The above definition gives us an entire hierarchy of deterministic time-complexity classes, the most famous among them being the class $\mathrm{P} = \bigcup_{k \geq 1} DTIME(n^k)$. Thus P is the class of all languages that can be decided by some Turing machine within a polynomial time bound. Another important time-complexity class is $\mathrm{EXP} = \bigcup_{k \geq 1} DTIME(2^{n^k})$.

Similarly, one can define space used by a Turing machine as the total number of cells required for computation. Here, however, one has to be a little more careful since we do not want to count the cells used solely for input or solely for output. This leads to the following definition.

Definition 2.8 Let $f : \mathbf{N} \longrightarrow \mathbf{N}$ and let M be a k-tape Turing machine, $k \geq 2$ with a read-only input tape and write-only output tape. We say that Turing machine M operates within space $f(n)$ if for every input $x \in \Sigma^*$ the total number of cells used by the Turing machine on its remaining $k - 2$ tapes (other than the input and output tapes) is at most $f(|x|)$.

We define the *complexity class* $DSPACE(f(n))$ as the set of all languages $L \subseteq \Sigma^*$ that are *decided* by Turing machines with multiple tapes with read-only input tape and write-only output tape operating within the space bound $O(f(n))$.

Note: Similarly as in the case of time-complexity classes, a precise definition of space-complexity classes does not use the O-notation. However, the result known as the *linear space compression theorem*[49, pp.288–289] shows that under the assumption that $f(n) = \omega(1)$, any k-tape Turing machine with read-only input tape and write-only output tape that runs in space $O(f(n))$ can be simulated by a k-tape Turing machine that runs in space $f(n)$.

There are two important space-complexity classes: PSPACE $= \bigcup_{k\geq 1} DSPACE(n^k)$, that is the class of all languages decidable by Turing machines operating within polynomial space; and L $= DSPACE(\log n)$, that is the class of all languages decidable by Turing machines that operate within logarithmic space.

2.2.2 Nondeterminism

It can be shown that the Turing machine model of computation can simulate other seemingly more powerful models of computation like random access machines with only a polynomial (if any) loss of efficiency. And, conversely, these models can simulate a k-tape Turing machine with only a polynomial loss of efficiency. Thus despite its clumsy appearance, the Turing machine model is both a powerful and realistic model of computation. In this subsection we introduce an unrealistic model of computation—the nondeterministic Turing machine. It can be simulated by the deterministic Turing machine with an *exponential* loss of efficiency. It remains an open question whether this exponential loss of efficiency is inherently associated with nondeterminism—this in fact is the famous P=NP problem (see below for more on this).

A nondeterministic Turing machine is very much like an ordinary (deterministic) Turing machine. The only difference is that now the transition function δ is no longer a function, but rather a multi-valued function or relation. Thus given a state of a k-tape nondeterministic Turing machine and a k-tuple of symbols currently scanned by the tape heads, one computation of the Turing machine can at each step choose among several possible moves. Thus the computation of a nondeterministic Turing machine can be viewed as a tree of different computation paths, rather than a single path as was the case of a deterministic TM.

Formally, we can define a nondeterministic k-tape Turing machine as follows.

Definition 2.9 A nondeterministic k-tape Turing machine is a quintuple $M = (Q, \Sigma, \Gamma, \Delta, F)$, where Q is the finite set of states with 'START' $\in Q$, $F \subseteq Q$ is the set of final states, Σ is the set of input symbols not containing the blank symbol B, Γ is the set of tape symbols

($\Sigma \subseteq \Gamma$ and $B \in \Gamma$), and Δ is the transition relation, $\Delta \subseteq [(Q \setminus F) \times \Gamma^k] \times [Q \times \Gamma^k \times D^k]$, where $D = \{\leftarrow, \rightarrow, -\}$.

What makes nondeterministic machines so different and powerful are our very soft requirements on what it means for such a machine to "solve a problem."

Definition 2.10 We say that nondeterministic Turing machine N accepts a language $L \subseteq \Sigma^*$, if for any $x \in L$, there exists a computation path that ends in the 'ACCEPT' state.

Notice again the asymmetry in the concept of acceptance by a non-deterministic Turing machine. In order to say that $x \in L$, we need just *one* computation path that halts in the 'ACCEPT' state. However, to say that $x \notin L$, we have to make sure that *all* computation paths either halt in the 'REJECT' state or do not halt at all.

Definition 2.11 Let N be a nondeterministic k-tape Turing machine and let $f : \mathbf{N} \longrightarrow \mathbf{N}$. We say that N accepts language L in time $f(n)$, if N accepts L and for any $x \in \Sigma^*$ any computation path of N terminates within $f(|x|)$ steps. If in addition, N has a read-only input tape and write-only output tape, we say that N accepts language L within space $f(n)$, if N accepts L and for any $x \in \Sigma^*$ any computation path of N uses at most $f(|x|)$ tape cells on the working tapes.

Thus the time we charge to N is really the number of steps in the longest computation path, i.e. we charge to N the depth of the computation tree. Clearly the total time needed by all computation paths can be exponentially bigger—see Figure 2.2.

Now we are ready to define the nondeterministic time and space complexity classes. Because of the linear speed-up and linear space compression theorems, we will use directly the O-notation.

Definition 2.12 Let $f : \mathbf{N} \longrightarrow \mathbf{N}$. The complexity class $NTIME(f(n))$ is the set of all languages $L \subseteq \Sigma^*$ that are accepted by nondeterministic Turing machine with multiple tapes operating within the time bound $O(f(n))$.

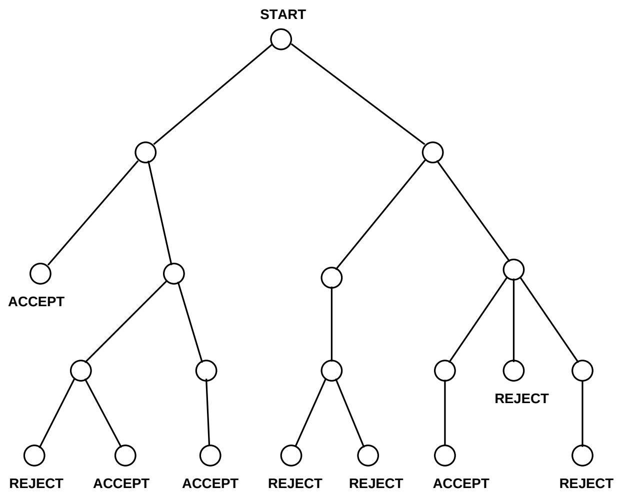


Figure 2.2: Computation tree of a nondeterministic Turing machine—nodes correspond to states of the Turing machine, edges correspond to legal moves

The complexity class $NSPACE(f(n))$ is the set of all languages $L \subseteq \Sigma^*$ that are accepted by nondeterministic Turing machines with multiple tapes with read-only input tape and write-only output tape operating within the space bound $O(f(n))$.

The important nondeterministic complexity classes are NP $= \bigcup_{k\geq 1} NTIME(n^k)$ and NL $= NSPACE(\log n)$. It turns out (by Savitch's theorem) that the class NPSPACE $= \bigcup_{k\geq 1} NSPACE(n^k)$ coincides with the class PSPACE.

As mentioned above, nondeterministic Turing machines can be simulated by deterministic Turing machines with an exponential slow-down; whether we can do better than that remains an important open question.

Theorem 2.1 *Let N be a nondeterministic k-tape Turing machine that accepts language L within the time bound $f(n)$. Then there exists a 3-tape deterministic Turing machine M with read-only input tape and write-only output tape that decides L in time $O(c^{f(n)})$, where $c \geq 2$ is some constant depending on N.*

Proof: Essentially, the machine M has to traverse all the paths of the computation tree of N. We first construct machine M' as follows.

Define d to be a degree of nondeterminism of N; that is, d is an upper bound on the number of choices N can make at each step. Each computation of N is a sequence of t nondeterministic choices, i.e. essentially a sequence of t integers in the range $0, 1, \ldots, d-1$. The simulating deterministic machine M' considers all such sequences of choices, in order of increasing length, and simulates N for each sequence. The sequence of choices is maintained on an extra tape and can be represented as an integer r in d-ary notation using d extra symbols $s_0, s_1, \ldots, s_{d-1}$, different from the tape alphabet of N. By adding 1 to r, we simply get a new sequence of choices, possibly extending the length of the sequence.

Several things can happen while simulating one computation path of N on input x for some given sequence r of t choices.

1. While simulating a computation path of N, we reach state 'ACCEPT' of N. Then M' accepts x.

2. At some step of the simulation, we need to make a choice but we have already made t choices. In such case we halt the current computation, and start a new computation with r now being a sequence of $t+1$ symbols s_0, that is we now consider the first sequence $\langle 0, 0, \ldots, 0 \rangle$ of $t+1$ choices, possibly skipping some sequences of length t.

3. (a) While simulating a computation path of N, we reach state 'REJECT', or

   (b) at some step of the simulation, we are supposed to make an unavailable choice, that is a choice α $(1 \leq \alpha \leq d)$ but the number of choices at this step is less than α.

   In either case, we simply start a new computation with sequence $r+1$ of length t. If length of $r+1$ is $t+1$, that is we have checked all the sequences of length t, with each corresponding computation halting in a 'REJECT' state or trying to make an unavailable choice, then M' rejects x.

It is clear from the description above that M' decides L, since we have to check all the sequences of choices of length at most $f(n)$ (most likely much less). Notice that here the concept of accepting a language is in fact equivalent to deciding a language since we have an (unknown) bound on the running time of N, that is no computation of N can go on forever and hence we can just wait for each computation to halt.

The total running time of M' can be bound from above in the following way: M' simulated a computation path of N at most $d^{f(n)}$-times. Each time we performed at most $f(n)$ steps simulating N and another $O(f(n))$ steps were needed to add 1 to the sequence of choices. Thus we need total time of $O(f(n)\, d^{f(n)})$ to simulate N using $(k+1)$-tape Turing machine M'. Using the tape-reduction theorem, one can simulate M' by 3-tape Turing machine M with read-only input tape and write-only output tape with at most a quadratic slowdown, thus the running time of M is $O(f^2(n)\, d^{2f(n)}) = O\left((d^2+1)^{f(n)}\right)$. □

In terms of time-complexity classes, we can restate the above theorem simply as

$$NTIME(f(n)) \subseteq \bigcup_{c \geq 2} DTIME(c^{f(n)}).$$

2.2.3 Hierarchy Theorems

We have defined four different types of complexity classes—DTIME, NTIME, DSPACE, and NSPACE. There are other ways to classify problems into different groups, most notably the randomized time and space complexity classes. There are many interesting relationships among the complexity classes—classes are subsets of some other classes, proper subsets of some other classes, classes have empty (or non-empty) intersections, etc. Many of these relationships are trivial; many of these relationships are non-trivial but known; and surprisingly, most of these relationships are either conjectured or totally uncertain. Thus one can very often see theorems in the form of implications, something like: "If these two classes are equal, then these and these classes are also equal." Researchers in complexity theory try to establish these relations but many (very hard) questions still remain open. The most famous open question in complexity theory is whether P = NP.

We will present here just a few basic hierarchy results so that the reader can get some limited familiarity with the area. For more on relationships between complexity classes see [74].

Theorem 2.2 *Let $f : \mathbf{N} \longrightarrow \mathbf{N}$ be a proper complexity function (any reasonable non-decreasing function satisfies this requirement). Then*

1. *$DSPACE(f(n)) \subseteq NSPACE(f(n))$ and $DTIME(f(n)) \subseteq NTIME(f(n))$.*

2. *$NTIME(f(n)) \subseteq DSPACE(f(n))$.*

3. *$NSPACE(f(n)) \subseteq \bigcup_{c \geq 2} DTIME(c^{\log n + f(n)})$.*

4. *P is a proper subset of EXP.*

5. *If $f(n) \geq \log n$, then $NSPACE(f(n)) \subseteq DSPACE(f^2(n))$ (Savitch's Theorem).*

2.2.4 Decision Problems and Encoding Schemes

In this subsection we consider decision problems and their relationship to languages. A decision problem is a problem that has only two possible solutions—either the answer is "yes" or the answer is "no". We will specify the decision problem in two parts—the first part describes a *generic instance* of the problem in terms of various components, which can be sets, graphs, functions, numbers, etc.; the second part states a yes-no question asked in terms of the generic instance. If the answer to the question for a particular instance I is "yes", we say that I is a *yes-instance* of the decision problem.

Let us now present several well-known decision problems.

REACHABILITY

Instance: A graph $G = (V, E)$ and two nodes $u, v \in V$.

Question: Is there a path from u to v?

Definition 2.13 Let $U = \{u_1, u_2, \ldots, u_n\}$ be a set of boolean variables. A *truth assignment* for U is a function $t : U \longrightarrow \{T, F\}$. We say that $u \in U$ is true under t, if $t(u) = T$. If $t(u) = F$, we say that u is false. If $u \in U$, then u and $\bar{u}$ are literals over U. The literal u is true under t if and only if the variable u is true under t; the literal $\bar{u}$ is true if and only if the variable u is false.

A *clause* over U is a set of literals over U. It represents the disjunction of those literals. We say that a clause is *satisfied* by a truth assignment if at least one of its members is true under that assignment. Otherwise we say that the clause is not satisfied. For example the clause $\{u_1, \bar{u}_5\}$ is satisfied by exactly those truth assignments t for which $t(u_1) = T$ or $t(u_5) = F$. A collection $\mathbf{C}$ of clauses over U is *satisfiable* if there exists some truth assignment for U that simultaneously satisfies *all* the clauses in $\mathbf{C}$. We call such a truth assignment a *satisfying truth assignment* for $\mathbf{C}$.

Satisfiability (SAT)

Instance: A finite set U of boolean variables and a collection $\mathbf{C}$ of clauses over U.

Question: Is there a satisfying truth assignment for $\mathbf{C}$?

Clique—decision version

Instance: An undirected graph $G = (V, E)$ and number K, $1 \leq K \leq |V|$.

Question: Does G contain a clique of size at least K?

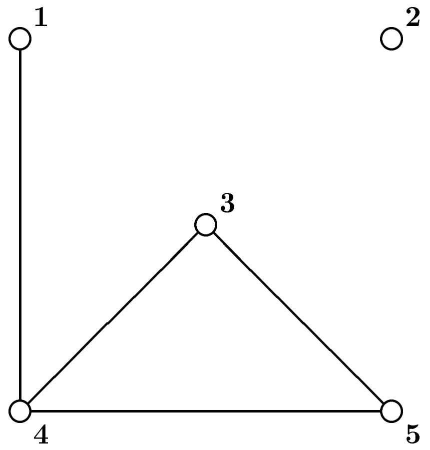
Hamiltonian Cycle (HC)

Instance: An undirected graph $G = (V, E)$.

Question: Does G have a Hamiltonian cycle?

We would like to use Turing machines to decide these problems. In order to do so, we first have to represent a particular instance of a problem as a string of symbols using some *encoding scheme*. It is easy to see that any "finite" mathematical object can be represented by a finite string over an appropriate alphabet. In fact there are usually several different encoding schemes that lead to several different encodings of the same object.

For example, a particular undirected graph can be represented by an *adjacency list* over the alphabet $\Sigma = \{0, 1, \ldots, 9, (,), ;\}$, where one labels the vertices of the graph by integers and simply lists all the vertices, each with its adjacent neighbors. Using this *encoding scheme*, the graph in Figure 2.3 would be represented as "$(1;4)(2)(3;4;5)(4;1;3;5)(5;3;4)$",



(1;4)(2)(3;4;5)(4;1;3;5)(5;3;4)

00010;00000;00011;10101;00110

Figure 2.3: Encodings of a graph

which is a string of length 31. If instead we represent the integers in binary, our alphabet would be $\Sigma = \{0, 1, (,), ;\}$, and the graph in Figure 2.3 would be encoded as "$(1; 100)(10)(11; 100; 101)(100; 1; 11; 101)(101; 11; 100)$", which is a string of length 49. Similarly, one could use an *adjacency matrix* instead, namely a matrix of 0's and 1's such that the entry in row i and column j is 1 iff there is an edge $(i, j) \in E$, and separate the rows of the matrix by ';' for example. In such case, our encoding alphabet would be $\Sigma = \{0, 1, ;\}$. Using this encoding scheme, the graph from Figure 2.3 would be encoded as "$00010; 00000; 00011; 10101; 00110$", that is a string of length 29. A weighted graph can be represented using a different type of adjacency matrix where 1's for existing edges are simply replaced by the edge weights and 0's for non-existing edges are replaced by some different symbol, say ∞. The entries in one row of the adjacency matrix could be separated by ';', rows of the matrix by ';;'. In such case, our encoding alphabet would be $\Sigma = \{0, 1, ;, \infty\}$ if we represent integer weights in binary.

The encodings above are all "reasonable" encodings in the following sense:

1. the encoding of an instance I is concise and not padded with unnecessary information or symbols, and

2. numbers occurring in I are represented in binary (or decimal, or in any fixed base other than 1).

It turns out that the lengths of representations of the same object using different "reasonable" encoding schemes differ at most polynomially from each other. Thus if there is a Turing machine that solves a given problem in polynomial time under one encoding scheme, then there is also a Turing machine that solves the problem in polynomial time under a different encoding scheme.

Note: An example of an *unreasonable encoding* would be a so-called unary encoding, where an integer a is represented by a string of a 1's. Under such encoding, an algorithm that runs in time, say, $O(a)$ would have running time linear in the size of input whereas under a binary encoding, where a is represented by a string of length $\lfloor \log_2 a \rfloor + 1$, the running time would be exponential in the size of input since $O(a) = O(2^{\log_2 a})$. Thus unnecessary padding of encoding strings might result in a seemingly favorable performance.

For a given decision problem Π, let $\mathbf{e}$ be the encoding scheme and let Σ be the encoding alphabet. Consider now the following language:

$$\begin{aligned} L[\Pi, \mathbf{e}] = \{x \in \Sigma^* \quad | \quad & x \text{ is an encoding of some instance } I \text{ of } \Pi \\ & \text{using encoding scheme } \mathbf{e}, \text{ and } I \text{ is a yes-instance of } \Pi\} \end{aligned}$$

Thus I is a yes-instance of problem Π if and only if its corresponding encoding x belongs to $L[\Pi, \mathbf{e}]$, and conversely, members of $L[\Pi, \mathbf{e}]$ are exactly those strings in Σ^* that are encodings of some meaningful yes-instances of Π. Notice that the encoding scheme $\mathbf{e}$ partitions Σ^* into three classes of strings: those that are not encodings of instances of Π, those that are encodings of no-instances of Π, and finally those that are encodings of yes-instances of Π.

The discussion above shows that there is a correspondence between decision problems and languages via encoding schemes. Formally, we say that if a result holds for a language $L[\Pi, \mathbf{e}]$, then it holds for the problem Π under the encoding scheme $\mathbf{e}$. Usually, it is the case that a property of the language $L[\Pi, \mathbf{e}]$ is encoding-independent as long as we use a "reasonable" encoding. That is, given any two encodings $\mathbf{e}$, $\mathbf{e}'$ of problem Π, either both $L[\Pi, \mathbf{e}]$ and $L[\Pi, \mathbf{e}']$ have the property or neither of them does. In such a case we can informally say that the property holds (does not hold) for problem Π, without specifying a particular encoding scheme. However, it is implicit in such a statement that we could, if necessary, specify a particular reasonable encoding scheme $\mathbf{e}$ such that the property holds (does not hold) for $L[\Pi, \mathbf{e}]$.

An example of an encoding-independent property is membership of a problem in some complexity class. Thus to show that some problem Π belongs for example to P, one could choose some reasonable encoding and then construct a Turing machine that decides the corresponding language within polynomial time. However, we are usually less formal. In order to show that $\Pi \in$ P, it is enough to describe an algorithm (that is give a high-level description of a Turing machine) that solves a particular instance. We then argue that this algorithm, if implemented using a Turing machine (or some other equivalent model of computation), would run in time polynomial in the size of the instance.

Example 2.2 REACHABILITY $\in$ P.

Solution: Consider the following algorithm: Given an instance of REACHABILITY, that is a graph $G = (V, E)$ and two distinct vertices u and v, we color vertex u black and all its neighbors grey. Then we repeatedly pick a grey vertex, color it black, and color grey all its not yet colored neighbors. The algorithm stops when there are no more grey vertices and outputs "*yes*" if vertex v is black and "*no*" otherwise. Clearly this algorithm solves the REACHABILITY problem. Using a 3-tape Turing machine, it could be easily implemented to run in time $O((|V| + |E|)^2 \log |V|)$, using the second tape to keep track of grey vertices and third tape to keep track of black vertices. And since the size of any instance of REACHABILITY is $O((|V| + |E|) \log |V|)$ using, for example, the adjacency list representation, our algorithm runs in time polynomial in the size of the input. □

Similarly as in the case of membership in P, to show that problem Π belongs to NP, one usually does not construct a Turing machine and use a particular encoding scheme. Rather we give a high-level description of an (encoding-independent) nondeterministic algorithm that consists of two stages—the guessing stage in which we guess some structure, and the verifying stage in which we verify that this structure is a yes-instance of the given problem. We then argue that the length of the guess depends polynomially on the size of the instance and that the verifying stage of the algorithm, if implemented using a Turing machine (or some other equivalent model of computation), would run in time polynomial in the size of the instance.

Example 2.3 HAMILTONIAN CYCLE $\in$ NP.

Solution: Consider the following nondeterministic algorithm: Given an instance of HC, that is a graph $G = (V, E)$ with $|V| = n$, guess a sequence $\langle v_1, v_2, \ldots, v_n \rangle$ of not necessarily distinct vertices. Then check whether the guessed vertices are all distinct, whether $(v_i, v_{i+1}) \in E$ for each $i = 1, \ldots, n-1$, and whether $(v_n, v_1) \in E$. If all these conditions are satisfied, then the guessed sequence of vertices is indeed a Hamiltonian cycle, and hence the algorithm outputs "*yes*". If any of the conditions is violated, the algorithm outputs "*no*". Clearly, the above is a high-level description of a nondeterministic Turing machine that accepts HC under any reasonable encoding scheme. It is easy to check that the length of the guessed sequence and running time of the algorithm depend polynomially on the size of the input instance—using for example the adjacency matrix encoding scheme, the size of the instance is $O(n^2)$, the size of the guessed sequence is $O(n \log n)$, and the algorithm could be implemented using a 2-tape Turing machine to run in time $O(n^3 \log n)$. □

2.2.5 Reductions and NP-completeness

Definition 2.14 Given two languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, we say that L_1 is *(polynomial-time) reducible* to L_2, if there exists a function $f : \Sigma_1^* \longrightarrow \Sigma_2^*$ that can be computed by some Turing machine in polynomial time, such that

$$x \in L_1 \Longleftrightarrow f(x) \in L_2.$$

In such a case, we call f a *(polynomial-time) reduction.*

Given two problems Π and Λ we say that Π is *(polynomial-time) reducible* to Λ if there exist reasonable encoding schemes $\mathbf{e}_\Pi$ and $\mathbf{e}_\Lambda$ such that $L[\Pi, \mathbf{e}_\Pi]$ is (polynomial-time) reducible to $L[\Lambda, \mathbf{e}_\Lambda]$.

Note: The function f is often called a *polynomial-time many-one reduction* or *Karp reduction.*

Note: The concept of reduction and the entire theory of NP-completeness stems from similar concepts in the theory of recursive functions—see [81] for details. There, however, one does not usually worry about the length of computation.

The following proposition is an easy consequence of the fact that a composition of two polynomials is a polynomial.

Proposition 2.1 *If language L_1 is reducible to L_2 and L_2 is reducible to L_3 then L_1 is reducible to L_3. Thus reducibility is transitive.*

The concept of reduction will allow us to compare problems (languages) based on their difficulty. Consider two languages L_1 and L_2, such that L_1 reduces to L_2. Let M_2 be a Turing machine that decides L_2, and let M be a Turing machine that computes the reduction f. We can construct a new Turing machine M_1 that works as follows—given an input string $x \in \Sigma_1^*$, M_1 first simulates M to get the string $f(x) \in \Sigma_2^*$, then simulates M_2 on $f(x)$. If $M_2((f(x)) = $ "*yes*", M_1 accepts, if $M_2((f(x)) = $ "*no*", M_1 rejects. Clearly $M_1(x) = $ "*yes*" if and only if $x \in L_1$, thus M_1 solves (decides) L_1. Therefore the fact that L_1 is reducible to L_2 allows us to claim that the language L_2 is at least as hard to decide as the language L_1.

Let p be a polynomial such that M runs within time $p(n)$, then clearly $|f(x)| \leq p(|x|)$ for any $x \in \Sigma_1$. If M_2 runs within time $p_2(n)$ for some polynomial p_2, then, given $x \in \Sigma_1$, M_1 runs within time $O(p(|x|) + p_2(|f(x)|)) = O(p(|x|) + p_2(p(|x|))) = p_1(|x|)$ for some polynomial p_1. Therefore if M_2 runs within polynomial time, so does M_1.

When we consider reducibility between problems, we are usually a bit informal and define the reduction in terms of instances of the problems rather than in terms of particular encodings of instances. It is however implicit in such a construction that there are reasonable encodings of both problems under which the reduction is computable by some Turing machine (or some other equivalent model of computation) in time polynomial in the length of the input.

Now we are ready to define the most important concept in the theory of computation.

Definition 2.15 We say that a decision problem Π is *NP-hard* if every problem in NP is reducible to Π. If in addition problem Π itself belongs to NP, Π is said to be *NP-complete*.

Thus NP-complete problems are the "hardest" of all problems in NP. Moreover, the above discussion implies that if one found a polynomial-time algorithm for solving one NP-complete problem, we would have a polynomial time algorithm for *all* problems in NP, that is, we would show that NP=P. So far, nobody has been able to show either P=NP or P$\neq$NP and this question remains the most famous open problem in theoretical computer science. Nevertheless, all the circumstantial evidence suggests that in fact P$\neq$NP. This justifies the current interest of many researchers in the analysis and design of polynomial-time algorithms that *approximate* the solutions of NP-hard optimization problems—see Section 2.5 for details.

It is not clear from the above discussion that NP-complete problems even exist. That is the main reason why the following theorem is so important.

Theorem 2.3 (Cook, 1971) SATISFIABILITY *is NP-complete.*

Proof: It is easy to see that SAT is in NP. A nondeterministic algorithm for it would simply guess a truth assignment and then check whether all clauses are satisfied.

To see that all languages in NP are reducible to SAT is quite involved. Here we only outline the main idea of the proof as presented in [37].

To show that language $L \in$ NP reduces to SAT, we work with the corresponding nondeterministic 1-tape Turing machine N that accepts L within polynomial time $p(n)$.

The goal here is to construct a polynomial-time transformation f_L from input strings of N to instances of SAT in such a way that input string x is accepted by N if and only if the corresponding instance $I = f_L(x)$ of SAT has a satisfying truth assignment. Given x, this is done by introducing several types of boolean variables to be contained in the set U and then creating a collection of clauses over U that is satisfiable iff there is an accepting computation of N on x.

If we index the finitely-many states of N by k, the polynomially-many time steps by i, the polynomially-many tape cells used by any computation of N by j, and the finitely-many tape symbols by l, we have the following variables:

$Q[i, k]$: True iff at time i machine N is at state q_k;

$H[i, j]$: True iff at time i the head is scanning cell j;

$S[i, j, l]$: True iff at time i cell j contains the l-th symbol.

We then construct a collection **C** of clauses that correspond to some meaningful requirements on the accepting computation of a nondeterministic Turing machine. Thus **C** contains clauses that assure that, at each time, N is in exactly one state, that, at each time, the tape head is scanning exactly one tape cell, that, at each time, each cell contains exactly one tape symbol, that, at time 0, the computation is in the starting state 'START' with head scanning the first tape cell, that, by the time $p(|x|)$, N has entered 'ACCEPT' state, and finally that the moves of N are all legal according to the transition relation Δ.

If there is an accepting computation of N on x, then this computation defines a truth assignment for variables in U such that **C** is satisfied. If there is no accepting computation of N on x, then there is no truth assignment that would satisfy all clauses in **C**. It is now relatively straightforward to make sure that the above described reduction f_L could be computed in polynomial time. Thus f_L is a reduction of L to SAT. And since L was a generic language in NP, SAT is NP-complete. □

Once we have proved that one problem is NP-complete, the procedure for proving that other problems are NP-complete is fortunately much simpler than the proof of Cook's

Theorem. By transitivity of reductions, in order to show that problem Π is NP-complete, it is enough to show that Π is in NP and that SAT or some other NP-complete problem is reducible to Π. Thus the more NP-complete problems we have, the easier it is to show that other problems are NP-complete. It should be noted, however, that some of the reductions can be quite "tricky" and complicated—see [37] for several examples. Using this technique, Karp in [56] identified additional 20 NP-complete problems. We will illustrate this concept by showing that CLIQUE$_D$, the decision version of the CLIQUE problem, is NP-complete by establishing a reduction from SAT to CLIQUE$_D$.

Proposition 2.2 *The decision version of* CLIQUE *is NP-complete.*

Proof: It is easy to see that CLIQUE$_D \in$ NP since a nondeterministic algorithm, given a graph $G = (V, E)$ and a number K, only needs to guess K vertices and check whether they form a complete subgraph of G.

To see that CLIQUE$_D$ is NP-complete, we show that SAT is reducible to CLIQUE$_D$. Given an instance I of SAT, that is a collection $\mathbf{C} = \{C_1, \ldots, C_n\}$ of clauses over some finite set U of boolean variables, we construct graph $G = (V, E)$ as follows.

The elements of V will be pairs $[\sigma, i]$ where σ is a literal over U and $\sigma \in C_i$. Thus $|V| = |C_1| + \cdots + |C_n|$.

The elements of E are exactly those pairs $[\sigma, i]$, $[\delta, j]$ of vertices that come from different clauses, i.e. with $i \neq j$, and such that $\sigma \neq \bar{\delta}$. Thus an edge corresponds to a pair of literals from two different clauses that can be simultaneously satisfied by some truth assignment.

Finally, we set $K = n$. This completes the construction of an instance J of CLIQUE$_D$.

It is straightforward to see that this construction can be accomplished within polynomial time using some reasonable encoding schemes for both problems.

Thus it remains to show that I is a yes-instance of SAT iff J is a yes-instance of CLIQUE$_D$. Let t be a satisfying assignment for $\mathbf{C}$. Then there is a least one literal in each clause that is true. Let us select exactly one true literal from each clause and let V' be the set of the corresponding vertices. Since all the literals are from different clauses and all are satisfied simultaneously, all vertices are pair-wise adjacent and hence the subgraph G'

generated by V' is a complete subgraph of G of size $K = n$, that is J is a yes-instance of the CLIQUE_D problem.

Conversely, if J is a yes-instance of the CLIQUE_D problem, then there is a complete subgraph $G' = (V', E')$ of G, such that $|V'| \geq n$. However there are edges only between vertices corresponding to literals from different clauses, hence $|V'| = n$ and each vertex corresponds to one literal in each clause. Since all the vertices are adjacent, the corresponding literals can all be simultaneously satisfied by some truth assignment. This assignment satisfies all the clauses, hence I is is a yes-instance of SAT. $\square$

Similarly, using a reduction from CLIQUE_D to the VERTEX COVER problem, and a reduction from VERTEX COVER to HAMILTONIAN CYCLE, one can show that the HAMILTONIAN CYCLE problem is also NP-complete—see [37] for details.

Currently, hundreds of problems have been shown to be NP-complete. The interested reader can find most of the "basic" NP-complete problems in [37].

2.3 The Hypothetical Computer

In the previous section we saw that Turing machines are a very precise and convenient model for formally defining complexity classes and other concepts in computation complexity. However, it should be clear by now that they are not so convenient for implementing complicated algorithms and establishing their running times. Consider, for example, the simple and straightforward algorithms from Examples 2.2 and 2.3. These algorithms could be easily implemented on a real-world computer using a programming language like C to run in times $O(|V| + |E|)$ and $O(|V|^2)$ respectively. However, on a Turing machine, the running times become longer due to the time lost by locating information on the tapes of the Turing machine, that is by not being able to access all data directly, and by computations that depend on the size of the operands (like comparing two integers). Thus one would like to use a model of computation that would resemble more closely today's conventional computers and at the same time be polynomially equivalent in its power to the Turing machine model of computation. Indeed, such models exist; in the rest of this thesis, whenever we give a specific bound on the running time of some algorithm, we will have in mind one such model—we call it a *hypothetical computer*.

Definition 2.16 The *hypothetical computer* is a model of computation with the following properties:

- The computer has an unlimited random access memory. The memory stores logical constants, integers, and rational numbers in words of any required size. The access time is unaffected by the size of the words and by the number of words stored. For simplicity we assume that the word size depends polynomially on the size of the input—typically the word size can be considered independent of the problem size.
- The computer can perform all the standard instructions such as arithmetic operations, comparisons, branching, etc. expected from a conventional computer. We assume that each executed instruction requires one unit of time independent of the size of operands.

This hypothetical computer (known also as the unit-cost RAM) is certainly an idealization of a real-world computer. However, for the purposes of establishing the asymptotic running times of our algorithms, it is an adequate and convenient abstraction of existing computers.

It can be shown that the Turing machine model of computation can simulate the hypothetical computer with only a polynomial loss of efficiency (provided that the word size is bounded by some polynomial). Conversely, the hypothetical computer can simulate a k-tape Turing machine with no loss of efficiency. Thus the hypothetical computer is equivalent to other (standard) models of computation. Therefore an algorithm runs in polynomial time on the hypothetical computer if and only if it runs in polynomial time (perhaps polynomial of a different degree) on another standard model of computation.

The fact that all the standard models of computation are polynomially related shows that the theory of NP-completeness is independent from the model of computation used.

Note: There are (non-standard) models of computation that may be more powerful than the hypothetical computer or the Turing machine. An important example is the quantum computer. However, these models are only theoretical and (currently) cannot be realized in practice. Therefore this thesis and the classical combinatorial optimization are only concerned with the standard models of computation.

2.4 Combinatorial Optimization

Combinatorial optimization is a theory about solving *optimization problems*, namely problems where the goal is to find a "best solution" satisfying given requirements. A (feasible) *solution* is usually an arrangement or grouping of given objects that satisfies certain required properties, while a *best* solution is a solution that is "best" with respect to some given measure, that is, a solution that minimizes (or maximizes) the value of the measure over all possible (feasible) solutions. We will specify optimization problems in two parts—the first part describes a *generic instance* of the problem in terms of various components which can be sets, graphs, functions, numbers, etc.; the second part specifies the set of feasible solutions and the criteria on the "best" solution in the form of a goal.

Perhaps the best way how to convey the nature of combinatorial optimization is to give some specific examples of optimization problems. Many of these problems are abstractions of problems naturally arising in everyday applications.

Minimum Spanning Tree (MST)
Instance: An undirected edge-weighted graph $G = (V, E)$.
Goal: Find a spanning tree of minimum length.

Clique
Instance: An undirected graph $G = (V, E)$.
Goal: Find a clique of maximum size.

TRAVELING SALESMAN PROBLEM (TSP)

Instance: A complete edge-weighted graph $G = (V, E)$.

Goal: Find a Hamiltonian cycle of minimum length.

Note: In the context of TSP, a Hamiltonian cycle is sometimes referred to as a tour.

Definition 2.17 An *edge coloring* of a graph $G = (V, E)$ is a coloring of edges of G such that no two adjacent edges have the same color.

EDGE COLORING

Instance: An undirected graph $G = (V, E)$.

Goal: Find an edge coloring that uses a minimum number of colors.

All the above problems are associated with graphs. But there are many other optimization problems.

KNAPSACK PROBLEM

Instance: A capacity B of the knapsack and a set $U = \{1, 2, \ldots, n\}$ of items where each item i has a size s_i and a profit p_i associated with it.

Goal: Find a subset $U' \subseteq U$ of items such $\sum_{i \in U'} s_i \leq B$ (i.e. the items fit in the knapsack) and the total profit $\sum_{i \in U'} p_i$ is maximum.

Definition 2.18 Given a collection $\mathbf{S} = \{S_1, \ldots, S_n\}$ of subsets of some finite set U, we say that $\mathbf{S}$ is a *cover* of U if $\bigcup_{i=1}^{n} S_i = U$. We say that $\mathbf{S}^* = \{S_{i_1}, \ldots, S_{i_l}\}$ is a *subcover* of U if $\mathbf{S}^* \subseteq \mathbf{S}$ and $\mathbf{S}^*$ itself is a cover of U.

SET COVER

Instance: A finite set U, a cover $\mathbf{S} = \{S_1, \ldots, S_n\}$ of U.

Goal: Find a subcover $\mathbf{S}^* \subseteq \mathbf{S}$ of U of minimum cardinality.

The above problems are just a small sample of the field of combinatorial optimization. Even though some of them seem very similar, we will show that they are quite different in their nature.

2.4.1 Solving Optimization Problems

What does it mean to "solve" an optimization problem? For a mathematician, solving the above problems is trivial—each problem has only finitely many feasible solutions, so one can "check" all of them and choose the best one. This is, however, not a way one would like to use in practice. Consider, for example, the TRAVELING SALESMAN problem and imagine that we have a computer fast enough to check 10^9 feasible solutions in one second. Clearly, for an n-city TSP, there are $(n-1)!$ different tours, thus it would take almost 4 years to solve a 20-city problem, about 77 years to solve a 21-city problem, and over 16 hundred years to solve a 22-city problem. Thus solving even a small-size TSP by a complete enumeration is computationally infeasible and therefore this way of solving optimization problems is unacceptable.

The goal of combinatorial optimization is to develop *efficient* algorithms for solving optimization problems; that is, algorithms that need only a relatively small number of elementary computational steps. Formally, *an algorithm is considered efficient if the required number of elementary computational steps is bounded by a polynomial in the size of the input.* In such a case we say that the algorithm runs in time polynomial in the size of the input or simply that it runs in polynomial time. To establish running time of an algorithm we will use the hypothetical computer model from previous section, but the reader should keep in mind that whether an algorithm is efficient or not is independent of the computation model used and the specifics of "elementary computational steps".

In Section 2.2 we defined size of the input as the length of the string needed to encode a specific instance of the problem under some encoding scheme. We also argued that all reasonable encoding schemes are polynomially related hence a particular encoding scheme does not change whether an algorithm is efficient or not. In practice, we often simplify the notion of size of the problem. Consider, for example, the n-city TSP. Here the instance of the problem is fully specified by the lengths of each edge in the complete graph $G = (V, E)$. Thus the size of the problem is roughly αn^2, where $\alpha = \log_2 \max\{w_e \mid e \in E\}$. This is because we have $n(n-1)/2 = \Theta(n^2)$ edges, that is $\Theta(n^2)$ edge weights and each weight can be represented using at most α bits. However, instead of expressing the running time of an algorithm in the actual length of the input, we often express it more naturally as a function of n.

2.4.2 NP-completeness and Optimization Problems

Despite the fact that one can often see statements like 'TSP is NP-complete', the theory of NP-completeness as such does not apply to optimization problems. In order to be able to use the results of Section 2.2 we have to switch from optimization problems to their decision versions. That is, instead of asking for the optimal solution to, say, TSP, we can ask: *'Is there a tour of length at most K for some given K?'* Thus a statement like 'TSP is NP-complete' is very common (and harmless) abuse of terminology. What is meant by such statement is that the corresponding *decision version* of the optimization problem is NP-complete.

As an example, let us now state the decision version of TSP. The decision version of CLIQUE is given in Section 2.2, decision versions of other optimization problems can be obtained in a similar way.

TSP$_D$—THE DECISION VERSION OF TSP
Instance: A complete edge-weighted graph $G = (V, E)$, and a number K.
Question: Is there a Hamiltonian cycle of length at most K?

Let Π be an optimization problem, and let Π_D be the corresponding decision version of problem Π. If we had an algorithm for solving problem Π, that is, a Turing machine M that, given some (encoding of an) instance of Π, outputs (an encoding of) an optimum solution, we could use it to solve the problem Π_D by simply comparing the value of the optimum solution with the bound K. This "reduction" of problem Π_D to problem Π shows that if we can solve the optimization problem in polynomial time, then we can solve the decision problem in polynomial time for any given K.

The converse of the above statement is also true, assuming that the problem parameters are integers or rational numbers—a reasonable assumption for any optimization problem. To see this, consider, for example, the TSP problem with integer distances between cities and assume that we have a polynomial-time algorithm A for deciding problem TSP_D that each time either outputs "*no*" (if the lengths of all tours are larger than K) or outputs "*yes*" together with a "certificate" (a tour whose length is smaller than or equal to K). Consider now the following algorithm B:

1. Set M equal the sum of lengths of all edges of graph G. The length of an optimum tour is certainly between 0 and M. Set $a = 0$, $b = M$.

2. Call algorithm A with $K = (a+b)/2$. If the answer is "*yes*", we know that the length of the optimum tour is somewhere between a and $(a+b)/2$, hence set $b = (a+b)/2$. If the answer is "*no*", the length of the optimum tour is somewhere between $(a+b)/2$ and b, hence set $a = (a+b)/2$.

3. Repeat step 2 until obtaining an interval of length less than 1.

4. The "certificate" output by A with $K = b$ is then an optimum solution to TSP.

Clearly, B solves TSP and calls algorithm A at most $\log_2 M$ times. And since A is a polynomial-time algorithm, algorithm B also runs in polynomial time.

The above described reductions are cases of the so called *Turing reduction* used by some authors to define the concept of NP-hardness. A problem Π is said to be NP-hard if all problems in NP are Turing reducible to Π. Turing reduction is a generalization of Karp reduction and hence allows more problems to be classified as NP-hard. Roughly speaking, problem Π Turing reduces to problem Λ if there is a polynomial-time algorithm M that solves problem Π by calling (possibly several times) as its subroutine algorithm M_Λ that solves the problem Λ—the running time of M_Λ does not contribute to the running time of M. Clearly Karp reduction is a special case of Turing reduction where the subroutine is called only once at the end of the computation of M.

The fact that decision versions of optimization problems are trivially Turing reducible to the optimization problems justifies why optimization problems whose decision versions are NP-complete are said to be NP-hard.

Example 2.4 *The decision version of the* TRAVELING SALESMAN *problem is NP-complete; that is,* TSP *is NP-hard.*

Solution: Clearly TSP_D is in NP—a nondeterministic algorithm would simply guess a permutation of vertices and check if the total length of the edges on the tour is at most K.

To show that TSP_D is NP-complete, it is enough to show that some NP-complete problem reduces to TSP_D. We will show that HAMILTONIAN CYCLE reduces to TSP_D. To see this, consider an instance of HC, that is an undirected graph $G = (V, E)$ and construct an instance of TSP_D as follows: Let $G' = (V, E')$ be a complete graph on V. For each edge $e \in E'$ set $l_e = 1$ if $e \in E$, and $l_e = 2$ otherwise. Set $K = |V|$. Clearly, this construction can be accomplished in polynomial time and G has a Hamiltonian cycle if and only if there is a tour in G' of length $|V|$. $\square$

It should be noted that the list of NP-hard optimization problems is fairly large. Karp in his seminal work [56] described first 21 NP-complete decision problems (many of them being decision versions of optimization problems) and many others followed. The list of NP-hard optimization problems given by Garey and Johnson in [37] is periodically updated in the *Journal of Algorithms* and currently contains hundreds of problems.

2.5 Approximation Algorithms

With the exception of the MST problem, all the other problems defined in the previous section are NP-hard. Hence it is very unlikely that these problems could be solved by a polynomial-time algorithm. In fact, all known algorithms solving these and other NP-hard problems run in time exponential in the size of the input. However, these problems still have to be solved in practice. In order to do that, we have to relax some of the requirements on solving the problem. There are, in general, three different possibilities.

Superpolynomial-time algorithms: Even though an optimization problem is NP-hard, there are good and not so good algorithms for solving it exactly. Among the "not-so-good" algorithms certainly belong most of the simple enumeration methods where one would enumerate all the feasible solutions and then choose the one with the optimal value of the objective function. Such methods have very large time complexity. Among the "good" algorithms belong methods like "branch-and-bound" where an analysis of the problem in hand is used to discard most of the feasible solutions before they are even considered. These approaches allow one to obtain exact solutions of reasonably large problem instances, but their running time still depends exponentially on the size of the problem.

Average-case polynomial-time algorithms: For some problems, it is possible to have algorithms which require super-polynomial time on only a few (usually artificial) instances and in general run in polynomial time. A famous example is the "Simplex Method" for solving problems in Linear Programming.

Approximation Algorithms: We may also relax the requirement of obtaining an exact solution to the optimization problem and satisfy ourselves with a solution which is "not too far" from the optimum. This is partially justified by the fact that, in practice, it is usually enough to obtain a solution that is only slightly sub-optimal.

As the thesis title suggests, design and analysis of *approximation algorithms* will be our main concern here.

Clearly, there are good approximation algorithms and bad ones as well. What we need is some means of determining the quality of an approximation algorithm and a way of comparing different algorithms. There are a few criteria to consider:

Average-case performance: One has to consider some probability distribution on the set of all possible instances of a given problem. Based on this assumption, an expectation of the performance can then be found. Results of this kind strongly depend on the choice of the initial distribution and do not provide us with any information about the performance on a particular instance.

Experimental performance: This approach is based on running the algorithm on a few "typical" instances. It has been used mostly to compare performance of several approximation algorithms. The results of course depend on the choice of the "typical" instances and may vary from experiment to experiment. This approach does not provide us with any information about the performance of the algorithm on different instances.

Worst-case performance: This is usually done by establishing upper or lower bounds on the size of approximate solution in terms of the size of the optimum solution. In case of minimization problems, we try to establish upper bounds, in case of maximization problems, one wants to find lower bounds.

In this thesis, we establish upper bounds on the performance of several efficient algorithms that approximate solutions to NP-hard minimization problems.

The advantage of worst-case bounds on the performance of approximation algorithms is the fact that given *any* instance of the optimization problem, we are *guaranteed* that the size of the approximate solution stays within these bounds. It should also be noted that approximation algorithms usually output solutions much closer to the optimum than the worst-case bounds suggest. Thus it is of independent interest to see how tight the bounds on the performance of each algorithm are; that is, how bad the approximate solution can really get. This is usually done by providing examples of specific instances for which the approximate solution is very far from the optimum solution.

Establishing worst-case performance bounds for even simple algorithms often requires a very deep understanding of the problem at hand and the use of powerful theoretical results from areas like linear programming, combinatorics, graph theory, probability theory, etc. This is why combinatorial optimization often thrives on results from other areas of mathematics and theoretical computer science.

Let us now turn our attention to the specific approximation algorithms that will illustrate different ways of measuring their performance and show significant differences in the quality of approximation that can be achieved for different optimization problems. We will also present some results about hardness of approximation.

Notation: In what follows, given an instance I of an optimization problem, and some algorithm for its approximation, we will denote by $OPT = OPT(I)$ the value of an optimum solution and by $APP = APP(I)$ the value of the solution output by the approximation algorithm.

2.5.1 Absolute Performance Guarantee

Assuming $P \neq NP$, solving an NP-hard optimization problem exactly is in general computationally infeasible. The next best thing one might try is to obtain an approximate solution whose value differs from the optimum solution by a small constant. This is formalized in the following definition.

Definition 2.19 An approximation algorithm for problem Π has an *absolute performance guarantee* of k, for some constant $k > 0$, if for every instance I of Π

$$|APP(I) - OPT(I)| \leq k.$$

It turns out that only a few problems can be approximated with an absolute performance guarantee. One of them is the EDGE COLORING problem. Let us first give a few preliminary results.

Definition 2.20 Given a graph $G = (V, E)$, a *degree* $d(v)$ of vertex $v \in V$ is the number of edges incident on v. Given an integer $k > 0$, we say that G is *k-regular* if $d(v) = k$ for each $v \in V$.

Theorem 2.4 (Vizing's Theorem) *Let $G = (V, E)$ be an undirected graph, and let $\Delta = \max_{v \in V} d(v)$ be the maximum degree of vertices in V. Then there is an edge-coloring of G that needs at most $\Delta + 1$ colors.*

Proof: We will briefly outline the main ideas of the proof as presented in [66, pp.286–287].

Clearly, any edge coloring needs at least Δ colors. The existence of an edge coloring that uses $\Delta + 1$ colors is proved by induction on the number of vertices and induction on the number of edges adjacent to the most recently added vertex. The main induction step consists of possibly re-coloring edges along some path in order to make an extra color available for coloring an edge adjacent to the most recently added vertex. The proof of the theorem is constructive and gives a polynomial-time algorithm that can be implemented in a straightforward way using recursion. In what follows, we will refer to this algorithm as the "Vizing's algorithm". Since the description of the algorithm is rather complicated, we will not present it here. The interested reader can find details of the algorithm and the complete proof in [66]. $\square$

Whether Edge Coloring is NP-hard was an open question for several years.

Theorem 2.5 (Holyer, 1981) Edge Coloring *is NP-hard.*

Proof: We will only outline the main points of Holyer's proof.

In fact, Holyer proved a slightly stronger result. He showed that it is NP-complete to decide whether a 3-regular graph can be colored with 3 or 4 colors. He used a reduction from 3SAT—a version of SAT where each clause contains exactly 3 literals. 3SAT can easily be proved to be NP-complete using a reduction from SAT. Given an instance of 3SAT, that is a collection of clauses $\mathbf{C}$ with each clause having exactly 3 literals, Holyer described a construction of a 3-regular graph G such that $\mathbf{C}$ is satisfiable iff G is 3-colorable. This construction is quite involved and we will not present it here. The interested reader can find details in [48]. $\square$

Thus the difficulty of the Edge Coloring problem is to determine whether one needs Δ or $\Delta + 1$ colors. This gives us the following theorem.

Theorem 2.6 *Vizing's algorithm approximates the* Edge Coloring *problem with an absolute performance guarantee of* $k = 1$.

It should be clear from the above example that only very special types of optimization problems can be approximated with an absolute performance guarantee. In general, these problems are such that the value of the optimum solution can be narrowed down to a small range, and the difficulty lies in determining the exact value of the optimum solution within this range. An algorithm with an absolute performance guarantee merely outputs a trivial approximate solution whose value is also in this range.

Can we actually prove that a problem cannot be approximated within an absolute performance bound? Fortunately, there are ways to do this. Typically, one uses some scaling technique to prove that if a given problem could be approximated within certain performance bound by a polynomial-time algorithm, then it could be solved exactly in polynomial time, which would imply that P=NP. Hence, assuming that P≠NP, such performance guarantee is impossible. This is illustrated by the following proposition.

Proposition 2.3 *If P≠NP, then the* Clique *problem cannot be approximated with an absolute performance guarantee.*

Proof: Given an undirected graph $G = (V, E)$, define G^m to be a graph that consists of m copies of graph G such that any two vertices belonging to a different copy are connected. Thus graph G^m has exactly $m\,|E| + m(m-1)|V|^2/2$ edges. It is easy to see that if G has a clique of size α then G^m has a clique of size $m\,\alpha$. Conversely, if G^m has a clique C_m of size β, then one can construct a clique C in G of size $\lceil \frac{\beta}{m} \rceil$—just find a copy of G in G^m that contains the most vertices of clique C_m, these vertices are all adjacent, hence form a clique in G, and by the pigeon hole principle such clique has at least $\lceil \frac{\beta}{m} \rceil$ vertices. Thus the maximum clique in G is of size α iff the maximum clique in G^m is of size $m\,\alpha$.

Now, assume that there is some approximation algorithm A for Clique with absolute performance guarantee k. Given any graph $G = (V, E)$, one can run the algorithm on the graph G^{k+1} and obtain a clique C_{k+1} of size $APP(G^{k+1})$ such that

$$|APP(G^{k+1}) - OPT(G^{k+1})| \leq k.$$

We can use the construction described above to obtain a clique C in G of size $\alpha' = \lceil \frac{APP(G^{k+1})}{k+1} \rceil$, that is $APP(G^{k+1}) \leq (k+1)\alpha'$. Let α be the size of the optimum clique in G, that is $OPT(G) = \alpha$ and $OPT(G^{k+1}) = (k+1)\alpha$. Then

$$\begin{aligned} k &\geq |APP(G^{k+1}) - OPT(G^{k+1})| = OPT(G^{k+1}) - APP(G^{k+1}) \\ &\geq (k+1)\alpha - (k+1)\alpha' = (k+1)(\alpha - \alpha'). \end{aligned}$$

Therefore

$$|APP(G) - OPT(G)| = OPT(G) - APP(G) = \alpha - \alpha' \leq \frac{k}{k+1}.$$

But both α and α' are integers, $\alpha \geq \alpha'$, hence $\alpha = \alpha'$, that is $OPT(G) = APP(G)$. Thus we found an optimal clique using a polynomial-time algorithm which is possible only if P=NP. $\square$

2.5.2 Relative performance guarantee

As discussed in the previous subsection, only a handful of NP-hard optimization problems can be approximated by polynomial-time algorithms with absolute performance guarantee. That is why the quality of algorithms for approximating NP-hard optimization problems is usually expressed in terms of bounds on the *performance ratio*.

Definition 2.21 Let A be an approximation algorithm for an optimization problem Π. The *performance ratio* $R_A(I)$ of algorithm A on an input instance I is defined as

$$R_A(I) = \frac{APP(I)}{OPT(I)}.$$

Let $R : \mathbf{N} \longrightarrow \mathbf{R}^+$. We say that an algorithm A for approximating a minimization problem Π has a *(relative) performance guarantee* $R(n)$ if for any input instance I of size n we have

$$\frac{APP(I)}{OPT(I)} = R_A(I) \leq R(n).$$

We say that an algorithm A for approximating a maximization problem Π has a *(relative) performance guarantee* $R(n)$ if for any input instance I of size n we have

$$\frac{APP(I)}{OPT(I)} = R_A(I) \geq R(n).$$

Most of the NP-hard optimization problems can be approximated by polynomial-time algorithms with some (relative) performance guarantee. However, there are problems that cannot be approximated within any (relative) performance bound.

Proposition 2.4 *If P≠NP, the* TSP *problem cannot be approximated by a polynomial-time algorithm with a relative performance bound $R(n)$ for any function $R : \mathbf{N} \longrightarrow \mathbf{R}^+$.*

Proof: Let A be an algorithm for approximating TSP with performance guarantee $R(n)$ for some function R. Consider now the following algorithm B for solving the HAMILTONIAN CYCLE problem on a given graph $G = (V, E)$ with n vertices.

1. Construct a complete edge-weighted graph $G' = (V, E', l)$ on the vertices of G such that for any $e \in E'$, $l_e = 1$ if $e \in E$ and $l_e = 1 + nR(n)$ for $e \notin E$;

2. Run algorithm A on G' to obtain a tour T in G' of total length $l(T)$.

3. Output "*yes*" if $l(T) = n$, else output "*no*".

Clearly, $l(T) = n$ iff T contains only edges of G. Thus if B outputs "*yes*", G has a Hamiltonian cycle.

Assume now that G has a Hamiltonian cycle but B outputs "*no*". Then $OPT(G') = n$ and $l(T) \neq n$. Thus T contains at least one edge that does not belong to E, hence $APP(G') = l(T) \geq n + nR(n)$, that is $R_A(G') \geq R(n) + 1$ which contradicts the performance guarantee on A.

Therefore G has a Hamiltonian cycle iff the length of the tour output by A equals n. Thus B is a polynomial-time algorithm that solves HC and that is possible only if P=NP.
□

The above example is clearly not very natural, in fact, for most of the instances of TSP considered in practice, the length function satisfies the triangle inequality, that is for any three vertices $u, v, w \in V$, $l(u, w) \leq l(u, v) + l(v, w)$. (Here $l(u, v) = l_e$ is the length of the edge $e = (u, v)$, i.e. the distance from u to v.) In such a case, TSP is sometimes referred to as the Metric Traveling Salesman Problem and can be approximated quite well. In what follows, we will assume that the cost function indeed satisfies the triangle inequality and will refer to Metric TSP simply as TSP.

Consider now the *nearest-neighbor algorithm* for approximating (metric) TSP. Given an undirected edge-weighted graph $G = (V, E, l)$ with n vertices, the nearest-neighbor algorithm builds the TSP tour one edge at a time, at each step adding the unused vertex closest to the end of the current path. More formally, it can be described as follows.

1. Select and arbitrary vertex $a_1 \in V$—the beginning of a path.
2. If the current path is $\langle a_1, a_2, \ldots, a_k \rangle$, find a vertex a_{k+1} not yet on the path closest to a_k, and add a_{k+1} to the path.
3. Repeat step 2 until the path visits all vertices of G.
4. Create the TSP tour by adding an edge connecting vertex a_n with the starting point a_1 of the path.

Note: We assume that ties in step 2 are broken arbitrarily. However, since we are establishing *worst-case* performance bounds, we always have to account for the worst scenario. Thus we always have to consider the possibility that the algorithm chooses vertices from among the ones closest to the end of the path that lead, ultimately, to the worst-possible solution.

Theorem 2.7 (Rosenkrantz, Stearns, and Lewis, [82]) *The nearest-neighbor algorithm approximates (metric)* TSP *with performance guarantee of* $R(n) = 1/2\lceil \log_2(n) \rceil + 1/2$.

We will not present the proof of this theorem here. The interested reader can find it in [82].

Theorem 2.8 (Rosenkrantz, Stearns, and Lewis, [82]) *For each $m > 3$, there exists an undirected edge-weighted graph $G = (V, E, l)$ on $n = 2^m - 1$ vertices with the distance function satisfying the triangle inequality such that the tour output by the nearest-neighbor algorithm satisfies*

$$\frac{APP(G)}{OPT(G)} > \frac{1}{3}\log_2(n+1) + \frac{4}{9}.$$

Proof: The proof is quite complicated and we will not present it here. The main idea is to construct a graph in such a way that the nearest-neighbor algorithm uses mostly short intermediate edges to build the TSP tour but the last edge is very long. On the other hand, the optimum tour consists of short edges only. □

Thus the bound of $R(n) = 1/2\lceil\log_2(n)\rceil + 1/2$ on the performance of the nearest-neighbor algorithm is asymptotically tight.

The nearest-neighbor algorithm is an example of approximation algorithm where the performance bound depends on the size of the input instance. Can we do better than this? It turns out, that TSP can be approximated with constant performance guarantee. Before we prove this, let us discuss the Minimum Spanning Tree problem.

It is well-known that MST can be solved *exactly* in polynomial time. Most of the algorithms for solving this problem are based on a greedy strategy, building the spanning tree one edge at a time, at each step choosing the shortest edge satisfying certain properties. Consider, for example, the following algorithm that given a graph $G = (V, E)$ builds the spanning tree T one edge at a time, each time adding to T the shortest edge "leaving" T, that is the shortest edge among those with one end-point in $V(T)$ and other end-point in $V \setminus V(T)$.

1. Set $T = \{v_1\}$ for some arbitrary vertex $v_1 \in V$.

2. Add to T the shortest edge leaving T.

3. Repeat step 2 until tree T spans entire G.

It is straightforward to show that tree T output by this algorithm is an optimum solution of the MST problem. The algorithm can be easily implemented to run in time $O(|V|^2)$—see [75, p.273].

One of the simple well-known algorithms for approximating (metric) TSP with a strong performance guarantee is the following *minimum-spanning-tree algorithm* (see for example [62] for details):

1. Find a minimum spanning tree T of G.
2. Build a TSP tour from T by traversing T in a depth-first fashion and adding each node encountered for the first time to the tour.

Note: Step 2 can be equivalently described as first creating a closed path by the depth-first traversal of T and then introducing shortcuts by removing all repeated vertices.

Note: Depth-first traversal refers to the order in which the vertices of a tree are systematically visited. We start with a given vertex, go to its previously unvisited neighbor, then go to the previously unvisited neighbor of the new vertex, and repeat this until we get stuck, that is until we reach a vertex that does not have any unvisited neighbors (a leaf). In such case, we back-track to the vertex immediately before the current vertex in the traversal and repeat the traversing process until this vertex has no unvisited neighbors. Then we again backtrack to the vertex before this one, and so on. The traversal is over when all the vertices of the tree have been visited, that is when we have back-tracked to the starting vertex and it has no unvisited neighbors.

Figure 2.4 shows a tree together with the order in which the vertices of the tree were first visited by a depth-first traversal.

The following two theorems can be found in [62] or [75] and are considered part of the folklore.

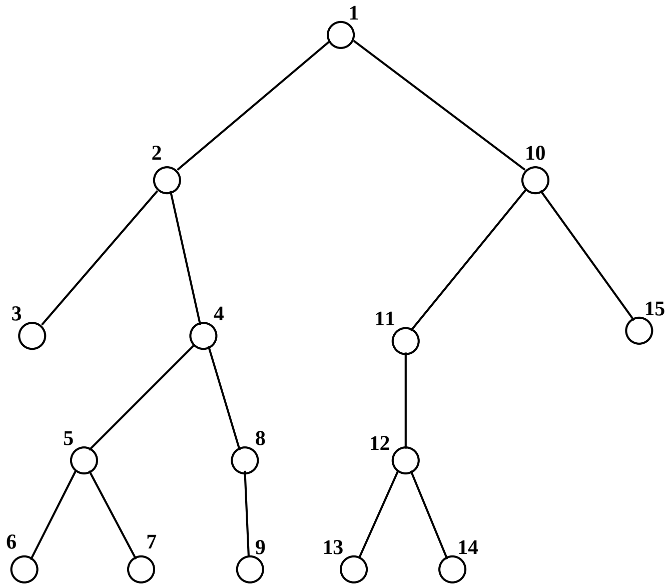


Figure 2.4: Depth-first traversal of a tree

Theorem 2.9 *Given a complete edge-weighted graph $G = (V, E)$ with $|V| = n$ such that the distance metric is symmetric and satisfies the triangle inequality, the minimum-spanning-tree algorithm described above approximates* TSP *with performance guarantee of*

$$R(n) = 2 - \frac{2}{n} \leq 2$$

and runs in time $O(n^2)$.

Proof: Let OPT be the length of the optimum TSP tour, APP the length of the tour output by the minimum-spanning-tree algorithm, and $l(T)$ the length of some minimum spanning tree T. Since the distance function satisfies the triangle inequality, we have that $APP \leq 2l(T)$. On the other hand, removing an edge from some tour results in a spanning tree. In particular, if we remove the longest edge from the optimal tour, we obtain some spanning tree T' such that $l(T') \geq l(T)$. Clearly, the longest edge in the optimal tour has length at least $\frac{OPT}{n}$, hence $l(T') \leq (1 - \frac{1}{n})OPT$. Combining the above inequalities, we have

$$APP \leq 2l(T) \leq 2l(T') \leq (2 - \frac{2}{n})OPT.$$

The running time of the algorithm is dominated by finding the MST, hence it can be easily implemented in $O(|V|^2)$ time. $\square$

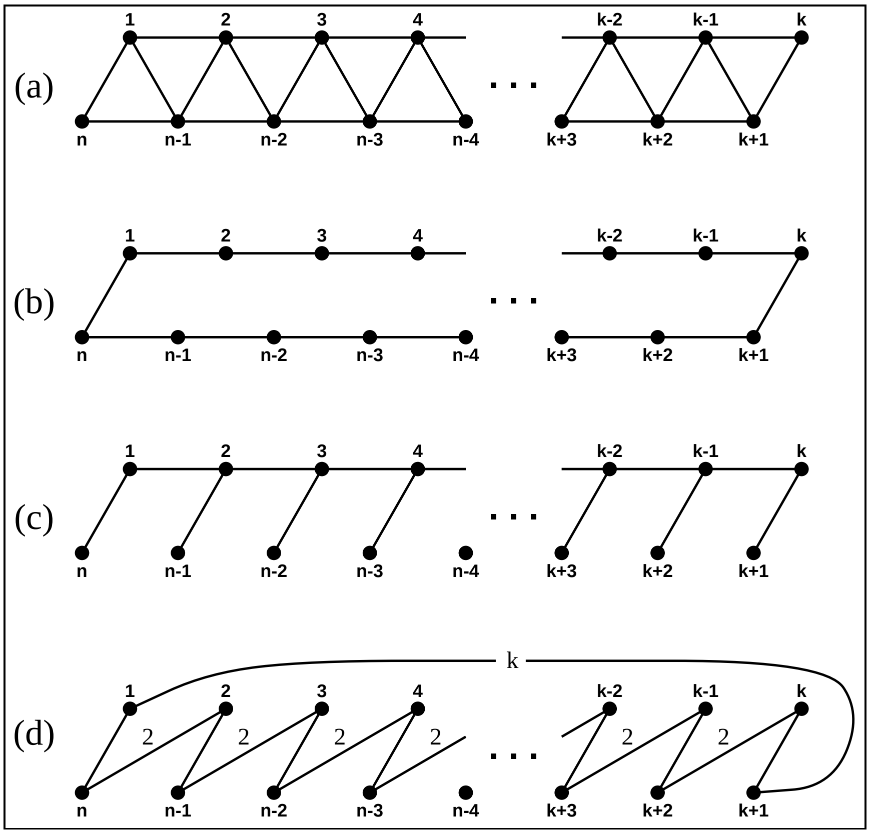


Figure 2.5: Graph $G = (V, E)$ with $n = 2k = |V|$, on which the performance ratio of the minimum-spanning-tree algorithm is $2 - 2/n$.

(a) A complete graph $G = (V, E)$ where only edges of length 1 are drawn. The edges that are not drawn have length equal to the length of the shortest path between the end-points along the drawn edges.

(b) The optimal TSP tour of length n.

(c) A minimum spanning tree T of length $n - 1$.

(d) The TSP tour of length $2n - 2$ obtained by a depth-first traversal with shortcuts of vertices of T starting at vertex 1, that first visits the high-numbered vertices.

Theorem 2.10 *For every $n \geq 2$ there exists an instance of* TSP *of size n, that is an edge-weighted graph $G = (V, E)$ with n vertices, for which the performance ratio of the* *minimum-spanning-tree algorithm is*

$$\frac{APP}{OPT} = \begin{cases} 2 - \frac{2}{n}, & \textit{for } n \textit{ even, and} \\ \\ 2 - \frac{3}{n}, & \textit{for } n \textit{ odd.} \end{cases}$$

Proof: Consider the complete graph $G = (V, E)$ with $|V| = n = 2k$ from Figure 2.5(a). Note that only edges of length 1 are drawn. The length of an edge $\{u, v\}$ that is not drawn is equal to the length of the shortest path between u and v consisting of drawn edges. Thus the length of the edge $\{2, n\}$ is 2, the length of the edge $\{3, n\}$ is 3, etc. Clearly, the length of the optimum TSP tour is $OPT = n$ and the length of a minimum spanning tree T is $l(T) = n - 1$. We assume that the depth-first traversal of T starts at vertex 1 and each time there is a choice, the vertex with higher label is visited first. Thus the vertices will be visited in the following order:

$\langle 1, n, 1, 2, n-1, 2, 3, n-2, 3, \ldots, k-1, k+2, k-1, k, k+1, k, k-1, k-2, \ldots, 3, 2, 1 \rangle.$

Deleting each vertex that appears more than once, that is short-cutting the tour obtained by the depth-first traversal of T, starting from the beginning, we obtain the approximate TSP tour

$$\langle 1, n, 2, n-1, 3, n-2, 4, \ldots, k-1, k+2, k, k+1 \rangle.$$

The tour contains k edges of length 1, $k-1$ edges of length 2 and 1 edge of length k, hence $APP = 4k - 2 = 2n - 2$. Therefore

$$\frac{APP}{OPT} = 2 - \frac{2}{n}.$$

The proof for $|V| = n = 2k + 1$ is almost identical. $\square$

Note: Theorems 2.9 and 2.10 present slightly stronger results than the ones in [75] or [62].

The above results show that the TRAVELING SALESMAN PROBLEM can be approximated by the *minimum-spanning-tree algorithm* with a performance bound of 2 and that this bound is asymptotically tight. Thus we can approximate (metric) TSP with a *constant* performance guarantee which is a significant improvement over the performance of the nearest-neighbor algorithm. But we can do even better! Before we do so, let us first discuss some auxiliary problems.

Definition 2.22 Given a graph $G = (V, E)$, a *matching* $M \subseteq E$ is a collection of edges of G such that no two edges in M share an endpoint. We say that M is a *perfect matching* if $|V|$ is even and $|M| = |V|/2$.

MINIMUM-WEIGHT PERFECT MATCHING

Instance: Undirected edge-weighted graph $G = (V, E)$ with an even number of vertices.

Goal: Find a perfect matching of minimum total weight.

Similar to MST, the MINIMUM-WEIGHT PERFECT MATCHING problem is also solvable in polynomial time even though the algorithm is more complicated than the one for MST and thus we will not present it here. The first algorithm for solving MINIMUM-WEIGHT PERFECT MATCHING was given by Edmonds in [30] and had a running time of $O(|V|^4)$. It was later improved by Gabow [35] to run in time $O(|V|^3)$—see [75, pp.255–269] for details. More recently, Gabow [35] and Gabow and Tarjan [36] designed even faster algorithms.

Definition 2.23 A *multigraph* $G = (V, E)$ is a (generalized) graph where multiple edges between pairs of vertices are allowed. An *Eulerian tour* in a multigraph $G = (V, E)$ is a closed path that visits each vertex of G at least once and uses each edge of G exactly once. A multigraph $G = (V, E)$ is called *Eulerian* if it has an Eulerian tour.

Theorem 2.11 *A multigraph $G = (V, E)$ is Eulerian if and only if G is connected and all vertices in V have even degree.*

Proof: We will closely follow the proof given in [75, p.412].

Clearly, if a multigraph $G = (V, E)$ has an Eulerian tour, then G is connected and all vertices have even degree. We will use induction on the number of edges of G to show how to construct an Eulerian tour in a connected multigraph with all vertices of even degree.

If G is a connected multigraph with no edges, that is G contains just one vertex v, the trivial path consisting of v is an Eulerian tour, hence G is Eulerian.

Let $G = (V, E)$ be a connected multigraph with all vertices of even degree and assume that the theorem is true for all multigraphs with smaller number of edges. Choose a vertex v in V and build a closed path P in G starting at v that uses each edge in E at most once and returns back to v. The construction of such path is trivial—at each step, choose any unused edge incident on the last visited vertex and continue until reaching v. The existence of such an edge is guaranteed by the fact that the degree of each vertex is even, thus we can never get "stuck" in any vertex different from v.

Now, remove all edges traversed by P from G. Thus G now consists of several connected components $G_1, \ldots, G_k$ each with all vertices of an even degree. Each component G_i has a smaller number of edges than the original graph G, hence it has an Eulerian tour P_i. Since the original graph G was connected, each tour P_i has at least one vertex in common with P. Denote such vertex by u_i. Now, create a new tour $P + P_1 + \cdots + P_k$ in the following way. Traverse edges of P starting at vertex v, each time a vertex u_i for some $i = 1, \ldots, k$ is encountered for the first time, traverse the Euler tour P_i returning back to u_i and continue traversing the remaining edges of P. Thus the tour $P + P_1 + \cdots + P_k$ is obtained from P by "inserting" the Eulerian tours P_i into P. Clearly, $P + P_1 + \cdots + P_k$ visits all vertices of G and uses each edge of G exactly once; hence it is an Eulerian tour in G. $\square$

The above proof gives a recursive algorithm for constructing an Eulerian tour in any connected multigraph with vertices of even degree. It can be easily implemented to run in time $O(|E|)$—see [75, p.413] for details.

Now we are ready to describe the *Christofides algorithm* [23] for approximating TSP. This algorithm is more complicated than the minimum-spanning-tree algorithm but achieves a performance guarantee of $\frac{3}{2}$, which is the best known performance bound for approximating metric TSP.

1. Find a minimum spanning tree T of $G = (V, E)$.

2. Let $G' \subseteq G$ be the complete subgraph of G induced by those vertices in T with an odd degree (G' has an even number of vertices). Find a minimum-weight perfect matching M in G'.

3. The multigraph $T \cup M$ is Eulerian hence we can construct an Eulerian tour in $T \cup M$.

4. Introduce shortcuts in the Eulerian tour to obtain a TSP tour without repeated vertices.

Theorem 2.12 *Given a complete edge-weighted graph $G = (V, E)$ with $|V| = n$ such that the distance metric is symmetric and satisfies the triangle inequality, the Christofides algorithm approximates* TSP *with a performance guarantee of*

$$R(n) = \frac{3}{2} - \frac{1}{n} \leq \frac{3}{2}$$

and runs in time $O(n^3)$.

Proof: Let OPT be the length of the optimum TSP tour, APP the length of the tour output by the Christofides algorithm, $l(T)$ the length of some minimum spanning tree T, $l(M)$ the length of the minimum perfect matching on odd-degree vertices of T, and $l(P)$ the length of the Eulerian tour in $T \cup M$. Since the Eulerian tour P traverses each edge of $T \cup M$ exactly once, it must be that $l(P) = l(T) + l(M)$. And since the distance function satisfies the triangle inequality, we have that $APP \leq l(P)$. Similarly as in the proof of Theorem 2.9, we have that $l(T) \leq (1 - 1/n)OPT$. It is easy to see that the minimum-weight perfect matching on any complete graph $G' \subseteq G$ with even number of vertices is no longer than a half of the optimum tour, thus $l(M) \leq OPT/2$. Combining the above inequalities, we have

$$APP \leq l(P) = l(T) + l(M) \leq (1 - \frac{1}{n})OPT + \frac{OPT}{2} = (\frac{3}{2} - \frac{1}{n})OPT.$$

The running time is dominated by finding the minimum-weight perfect matching hence the Christofides algorithm runs in time $O(|V|^3)$. $\square$

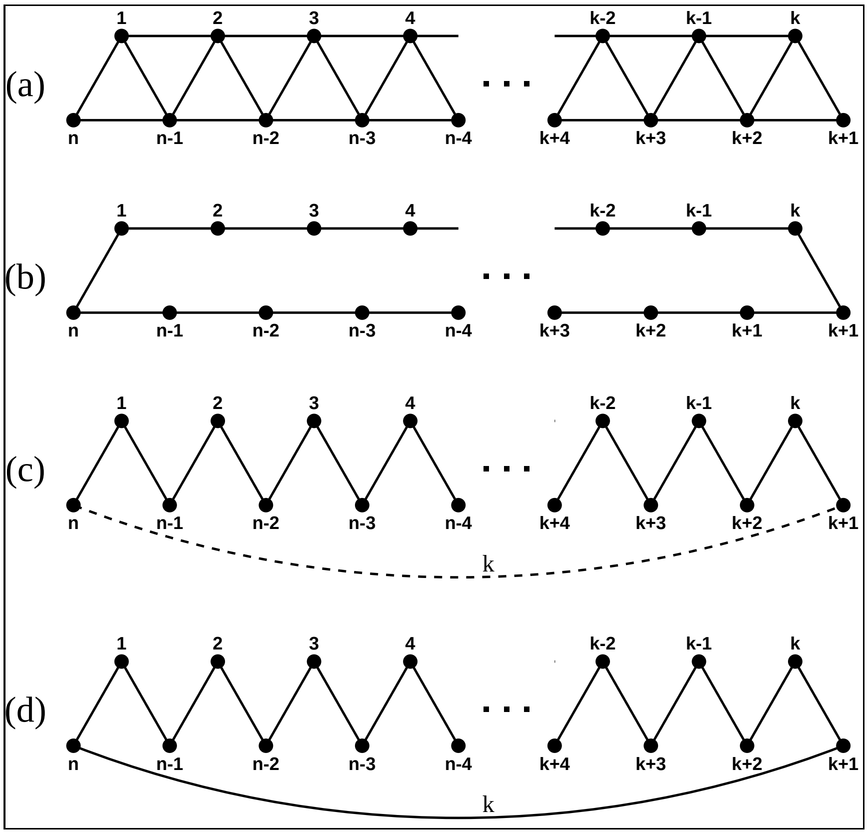


Figure 2.6: Graph $G = (V, E)$ with $n = 2k + 1 = |V|$, on which the performance ratio of the Christofides algorithm is $\frac{3}{2}(1 - 1/n)$.

(a) A complete graph $G = (V, E)$ where only edges of length 1 are drawn. The edges that are not drawn have length equal to the length of the shortest path between the end-points along the drawn edges.

(b) The optimal TSP tour of length n.

(c) A minimum spanning tree T of length $n - 1$—solid edges, and the minimum-weight perfect matching M on odd-degree vertices of T—dashed edge.

(d) The Eulerian tour in $T \cup M$ of length $\frac{3}{2}(n - 1)$. Since each vertex is visited exactly once, no shortcuts are necessary.

Theorem 2.13 ([62, p.162]) *For every $n \geq 2$ there exists an instance of* TSP *of size n, that is an edge-weighted graph $G = (V, E)$ with n vertices, for which the performance ratio* *of the Christofides algorithm is*

$$\frac{APP}{OPT} = \frac{3}{2}(1 - \frac{1}{n}).$$

Proof: Consider the complete graph $G = (V, E)$ with $|V| = n = 2k+1$ from Figure 2.6(a). As was the case in Figure 2.5, only edges of length 1 are drawn. The length of an edge $\{u, v\}$ that is not drawn is equal to the length of the shortest path between u and v consisting of drawn edges.

Clearly, the length of the optimum TSP tour is $OPT = n$ and the length of a minimum spanning tree T is $l(T) = n - 1 = 2k$. Graph G' that contains the odd-degree vertices of T consists of only two vertices—n and $k + 1$, hence matching M contains only one edge, namely $\{k+1, n\}$, hence $l(M) = k$. The edges of $T \cup M$ form a simple Eulerian tour, hence no shortcuts are necessary. The length of the tour is clearly $APP = k + 2k$, therefore

$$\frac{APP}{OPT} = \frac{3k}{n} = \frac{3}{2}(1 - \frac{1}{n}).$$

The proof for n even is almost identical. $\square$

Thus the Christofides algorithm approximates (metric) TSP with a *constant* performance guarantee of $\frac{3}{2}$ and this bound is asymptotically tight.

Another problem that can be approximated with constant performance guarantee and is closely related to our results in Chapter 5 is the Steiner Tree problem. Even though this problem seems very similar to the MST problem, it can be shown to be NP-hard—see [56] for details. The difficulty in solving this problem lies in the selection of extra Steiner vertices to be used in the tree spanning the terminals.

Definition 2.24 Given a graph $G = (V, E)$ and a subset W of V, a *W-Steiner tree* T is a subgraph of G such that T is a tree and $W \subseteq V(T)$. We call the vertices in W *terminals* and the vertices in $V \setminus W$ *Steiner vertices.*

Note: Even though the W-Steiner tree clearly depends on the set W, one usually refers to it simply as the Steiner tree. In the rest of the thesis, we will follow this standard terminology.

Steiner Tree Problem
Instance: Undirected edge-weighted graph $G = (V, E)$ and a set $W \subseteq V$ of terminals.
Goal: Find a Steiner tree of minimum length.

Steiner Tree can be approximated by a minimum spanning tree on W within a factor of 2—see, for example, [93] or [42]. This performance bound has been improved to $\frac{11}{6}$ by Zelikovsky [96, 97], later by Berman and Ramaiyer [16], and most recently by Karpinski and Zelikovsky to 1.644 [57]. These heuristics are more complicated and improve on the algorithm that computes the MST on W by taking into consideration additional vertices outside W.

Let us conclude this subsection by an example of an NP-hard optimization problem that cannot be approximated with a constant performance bound.

Theorem 2.14 (Bellare, Goldwasser, Lund, and Russell, [15]) Set Cover *cannot be approximated with a constant performance guarantee unless* P=NP*.*

The proof of this theorem is quite involved and uses techniques far beyond the scope of this chapter. In the following chapter, we will show that a simple greedy algorithm can approximate Set Cover with performance guarantee of $R(m) = \ln m - \ln\ln m + 0.78$, where $m = |U|$, and that this bound is tight up to a constant.

The above results illustrate the main goals of the design and analysis of approximation algorithms—to obtain algorithms with better and better performance bounds, and to make the algorithms faster and faster. On the other hand, there are results that impose limits on achievable performance of approximation algorithms. We presented results showing that the performance bound of a particular algorithm cannot go below a certain limit. Such results still leave some hope for finding a different algorithm with a better performance guarantee. We also presented more powerful results proving hardness of approximating a given problem, that is results establishing limits on the performance that cannot be improved by *any* approximation algorithm for a given problem unless some collapses in the polynomial hierarchy of complexity classes occur. And since such collapses are very unlikely, we are left with almost no hope for improving performance bounds beyond such limits. Thus it is of independent interest to find algorithms that at least match these limits. In Chapter 3 we analyze an algorithm that achieves such inapproximability bound.

2.5.3 Polynomial-time Approximation Schemes

The minimum-spanning-tree and Christofides algorithms for approximating the TRAVELING SALESMAN PROBLEM illustrate the trade-off between quality of approximation and the running time of the algorithms. This a typical phenomena occurring very often in the design of approximation algorithms. This subsection discusses algorithms with a built-in interplay between the running time and the quality of approximation. We will closely follow the exposition given in [69].

Definition 2.25 A *polynomial-time approximation scheme* (PTAS) for a problem Π is a family of polynomial-time algorithms $\{A_\varepsilon \mid \varepsilon > 0\}$ such that the value $APP(I, \varepsilon)$ of the approximate solution returned by A_ε on instance I satisfies

$$\frac{|APP(I,\varepsilon) - OPT(I)|}{OPT(I)} \le \varepsilon.$$

If Π is a minimization problem, this can be rewritten as

$$\frac{APP(I,\varepsilon)}{OPT(I)} \le 1 + \varepsilon.$$

If Π is a maximization problem, this can be rewritten as

$$\frac{APP(I,\varepsilon)}{OPT(I)} \ge 1 - \varepsilon.$$

Thus a PTAS is a family of algorithms that can approximate given problem with performance guarantee arbitrarily close to 1. We will design a PTAS for approximating the KNAPSACK problem. Let us first recall the definition of this problem.

KNAPSACK PROBLEM

Instance: A capacity B of the knapsack and a set $U = \{1, 2, \ldots, n\}$ of items where each item i has size s_i and profit p_i associated with it.

Goal: Find a subset $U' \subseteq U$ of items such $\sum_{i \in U'} s_i \leq B$ (i.e. the items fit in the knapsack) and the total profit $\sum_{i \in U'} p_i$ is maximum.

Consider the following natural algorithm for approximating KNAPSACK based on the greedy strategy. For any $U' \subseteq U$, we will denote by $p(U')$ the total profit of elements in U', that is $p(U') = \sum_{i \in U'} p_i$, and by $s(U')$ the total size of elements in U', that is $s(U') = \sum_{i \in U'} s_i$.

1. Sort the items in U in such a way that if $i < j$, then $\frac{p_i}{s_i} \geq \frac{p_j}{s_j}$, that is put items with high profit densities first.

2. Set $U' = \emptyset$.

3. **for** $i = 1$ **to** n **do**
   if $s(U') + s_i \leq B$ (that is if item i fits in the knapsack)
   then $U' \leftarrow U' \cup \{i\}$ (put item i in the knapsack)

Unfortunately, there are instances of KNAPSACK on which this simple algorithm does not perform well. Consider, for example, the case where set U has only two items—one of size 1 and profit 2, the other of size B and profit B. Our approximation algorithm will output a solution of total profit 2 whereas the optimum profit is B. This is certainly not acceptable for large values of B. However, a simple modification (called *k-enumeration*) of this naive algorithm will give us a PTAS for KNAPSACK.

Consider the following algorithm which we call B_k.

1. Sort the items in U in the order of decreasing profit densities.

2. For each subset U' of U with less than k items that fit in the knapsack, evaluate the total profit.

3. For each subset $\hat{U}$ of U containing exactly k items that fit in the knapsack do the following:

   (a) Set $U' = \hat{U}$.

   (b) **for** all $i \in U \setminus \hat{U}$ **do** (in the order of decreasing profit densities)
       if $s(U') + s_i \leq B$
           then $U' \leftarrow U' \cup \{i\}$

4. Output that set U' from steps 2 or 3 with maximum total profit.

Theorem 2.15 *For all $k \geq 1$, the algorithm B_k has a performance guarantee $R_k = 1 - \frac{1}{k+1}$ and runs in time $O(n^{k+1})$.*

Proof: Step 1 of algorithm B_k can be easily implemented to run in time $O(n \log n)$ using, for example, merge sort. Step 2 can be completed in time

$$O\left(\binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{k-1}\right) = O(n^{k-1}).$$

The time needed for step 3 of the algorithm is $O\left(\binom{n}{k}(n-k)\right) = O(n^{k+1})$. Thus the total running time of the algorithm B_k is $O(n^{k+1})$.

Given an instance $I = (B, U, \{p_1, p_2, \ldots, p_n\}, \{s_1, s_2, \ldots, s_n\})$ of the KNAPSACK problem, let $X \subseteq U$ be an optimum solution, $|X| = x$. If $x \leq k$ then algorithm B_k outputs an optimum solution and that trivially satisfies the performance bound.

Assume now that $x > k$. Without loss of generality assume that the items in U are numbered in the order of decreasing profit densities. Let $Y \subseteq X$ be the set of k items of X with the largest profits and let $Z = X \setminus Y$ be the remaining items in X.

Since $|Y| = k$, at some point, the algorithm B_k will try the set Y as its initial set $\hat{U}$. Let us now concentrate on this iteration of B_k. Denote by m the first item in Z that was not placed in the knapsack by B_k and set $Z_m = \{i \in Z \mid i < m\}$. Then Z_m contains items in Z already placed in the knapsack and $Z \setminus Z_m$ contains all the items in Z that until this point have not been placed in the knapsack. Hence all items in $Z \setminus Z_m$ have profit densities no bigger than item m. We denote by G the items added to U' in the step 2(b) of the algorithm B_k until the point when m was rejected. Thus $U' = Y \cup G$, $Z_m = U' \cap Z \subseteq G$, and G contains some items of $U \setminus X$. Set $\Delta = s(G) - s(Z_m)$, that is Δ is the total size of items in $G \setminus Z_m$. Then $p(G) \geq p(Z_m) + \Delta \frac{p_m}{s_m}$, since all the items in $G \setminus Z_m$ have higher profit densities than element m.

Since item m does not fit in the knapsack, we have $B - s(U') < s_m$. Clearly $s(Z \setminus Z_m) \leq B - s(Y) - s(Z_m)$, hence $p(Z \setminus Z_m) \leq s(Z \setminus Z_m)\frac{p_m}{s_m} \leq (B - s(Y) - s(Z_m))\frac{p_m}{s_m}$.

Therefore

$$\begin{aligned}
p(X) &= p(Y) + p(Z_m) + p(Z \setminus Z_m) \\
&\leq p(Y) + p(G) - \Delta\frac{p_m}{s_m} + (B - s(Y) - s(Z_m))\frac{p_m}{s_m} \\
&= p(Y) + p(G) + (B - s(Y) - s(G))\frac{p_m}{s_m} \\
&= p(U') + (B - s(U'))\frac{p_m}{s_m} \\
&< p(U') + p_m.
\end{aligned}$$

And since the solution output by B_k has value $APP(I,k) \geq p(U')$, we have

$$OPT(I) - APP(I,k) < p_m.$$

Now, X contains at least k other items with profits at least as big as p_m, hence $p(X) = OPT \geq (k+1)p_m$. Hence we have that

$$\frac{OPT(I) - APP(I,k)}{OPT(I)} < \frac{1}{k+1}$$

that is

$$\frac{APP(I,k)}{OPT(I)} > 1 - \frac{1}{k+1}.$$

□

To obtain a PTAS, it is now enough to define $A_\varepsilon = B_k$ for $k = \lceil \frac{1}{\varepsilon} \rceil - 1$, since for such k, $\frac{1}{k+1} \leq \varepsilon$, and hence $APP/OPT > 1 - \varepsilon$. This proves the following theorem.

Theorem 2.16 *The family $\{A_\varepsilon \mid \varepsilon > 0\}$ of algorithms described above is a polynomial-time approximation scheme for the* Knapsack *problem. For any $\varepsilon > 0$, the running time of algorithm A_ε is $O(n^{\lceil \frac{1}{\varepsilon} \rceil})$.*

The polynomial-time approximation scheme for Knapsack described above exhibits a trade-off between performance and running time typical for approximation schemes. In this case, the running time depends *exponentially* on $\frac{1}{\epsilon}$ hence any small improvement in the performance guarantee can be accomplished only by a significant increase in the running time. Thus, despite the fact that *theoretically* we have algorithms with performance guarantee arbitrarily close to 1, such algorithms have no practical use.

Thus in general, we would like to design approximation schemes where improvements in the performance do not require too big an increase in the running time. As was the case for the dependency of running time on the length of the input, our criteria for "not too big" increase in running time is again based on polynomial dependency.

Definition 2.26 A *fully polynomial-time approximation scheme* (FPTAS) $\{A_\varepsilon \mid \varepsilon > 0\}$ is a polynomial-time approximation scheme whose running time depends polynomially on both the length of the input instance and $\frac{1}{\varepsilon}$.

The proof of the following theorem together with further discussion on fully polynomial-time approximation schemes can be found in [69, pp.41–52]

Theorem 2.17 *The family $\{A_\varepsilon \mid \varepsilon > 0\}$ of algorithms described in [69, p.41] is a fully polynomial-time approximation scheme for approximating the* Knapsack *problem and runs in time $O\left(\frac{n^3 \log B}{\varepsilon}\right)$.*

For fully polynomial-time approximation schemes, the trade-off between running time and performance is much more acceptable. Clearly, this is the best relative performance bound one can hope for when approximating NP-hard optimization problems. Thus it should be no surprise that only a handful of optimization problems permit fully polynomial-time approximation schemes or even polynomial-time approximation schemes.

2.5.4 Conclusion

In this chapter, we introduced basic complexity classes, polynomial-time reductions between decision problems, and concepts of NP-completeness and NP-hardness. Using several different examples, we showed how one can approximate NP-hard optimization problems, what is involved in establishing tight performance bounds, how some problems can be approximated with only a small error whereas some others cannot be approximated within any reasonable bounds. We saw that the research in approximation algorithms can be roughly divided into two major areas—design and analysis of provably better and faster approximation algorithms, and proving hardness of approximation. We presented some basic results from both areas, however, the rest of this thesis is fully devoted to the design and analysis of approximation algorithms. For further reading, the interested reader is strongly encouraged to pursue the references mentioned throughout the chapter.

Chapter 3

The Set Cover Problem

In this chapter we establish significantly improved bounds on the performance of the greedy algorithm for approximating Set Cover. In particular, we provide a substantial improvement of the 20 year old classical harmonic upper bound $H(m)$ of Johnson, Lovász, and Chvátal, by showing that the performance ratio of the greedy algorithm is, in fact, *exactly* $\ln m - \ln\ln m + \Theta(1)$, where m is the size of the ground set. The difference between the upper and lower bounds turns out to be less than 1.1. This provides the first tight analysis of the greedy algorithm, as well as the first upper bound that lies below $H(m)$ by a function going to infinity with m.

We also show that the approximation guarantee for the greedy algorithm is better than the guarantee recently established by Srinivasan for the randomized rounding technique, thus improving the bounds on the integrality gap.

Our improvements result from a new approach which might be generally useful for attacking other similar problems.

3.1 Introduction

Set Cover is one of the oldest and most studied problems in the area of combinatorial optimization. It has a strong theoretical value and is also used in practical applications like testing of VLSI devices [22] or crew scheduling [21]. It can be informally stated as follows:

Given a collection of subsets of some ground set U that cover U, find a subcover containing the smallest possible number of subsets.

The decision version of SET COVER (Is there a cover of U consisting of at most K subsets?) was one of the first 21 problems shown to be NP-complete by Karp in his seminal paper "Reducibility among Combinatorial Problems" [56] published in 1972. One of the best polynomial-time algorithms for approximating SET COVER is the greedy algorithm: at each step choose the subset that covers the largest number of remaining elements. Johnson [55] and Lovász [64] independently showed that the performance ratio of the greedy method is no worse than $H(m)$, where $H(m) = 1 + \cdots + 1/m$ is the m^{th} harmonic number, a value which is clearly between $\ln m$ and $\ln m + 1$. Chvátal in [24] extended their results to the weighted version of the problem.

The original classical analysis of the greedy algorithm has remained essentially unchanged for the last 20 years, despite the fact that $H(m)$ was not known to be a lower bound on the performance ratio of the greedy algorithm. In fact, Johnson in his well-known paper [55] provides a lower bound of only about $0.48 \ln m$. A straightforward modification of his approach gives a lower bound of about $0.72 \ln m$.

Other, more complex approximation algorithms, have also been studied. For example, Halldórsson's "local improvements" modification of the greedy algorithm ([45], [46]) improved the upper bound to about $H(m) - 0.43$ and suggested that for large ground sets this improvement can be made even stronger. His bounds were very recently further improved by Duh and Fürer [28] to $H(m) - 1/2$. Srinivasan's analysis of randomized rounding ([92]) showed some further improvements on the performance ratio in special cases, making it appear at that point that the randomized rounding algorithm was better than the greedy method.

3.1.1 Hardness of Approximating the Set Cover Problem

This state of affairs was put more sharply into focus by recent hardness results of Lund and Yannakakis [67], Bellare, Goldwasser, Lund, and Russell [15], Feige [32], Raz and Safra

[79], Arora and Sudan [11], and others. First, Lund and Yannakakis [67] showed that unless NP $\subseteq DTIME(n^{\text{polylog}\, n})$, Set Cover cannot be approximated within $c \log_2 m$ for any $c < 1/4$. Later results tried to improve this bound, notably Bellare et al. showed that unless P = NP, Set Cover cannot be approximated within any constant ratio. Most recently, Raz and Safra [79], and Arora and Sudan [11] proved that, unless P = NP, there is no $c \log m$-approximation algorithm for Set Cover for some $c > 0$. The efforts to show hardness of approximation for the Set Cover problem were nearly completed by Feige [32]. He proved a very strong result showing that for any $\epsilon > 0$, no polynomial-time algorithm can approximate Set Cover within $(1-\epsilon)\ln m$ unless NP $\subseteq DTIME[n^{O(\log\log n)}]$. Hence, under a plausible structural complexity assumption, the performance ratio of any polynomial time algorithm can improve on the harmonic bound by at most a function $f(m) = o(\ln m)$.

The above hardness results originated from new probabilistic characterizations of NP, the most well-known among them being the PCP Theorem [10, 9]. This theorem states that NP = PCP$(\log n, 1)$; that is, the class NP equals to a class of languages with membership proofs probabilistically checkable by $(\log n, 1)$-restricted verifier. The interested reader can find details in the above mentioned papers and in [6]. Recent results about the hardness of approximating NP-hard optimization problems can be found in survey [8].

3.1.2 Our Results

In this chapter, we provide a substantial improvement of the harmonic bound for approximating the Set Cover problem. Using a new approach, we show that the performance ratio of the greedy algorithm is exactly $\ln m - \ln\ln m + \Theta(1)$. In fact, our analysis proves that the performance ratio never exceeds the value $\ln m - \ln\ln m + 0.78$ and might be as bad as $\ln m - \ln\ln m - 0.31$. Thus the lower and upper bounds differ by less than 1.1. This provides the first tight analysis of the greedy algorithm, as well as the first upper bound on approximating Set Cover that lies below $H(m)$ by a function going to infinity with m. Clearly, the above mentioned hardness result of Feige adds to the relevance of our result, since lower-order improvements in the harmonic bound for any polynomial-time algorithm are now the best we can hope for. Moreover, this improved analysis holds out the prospect of further identifying the exact approximation threshold at which the SET COVER problem becomes intractable.

The second part of this chapter extends our results to fractional covers. A fractional cover consists of fractions of covering sets with the condition that for each point the fractions add up to at least 1. This enables us to compare our results with Srinivasan's bounds [92], and show that our bound on the integrality gap for the greedy algorithm is significantly better than that for randomized rounding.

3.1.3 Formal Definitions

Definition 3.1 Given a collection $\mathbf{S} = \{S_1, \ldots, S_n\}$ of subsets of some finite set U, we say that $\mathbf{S}$ is a *cover* of U if $\bigcup_{i=1}^{n} S_i = U$. We say that $\mathbf{S}^* = \{S_{i_1}, \ldots, S_{i_l}\}$ is a *subcover* of U if $\mathbf{S}^* \subseteq \mathbf{S}$ and $\mathbf{S}^*$ itself is a cover of U. To distinguish the sets S_i and the set U, we call U the *ground set* and S_i the *covering (sub)sets.*

Now we are ready to define the SET COVER problem and its variations.

SET COVER
Instance: Finite set U, a cover $\mathbf{S} = \{S_1, \ldots, S_n\}$ of U.
Goal: Find a subcover $\mathbf{S}^* \subseteq \mathbf{S}$ of U of minimum cardinality.

We denote by m the number of elements of U and by OPT the size of an optimum subcover $\mathbf{S}^*$.

Many authors consider the following restriction of the SET COVER problem, where the covering sets have at most d elements.

d-SET COVER
Instance: Finite set U, a cover $\mathbf{S} = \{S_1, \ldots, S_n\}$ of U, such that $\lvert S_i \rvert \leq d$ for all $i = 1, \ldots, n$.
Goal: Find a subcover $\mathbf{S}^* \subseteq \mathbf{S}$ of U of minimum cardinality.

Throughout this chapter, we will consider the following version of the greedy algorithm for approximating SET COVER.

Algorithm "Greedy"

Input: A cover $\mathbf{S} = \{S_1, \ldots, S_n\}$ of a finite set U.

Output: A subcover $\hat{\mathbf{S}} = \{\hat{S}_1, \ldots, \hat{S}_k\} = \{S_{i_1}, \ldots, S_{i_k}\} \subseteq \mathbf{S}$ of U.

begin
    $i \leftarrow 0$;
    while $U \neq \emptyset$ **do**
        $max \leftarrow 0$;
        $i \leftarrow i + 1$;
        for $j \leftarrow 1$ **to** n **do**
            if $S_j \neq \emptyset$ **and** $|S_j| > max$ **then** $\{max \leftarrow |S_j|;\ i^* \leftarrow j;\}$
        $U \leftarrow U \setminus S_{i^*}$;
        $\hat{S}_i \leftarrow$ the original set S_{i^*};
        for $j \leftarrow 1$ **to** n **do** $S_j \leftarrow S_j \setminus S_{i^*}$;
    $APP \leftarrow i$;
    return $\hat{\mathbf{S}}$;
end

Informally, this algorithm at each step simply chooses the covering set with the maximum number of elements left, deletes these elements from the remaining covering sets, and repeats this process until the ground set U is covered. In case of a tie, the set with smaller subscript is chosen. We denote by APP the size of the subcover output by this algorithm.

In the rest of this thesis, we will denote by $H(d)$ the d^{th} harmonic number (sometimes denoted by $\mathcal{H}_d$), that is $H(d) = 1 + 1/2 + \cdots + 1/d$. It is easy to see from the graph of function $y = 1/x$ that

$$\ln d + \frac{1}{2} + \frac{1}{2d} < H(d) < \ln d + \frac{5}{8} + \frac{1}{2d},$$

thus

$$H(d) = \ln d + \Theta(1). \tag{3.1}$$

3.1.4 Related Results

Before we proceed any further, let us first define the decision version of the Set Cover problem.

Set Cover$_D$—the decision version of Set Cover
Instance: Finite set U, a cover $\mathbf{S} = \{S_1, \ldots, S_n\}$ of U, a number K.
Question: Is there a subcover $\mathbf{S}^* \subseteq \mathbf{S}$ of U such that $\lvert\mathbf{S}^*\rvert \leq K$?

Theorem 3.1 (Karp, [56]) Set Cover$_D$ *is NP-complete.*

Proof: The proof of this theorem is based on the seminal theorem of Cook [25] who proved that the Satisfiability problem is NP-complete; and on the now standard technique of proving NP-completeness pioneered by Karp, where the facts that a given problem Π is in NP and that some of the known NP-complete problems can be reduced (in polynomial time) to Π are sufficient to show that Π is NP-complete—see Theorem 2.3 and Subsection 2.2.5 for details.

Clearly Set Cover$_D$ is in NP since any solution (subcover) $\mathbf{S}^*$ can be verified in time $O(mn)$ to have at most K sets and to cover the entire ground set U. Karp proved that Satisfiability reduces to Clique, Clique reduces to Vertex Cover, and Vertex Cover almost trivially reduces to Set Cover—see Proposition 2.2 and [56] for details. A different reduction is used by Garey and Johnson in [37]. □

Let us now present the classical approximation results for Set Cover.

Theorem 3.2 (Johnson, [55] and Lovász, [64]) *The greedy algorithm for approximating* d-Set Cover *outputs a subcover of size* APP *satisfying*

$$\frac{APP}{OPT} \leq H(d).$$

Proof: We will present a modification of the proof given by Lovász [64]. Let t_j denote the number of covering sets from the greedy subcover that at the time of their selection contained exactly j so far uncovered elements of U. Thus

$$t_d + t_{d-1} + \cdots t_2 + t_1 = APP$$

and

$$dt_d + (d-1)t_{d-1} + \cdots 2t_2 + 1t_1 = m.$$

Now, at any step of the greedy algorithm, the remaining elements can be covered by at most OPT subsets. In particular, after performing $t_d + \cdots + t_{j+1}$ steps, there are $jt_j + (j-1)t_{j-1} + \cdots + 1t_1$ elements left uncovered, and any set now contains at most j of these elements. Thus for any $j = 1, \ldots, d$, we have

$$jt_j + (j-1)t_{j-1} + \cdots + 1t_1 \leq OPT \cdot j. \tag{3.2}$$

For each $j = 1, \ldots, d-1$ divide each inequality (3.2) by $j(j+1)$ to obtain

$$\frac{jt_j}{j(j+1)} + \frac{(j-1)t_{j-1}}{j(j+1)} + \cdots + \frac{t_1}{j(j+1)} \leq \frac{OPT}{j+1} \tag{3.3}$$

and for $j = d$, divide (3.2) by d to get

$$\frac{dt_d}{d} + \frac{(d-1)t_{d-1}}{d} + \cdots + \frac{t_1}{d} \leq OPT. \tag{3.4}$$

Let us now add inequalities (3.3) for $j = 1, \ldots, d-1$ and inequality (3.4) and collect like terms. The coefficient in front of each t_j is

$$j\left(\frac{1}{j(j+1)} + \frac{1}{(j+1)(j+2)} + \frac{1}{(j+2)(j+3)} + \cdots + \frac{1}{(d-1)d} + \frac{1}{d}\right). \tag{3.5}$$

This can be rewritten (using partial fractions) as

$$j\left(\frac{1}{j} - \frac{1}{j+1} + \frac{1}{j+1} - \frac{1}{j+2} + \cdots + \frac{1}{d-1} - \frac{1}{d} + \frac{1}{d}\right) = j\frac{1}{j} = 1. \tag{3.6}$$

Thus we have that

$$t_1 + t_2 + \cdots + t_d \leq \frac{OPT}{2} + \frac{OPT}{3} + \cdots + \frac{OPT}{d} + OPT = H(d) \cdot OPT \tag{3.7}$$

which was to be proved. □

The following theorem (modified from [55]) shows that the above bound on the performance of the greedy algorithm for approximating d-Set Cover is tight.

Theorem 3.3 (Johnson, [55]) *For any $d \geq 1$ there is an instance of d-*Set Cover *for which the greedy algorithm outputs a subcover of size APP satisfying*

$$\frac{APP}{OPT} = H(d).$$

Proof: If $d = 1$ the claim is trivial. Given an integer $d \geq 2$, let U be a set of $d\,d!$ elements. Divide U into d segments, each having $d!$ elements, and construct a cover $\mathbf{S}$ of U in the following way.

(a) Divide segment 1 into $d!/d = (d-1)!$ disjoint sets of d elements. Divide segment 2 into $d!/(d-1)$ disjoint sets, each containing $d-1$ elements. In general, divide segment j into $d!/(d-j+1)$ sets, each containing exactly $d-j+1$ elements. This gives us a cover $\hat{\mathbf{S}}$ of U consisting of

$$\frac{d!}{d} + \frac{d!}{d-1} + \cdots + \frac{d!}{1} = d!\,H(d)$$

disjoint sets.

(b) Divide U into $d!$ disjoint sets of size d, each containing exactly one element from each segment. This gives us a cover $\mathbf{S}^*$ of U consisting of $d!$ disjoint sets.

Set $\mathbf{S} = \hat{\mathbf{S}} \cup \mathbf{S}^*$. Clearly, $\mathbf{S}^*$ is the optimum subcover of $\mathbf{S}$ therefore $OPT = d!$.

Now assume that the greedy algorithm in case of a tie selects a set from $\hat{\mathbf{S}}$. Thus in the first $(d-1)!$ steps, the greedy algorithm would choose the sets of size d from $\hat{\mathbf{S}}$. After these have been selected, all points of segment 1 are covered, hence no other set has size bigger than $d-1$ and the greedy algorithm would continue selecting the $d!/(d-1)$ sets from the second segment all belonging to $\hat{\mathbf{S}}$. This would again leave no set of size bigger than $d-2$, thus the sets of size $d-2$ from the third segment would be selected, etc. It is now clear that the greedy algorithm would select the subcover $\hat{\mathbf{S}}$ as its solution.

Therefore $APP = d!\,H(d) = OPT \cdot H(d)$ and that was to be shown. $\square$

Note: Notice that in the above example we could make U smaller, namely we could take m/d to be the least common multiple of all numbers between 1 and d. However, it is clear that the harmonic bound can be made tight only when the ground set is much larger than d.

The result of Theorem 3.2 was later generalized by Chvátal [24]. He proved that the weighted version of d-SET COVER can be approximated within $H(d)$ using a modified greedy algorithm—see Chapter 4 for details.

Other, less efficient algorithms for approximating the d-SET COVER were able to slightly improve the harmonic bound. Halldórsson [45] showed that an algorithm which uses the greedy method until the remaining covering sets have size at most 3 followed by local improvement strategy for approximating 3-SET COVER has a performance bound of about $H(d) - 0.43$. Most recently, Duh and Fürer [28] used an algorithm which combines greedy method (until all covering sets have at most 6 elements) with their semi-local improvements technique to obtain a performance bound of $H(d) - 1/2$ for $d \geq 4$.

Let us now present previously known results about approximating (general) SET COVER. In view of Theorem 3.2, the following bound is almost trivial, since clearly $d \leq m$.

Corollary 3.1 (Johnson, [55]) *Given an instance of the* SET COVER *problem, the cover output by the greedy algorithm satisfies*

$$\frac{APP}{OPT} \leq H(m).$$

Notice that the harmonic bound is no longer tight. In fact it can be readily improved to at least $H(m-1)$ since if there is a covering set of size m than the greedy algorithm would choose it, thus returning the optimum subcover. However, even restricting ourselves to say covers having sets of size at most $m/2$ would guarantee bound $H(m/2)$ (in the case of the greedy algorithm) which is a constant improvement of about $\ln 2 \approx 0.69$ over $H(m)$. Thus any non-constant improvements of the harmonic bound are highly non-trivial.

The following theorem shows that the harmonic bound for approximating the general SET COVER by the greedy algorithm is asymptotically tight.

Theorem 3.4 (Johnson, [55]) *For any $m = 3 \cdot 2^k$, there is an instance $I = (U, \mathbf{S})$ of* Set Cover *with $|U| = m$, for which the value APP of the approximate solution output by the greedy algorithm given above satisfies*

$$\frac{APP}{OPT} = \Theta(\ln m).$$

Proof: Let U consist of $m = 3 \cdot 2^k$ points. Divide U into 3 segments of equal size. Now define the cover $\mathbf{S}$ of U to be $\mathbf{S} = \hat{\mathbf{S}} \cup \mathbf{S}^*$ where $\hat{\mathbf{S}}$ and $\mathbf{S}^*$ consist of the following covering sets:

(a) $k+1$ disjoint sets, each containing the same number of points from each segment. The first set contains $3 \cdot 2^{k-1}$ points, next set contains $3 \cdot 2^{k-2}$ points, ..., the second to last set contains $3 \cdot 2^0 = 3$ points, and last set contains the three remaining points. These sets will form the greedy subcover $\hat{\mathbf{S}}$.

(b) the 3 disjoint segments, each containing exactly $2^k - 1$ elements—this is the optimal subcover $\mathbf{S}^*$.

Thus $OPT = 3$, $APP = k+1$. Therefore

$$\frac{APP}{OPT} = \frac{k+1}{3} = \frac{1}{3}(\log_2 \frac{m}{3} + 1) \approx 0.48 \ln m.$$

□

The above construction can be improved by considering a set U of size $2 \cdot 2^k = 2^{k+1}$. This leads to $OPT = 2$, $APP = k+1$, and $APP/OPT \approx 0.72 \ln m$. Thus for *some* m there are instances of Set Cover with $|U| = m$ for which the performance of the greedy algorithm is as bad as $0.72 \ln m$. Notice that this lower bound is valid for only *some* m and that the difference between upper and lower bounds is approximately $0.28 \ln m = \Theta(\ln m)$ and hence grows without bounds as m increases.

Johnson is one of only a few authors who even mention the generalization of results for d-Set Cover to the case of Set Cover with no restrictions on the size of covering sets. The reason for this is most likely the fact that the observation $d \leq m$ is rather trivial and the bound of $H(m)$ (or any other bound where d would get simply replaced by m) does not give us anything really interesting.

It is now clear that in order to improve the harmonic bound for the (general) SET COVER problem by more than a constant, it is not enough to simply generalize the results obtained for d-SET COVER and one has to attack the problem in its full generality, i.e. assuming no restrictions on the sizes of covering sets. In the rest of this section we will present a result of Srinivasan obtained by such a general approach.

Before we proceed any further, let us introduce some concepts related to the use of linear and integer programming for approximating SET COVER. Given an instance of SET COVER—that is, a ground set $U = \{t_1, t_2, \ldots, t_m\}$ and a cover $\mathbf{S} = \{S_1, S_2, \ldots, S_n\}$—we denote by $A = (a_{ij})$ the m by n incidence matrix of the covering sets; that is,

$$a_{ij} = \begin{cases} 1 & \text{if } t_i \in S_j, \\ 0 & \text{otherwise.} \end{cases}$$

For any subcollection $\hat{\mathbf{S}}$ of $\mathbf{S}$, set $y_j = 1$ if the j^{th} covering set is in $\hat{\mathbf{S}}$ and $y_j = 0$ otherwise. Thus $\mathbf{y} = (y_1, \ldots, y_n)$ is the incidence vector of subcollection $\hat{\mathbf{S}}$. One can formulate the SET COVER problem as the following integer program.

Problem ISC:

$$\begin{aligned} OPT \; = \; \text{minimize} \quad & \sum_{j=1}^{n} y_j, \\ \text{subject to} \quad (a) \quad & \sum_{j=1}^{n} a_{ij} y_j \geq 1, \quad \text{for all } i = 1, \ldots, m, \\ (b) \quad & y_j \in \{0, 1\}, \qquad \text{for all } j = 1, \ldots, n. \end{aligned} \tag{3.8}$$

Clearly this is an equivalent formulation of the SET COVER problem—minimizing $\sum y_j$ is equivalent to minimizing the number of sets in the subcover, condition (a) guarantees that all the elements of U are covered, and condition (b) simply assures that a covering set is either chosen completely or not at all (as opposed to selecting just a fraction of a set—see below).

A now standard approach for approximating the optimum solution to an integer program is to relax the integrality condition (b) and allow y_j to vary among all (rational) numbers between 0 and 1. This leads to a linear program which can be solved in time depending polynomially on the size of the problem, using, for example, Karmarkar's algorithm or ellipsoid method.

Problem LSC:

$$
\begin{aligned}
OPT^* \;=\; \text{minimize} \quad & \sum_{j=1}^{n} y_j, \\
\text{subject to} \quad & \text{(a)} \quad \sum_{j=1}^{n} a_{ij} y_j \geq 1, \quad \text{for all } i = 1, \ldots, m, \\
& \text{(b)} \quad 0 \leq y_j \leq 1, \qquad \text{for all } j = 1, \ldots, n.
\end{aligned}
\tag{3.9}
$$

One can then use the optimal (fractional) solution to the LP-relaxation to construct an approximate solution to the original integer program. It should be noted however that such an approach is more of a theoretical value, since solving the linear program directly is usually quite slow. We will denote the optimal solution to Problem LSC by $(\mathbf{y}^*; OPT^*) = (y_1^*, \ldots, y_n^*; OPT^*)$. Clearly, $OPT^* \leq OPT$.

A relatively new technique for obtaining an approximate solution from the fractional solution is *randomized rounding*—see [70, 76] for details. Here one randomly rounds the fractions y_j^* of the optimal solution to either 0 or 1 with probability depending on the value of y_j^*. The probabilities have to be chosen in such a way that the output of this randomized algorithm has a non-zero probability of being a feasible solution to the original integer program with the value of the objective function not too far from the optimal fractional (hence integer) solution. This randomized algorithm can be turned into a deterministic algorithm by a derandomization procedure using a so-called *pessimistic estimator* which is simply an upper bound on certain conditional probabilities. The interested reader can find out more about these techniques in [70, 76].

Srinivasan [92] analyzed a deterministic algorithm for approximating the SET COVER problem based on randomized rounding combined with derandomization. He proved the following:

Theorem 3.5 (Srinivasan, [92]) *The subcover output by Srinivasan's algorithm has size APP_r which satisfies*

$$APP_r \leq OPT^* \left(\ln(\frac{m}{OPT^*}) + O(\ln\ln(\frac{m}{OPT^*})) + O(1) \right). \tag{3.10}$$

This bound clearly improves on the harmonic bound for (asymptotically) large values of OPT^*, however, for small values of OPT^*, this bound is even worse than $H(m)$.

3.1.5 Overview

Previous analyses of the greedy algorithm generally assumed some knowledge of the size m of the ground set U and the size OPT of the minimum cover, and then used various techniques to obtain bounds on APP, the size of the subcover output by the greedy algorithm. Our approach is different. We start with numbers OPT and APP, and obtain bounds on m. The key to our approach is the introduction of "greedy numbers" $N(k,l)$, which turn out to satisfy a simple recursion relation and which we prove have the following property: $N(k,l)$ is the size of the smallest set U for which it is possible to have a cover of U with $OPT = l$ and $APP = k$. This allows us to abstract from the Set Cover problem and analyze instead the function $N(k,l)$ which leads to the establishment of both upper and lower bounds on the performance ratio. In fact we show that for any set U with $|U| = m \geq 2$ and for any cover $\mathbf{S}$ of U

$$\frac{APP}{OPT} < \ln m - \ln\ln m + 0.78, \tag{3.11}$$

and that for all $m \geq 2$ there is a cover $\mathbf{S}$ of some set U with $|U| = m$ such that

$$\frac{APP}{OPT} > \ln m - \ln\ln m - 0.31. \tag{3.12}$$

Notice that our approach gives lower bounds in a stronger form than is customary. Namely, (3.12) holds for *all* values of m.

Generalization of our analysis to fractional covers is almost straightforward. The arguments are little more subtle, but lead, essentially, to the same upper and lower bounds. On our way to proving these bounds, we show that

$$APP \leq \left(OPT^* - \frac{1}{2}\right)(\ln m - \ln OPT^*) + OPT^* \tag{3.13}$$

which significantly improves on Srinivasan's bound.

3.2 Performance Bounds

Let us first define the "greedy numbers" $N(k,l)$. For given $l \geq 2$, set

$$a_1 = 1$$

and

$$a_i = \lceil \frac{a_1 + \cdots + a_{i-1}}{l-1} \rceil \tag{3.14}$$

for $i = 2, 3, \ldots$. We then define the (k,l) greedy number $N(k,l)$ as

$$N(k,l) = \sum_{i=1}^{k} a_i \quad \text{for } k = 1, 2, \ldots. \tag{3.15}$$

Obviously, for any $2 \leq l \leq k$, $N(l,l) = l$ and

$$N(k+1,l) = N(k,l) + \lceil \frac{N(k,l)}{l-1} \rceil = \lceil \frac{l}{l-1} N(k,l) \rceil. \tag{3.16}$$

Hence we can recursively generate $N(k,l)$ for any $k \geq l \geq 2$.

Consider now the Set Cover problem. The case $OPT = 1$, that is the case where U can be covered by one set, is not interesting, since the greedy algorithm will also output a single set, hence $APP = OPT$. Therefore in what follows, we will consider only covers for which $OPT \geq 2$.

Set $m = |U|$. At each step, i, of the greedy algorithm, we delete q_i elements. We have $k = APP$ steps, hence

$$q_1 + q_2 + \cdots + q_k = m. \tag{3.17}$$

We know that U can be covered by $l = OPT$ sets. By averaging argument, at least one of the sets in the minimum cover contains at least $\lceil \frac{m}{l} \rceil$ elements. Hence

$$q_1 \geq \lceil \frac{m}{l} \rceil.$$

Similarly,

$$q_2 \geq \lceil \frac{m - q_1}{l} \rceil$$

and, in general,

$$q_i \geq \lceil \frac{m - (q_1 + \cdots + q_{i-1})}{l} \rceil \tag{3.18}$$

for $i = 2, \ldots, k$. Using (3.17), we can rewrite (3.18) in the form

$$q_i \geq \lceil \frac{q_i + \cdots + q_k}{l} \rceil \tag{3.19}$$

which is equivalent to

$$q_i \geq \lceil \frac{q_{i+1} + \cdots + q_k}{l-1} \rceil \quad \text{for } i = 1, \ldots, k. \tag{3.20}$$

Clearly, $a_1 \leq q_k$, $a_2 \leq q_{k-1}$, …, $a_k \leq q_1$, hence

$$N(k,l) = \sum_{i=1}^{k} a_i \leq \sum_{i=1}^{k} q_i = m = |U|. \tag{3.21}$$

Therefore, if $m < N(k,l)$ and $OPT = l$, it must be that $APP < k$. The following example shows that the opposite is also true, namely, for any $k \geq l \geq 2$ and any $m \geq N(k,l)$ one can easily construct a cover $\mathbf{S}$ of some set U such that $|U| = m$, $OPT = l$, and $APP = k$.

Example 3.1 Let $k \geq l \geq 2$, and $m \geq N(k,l)$ be given. Define $U = \{1, 2, \ldots, m\}$. Since $m \geq N(k,l)$ there are positive integers $q_1 \geq q_2 \geq \cdots \geq q_k$ satisfying (3.20) and (3.17). Define a cover $\mathbf{S}$ of U in the following way.

(i) For $i = 1, \ldots, k$ set

$$S_i = \{q_1 + \cdots + q_{i-1} + 1, q_1 + \cdots + q_{i-1} + 2, \ldots, q_1 + \cdots + q_i\},$$

Thus each S_i contains exactly q_i elements, the sets are disjoint, and $\bigcup_{i=1}^{k} S_i = U$.

(ii) Set $d = \lceil m/l \rceil$. Then one can write $m = l_1 d + l_2(d-1)$ for some l_1, l_2 such that $l_1 + l_2 = l$. Define

$$S_{k+i} = \{i, i+l, \ldots, i + l(d-1)\} \quad \text{for } i = 1, \ldots, l_1$$

and

$$S_{k+i} = \{i, i+l, \ldots, i+l(d-2)\} \quad \text{for } i = l_1+1, \ldots, l.$$

Then, clearly, the first l_1 sets contain d elements each, the next l_2 sets contain $d-1$ elements each, the sets are disjoint, and $\bigcup_{i=1}^{l} S_{k+i} = U$. Thus $OPT = l$.

Figure 3.1 shows this construction for $m = 18$, $d = 3$, $l = 6$, $k = 11$.

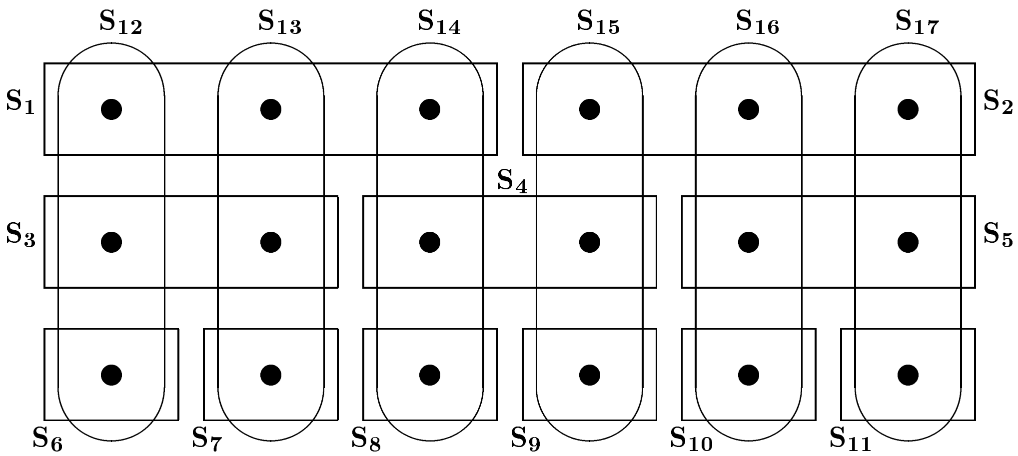


Figure 3.1: Instance of Set Cover where $APP = k = 11$, $OPT = l = 6$, and $m = N(k,l) = 18$.

We claim that the greedy algorithm outputs a cover $\mathbf{S}^* = \{S_1, \ldots, S_k\}$. Indeed, $q_1 \geq d$, hence in the first iteration S_1 is chosen. Assume that after the i-th step, the sets $S_1, \ldots, S_i$ have been chosen, leaving $q_{i+1} + \cdots + q_k$ elements to cover. Set $r = \max\{|S_{k+j}^{(i)}| : j = 1, \ldots, l\}$, where $S_j^{(i)}$ denotes the set S_j after the i-th iteration of the greedy algorithm, that is, after deleting the elements belonging to already chosen sets. Because of the construction of the sets in the cover, we have $r - 1 \leq |S_{k+j}^{(i)}| \leq r$ for all $j = 1, \ldots, l$. Hence $q_{i+1} + \cdots + q_k = \sum_j |S_{k+j}^{(i)}| > (r-1)l$ and thus $q_{i+1} \geq r$ by inequality (3.19). As a result, the greedy algorithm will choose the set S_{i+1} in its $(i+1)$-st step. Thus $APP = k$. $\square$

It is now clear that, for fixed $OPT = l$, $m < N(k,l)$ implies $APP < k$, and if $m \geq N(k,l)$, there are covers for which $OPT = l$ and $APP = k$. This proves the following Lemma:

Lemma 3.1 *For any set U with $|U| \geq 2$ and for any cover $\mathbf{S}$ of U,*

$$\frac{APP}{OPT} \leq \max\{\frac{k}{l} \mid N(k,l) \leq |U|\}. \tag{3.22}$$

Moreover, there are covers for which the equality is attained.

Lemma 3.1 establishes a tight bound on the quotient APP/OPT. Unfortunately, by itself, it is of little practical use since we know almost nothing about the numbers $N(k,l)$. Using (3.16), one can evaluate the bound for the quotient APP/OPT for small m, but it does not say much about asymptotic behavior. Let us now establish some lower and upper bounds on $N(k,l)$. This will enable us to find upper and lower bounds on APP/OPT.

Using (3.16), one can easily show that

$$N(k,l) \geq \left(\frac{l}{l-1}\right)^{k-l} N(l,l) = \left(\frac{l}{l-1}\right)^{k-l} l \geq e^{\frac{k-l}{l}} l \tag{3.23}$$

which gives $k \leq l(\ln N(k,l) - \ln l + 1)$, hence, by Lemma 3.1,

$$APP \leq OPT(\ln m - \ln OPT + 1). \tag{3.24}$$

But we can actually obtain a much better bound. The proof is in Section 3.5.

Lemma 3.2 *For any set U with $m = |U| \geq 2$ and any cover $\mathbf{S}$ of U,*

$$APP \leq (OPT - 1/2)(\ln m - \ln OPT) + OPT. \tag{3.25}$$

The fact that we can multiply by $(OPT - 1/2)$ instead of by OPT makes our analysis of the greedy algorithm stronger than previous results. This improvement together with Lemma 3.1 is crucial for establishing the following upper bound on APP/OPT. The proof can be found in Section 3.5.

Theorem 3.6 *For any set U with $|U| = m \geq 2$ and for any cover $\mathbf{S}$ of U, the greedy algorithm outputs a subcover of size APP satisfying*

$$\frac{APP}{OPT} \leq \ln m - \ln\ln m + 3 + \ln\ln 32 - \ln 32 \approx \ln m - \ln\ln m + 0.78. \tag{3.26}$$

The following lower bound on the performance of the greedy algorithm nicely complements the above result.

Theorem 3.7 *For every $m \geq 2$, there exist a set U with $|U| = m$ and a cover $\mathbf{S}$ of U, such that the greedy algorithm outputs a subcover $\hat{\mathbf{S}}$ with cost APP satisfying*

$$\frac{APP}{OPT} > \ln m - \ln\ln m - 1 + \ln 2 \approx \ln m - \ln\ln m - 0.31. \tag{3.27}$$

This Theorem shows that the upper bound (3.26) is tight (up to a constant). This makes the analysis of the performance of the greedy algorithm for approximating the Set Cover problem essentially complete.

Note: For any real number $u \geq 2$ set $M(u) = \max\{k/l \mid N(k,l) \leq u\}$, that is for $u = m$, $M(u) =$"the worst case for APP/OPT". Figure 3.2 shows the graphs of $M(u)$ (in the middle), and the functions $\ln u - \ln\ln u - 1 + \ln 2$ (lower bound) and $\ln u - \ln\ln u + 3 + \ln\ln 32 - \ln 32$ (upper bound), for small and large values of u. This shows that some improvements of the constants in (3.26) and particularly in (3.27) may still be possible.

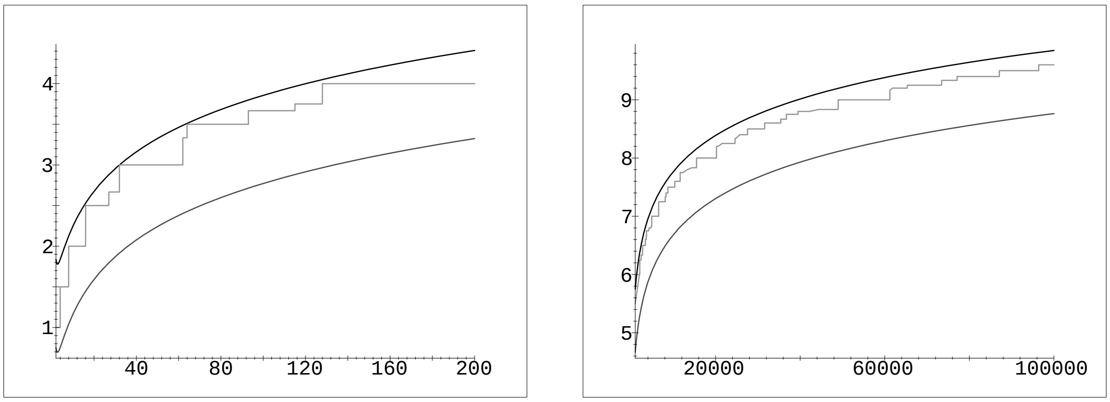


Figure 3.2: Graphs of $M(u)$ and the lower and upper bounds

3.3 Fractional Covers

Our bounds in the previous section are incomparable with Srinivasan's bound (3.10). In this section we further improve our estimates and show that the performance guarantee of the greedy algorithm is better than that of the randomized rounding technique.

Following [64], we define a fractional cover $\mathbf{T}$ of U to be a system of weights $\mathbf{T} = \{t_1, \ldots, t_n\}$ such that for all points $x \in U$ we have

$$\sum_{\{j \mid x \in S_j\}} t_j \geq 1.$$

Denote by $c^*(\mathbf{T})$ the "cost" of the fractional cover $\mathbf{T}$, i.e.

$$c^*(\mathbf{T}) = \sum_{j=1}^{n} t_j$$

and let

$$OPT^* = \min_{\mathbf{T}} c^*(\mathbf{T}).$$

This formulation is clearly equivalent to Problem LSC, the LP relaxation of Set Cover.

Let us follow the steps in Section 3.2. Set $l^* = OPT^*$. A simple argument shows that $OPT^* = 1$ implies $OPT = 1$, hence by considering only those covers for which $OPT = l \geq 2$, we actually consider covers for which $OPT^* = l^* > 1$. We define generalized greedy numbers as follows. Set $a_1^* = 1$ and

$$a_i^* = \lceil \frac{a_1^* + \cdots + a_{i-i}^*}{l^* - 1} \rceil.$$

Define

$$N^*(k, l^*) = \sum_{i=1}^{k} a_i^* \quad \text{for k=1,2,}\ldots \tag{3.28}$$

We have again that

$$N^*(k+1, l^*) = N^*(k, l^*) + \lceil \frac{N^*(k, l^*)}{l^* - 1} \rceil = \lceil \frac{l^*}{l^* - 1} N^*(k, l^*) \rceil, \tag{3.29}$$

and, with a small adjustment, $N^*(\lfloor l^* \rfloor, l^*) = \lfloor l^* \rfloor$ for any $l^* > 1$.

Now,

$$q_1 l^* = q_1 \sum_{j=1}^{n} t_j \geq \sum_{j=1}^{n} |S_j| t_j = \sum_{j=1}^{n} (\sum_{x \in S_j} t_j) = \sum_{x \in U} (\sum_{\{j \mid x \in S_j\}} t_j) \geq m,$$

hence $q_1 \geq \lceil \frac{m}{l^*} \rceil$. Similarly, $q_i \geq \lceil \frac{m-(q_1+\cdots+q_{i-1})}{l^*} \rceil$ and thus, as before, $\sum_{i=1}^{k} a_i^* \leq \sum_{i=1}^{k} q_i$. From the discussion in the previous section, we have again that $m < N^*(k, OPT^*)$ implies $APP < k$ hence the following counterpart of Lemma 3.1 holds.

Lemma 3.3 *For any set U with $|U| = m \geq 2$ and for any cover $\mathbf{S}$ of U,*

$$APP \leq \max\{k \mid N(k, OPT^*) \leq m\}.$$

Careful analysis shows that

$$N^*(k, l^*) \geq \left(\frac{l^*}{l^*-1}\right)^{k-\lfloor l^* \rfloor} N^*(\lfloor l^* \rfloor, l^*) \geq \left(\frac{l^*}{l^*-1}\right)^{k-\lfloor l^* \rfloor} \lfloor l^* \rfloor \geq e^{\frac{k-l^*}{l^*}} l^* \qquad (3.30)$$

for $k \geq l^*$, hence

$$\ln N^*(k, l^*) \geq k/l^* + \ln l^* - 1. \qquad (3.31)$$

Thus Lemma 3.3 gives the following.

Proposition 3.1 *For any set U with $|U| = m \geq 1$ and for any cover $\mathbf{S}$ of U, the cover output by the greedy algorithm satisfies*

$$APP \leq OPT^*(\ln m - \ln OPT^* + 1). \qquad (3.32)$$

Proposition 3.1 already improves on Srinivasan's bound (3.10). Proceeding as in Section 3.2, we can further improve (3.32) and obtain the following analogy of Lemma 3.2. The proof is in Section 3.5.

Theorem 3.8 *For any set U with $|U| = m \geq 1$ and for any cover $\mathbf{S}$ of U, the cover output by the greedy algorithm satisfies*

$$APP \leq \left(OPT^* - \frac{1}{2}\right)(\ln m - \ln OPT^*) + OPT^*. \qquad (3.33)$$

Theorem 3.8 shows that the performance guarantee for the greedy algorithm is substantially better than the performance guarantee (3.10) for Srinivasan's algorithm. This immediately gives an improved bound on the integrality gap

$$\frac{OPT}{OPT^*} \le \left(1 - \frac{1}{2OPT^*}\right)(\ln m - \ln OPT^*) + 1.$$

Moreover, inequality (3.33) is of the same form as the bound in Lemma 3.2, only OPT is replaced by OPT^*. Thus a simple repetition of the steps in the proof of Theorem 3.6 proves that

$$\frac{APP}{OPT^*} \le \ln m - \ln\ln m + \ln 2 + \epsilon$$

for all m large enough.

It is obvious that for any cover, $OPT^* \le OPT$, hence for every $m \ge 2$, there are covers for which

$$\frac{APP}{OPT^*} > \ln m - \ln\ln m + \ln 2 - 1.$$

3.4 Generalization for the Partial Cover Problem

Let us now briefly mention an extension of our results to the Partial Cover problem which further generalizes Set Cover. Given p, $0 < p \le 1$, the goal here is to find a minimum *partial* subcover, i.e. a collection of covering sets such that at least a p-fraction of points in the ground set are covered—see the next chapter or [86] for a detail treatment of Partial Cover. The greedy algorithm works exactly as in the "complete" Set Cover case (at each step choose the currently largest set until a p-fraction of points are covered), hence only a slight modification of the approach of Section 3.2 would prove the following:

Theorem 3.9 *Let U be a finite set of size $|U| = m$, $\mathbf{S}$ be a cover of U, and $0 < p \le 1$ be such that $\lceil pm \rceil = u \ge 2$. Denote by OPT the size of a minimum p-partial cover of U. The greedy algorithm outputs a partial cover of size APP satisfying*

$$\frac{APP}{OPT} \le \ln u - \ln\ln u + 3 + \ln\ln 32 - \ln 32 \approx \ln u - \ln\ln u + 0.78. \tag{3.34}$$

Moreover, for any $u \geq 2$ there is a cover $\mathbf{S}$ *of* U*, such that the greedy algorithm outputs a p-partial cover* $\mathbf{S}^*$ *with cost* APP *satisfying*

$$\frac{APP}{OPT} > \ln u - \ln\ln u - 1 + \ln 2 \approx \ln u - \ln\ln u - 0.31. \tag{3.35}$$

One can similarly generalize the results of Section 3.3. The only difference would be the definition of partial fractional cover, where we want the condition $\sum_{\{j \mid x \in S_j\}} t_j \geq 1$ to hold for at least $u = \lceil pm \rceil$ points.

3.5 Proofs

Proof [Lemma 3.2]**:** Since the function $y = 1/x$ is convex (=concave up), we have

$$\ln(a+b) - \ln a > \frac{b}{a + \frac{b}{2}} = \frac{2b}{2a+b}$$

for any $a, b > 0$. Hence, by the first part of inequality (3.23),

$$\ln N(k,l) \geq \ln l + (k-l)(\ln l - \ln(l-1)) > \ln l + (k-l)\frac{2}{2l-1}. \tag{3.36}$$

Rearranging (3.36) immediately gives

$$k \leq (l - 1/2)\left(\ln N(k,l) - \ln l\right) + l \tag{3.37}$$

for any $k \geq l \geq 2$. The discussion preceding Lemma 3.1 easily concludes the proof. $\square$

Proof [Theorem 3.6]**:** We can rewrite (3.37) in the form

$$\frac{k}{l} \leq (1 - \frac{1}{2l})(\ln N(k,l) - \ln l) + 1,$$

hence for any real $u \geq N(k,l)$

$$\ln u - \frac{k}{l} \geq \frac{\ln u - \ln l}{2l} + \ln l - 1. \tag{3.38}$$

That motivates the rest of the proof. To simplify the reasoning, allow $l \geq 2$ to be a real number. Define $g(l,u) = \dfrac{\ln u - \ln l}{2l} + \ln l - 1$ and $f(u) = \min_{2 \leq l \leq u} g(l,u)$. One can easily show, that for fixed $1 \leq u \leq 2e^3$, $g(l,u)$ is an increasing function of l, and for fixed $u > 2e^3$, $g(l,u)$ is a unimodal function with both relative and absolute minimum at $l = \hat{l}$, where $\hat{l}$ satisfies $\ln u = h(\hat{l})$ with

$$\ln u = h(\hat{l}) = \ln \hat{l} + 2\hat{l} - 1. \tag{3.39}$$

Therefore

$$f(u) = \begin{cases} g(2,u) = \frac{\ln u}{4} + \frac{3\ln 2}{4} - 1 & \text{for } 1 \leq u \leq 2e^3, \\ g(\hat{l},u) = \ln \hat{l} - \frac{1}{2\hat{l}} = \ln h^{-1}(\ln u) - \frac{1}{2h^{-1}(\ln u)} & \text{for } u > 2e^3. \end{cases}$$

Here h^{-1} is the inverse of h. Since h is increasing, so is h^{-1}. Let us establish a lower bound on $f(u)$ for large u. Clearly, for any small $\omega > 0$, $\ln u = 2\hat{l} + \ln \hat{l} - 1 \leq (2+\omega)\hat{l}$, for $\hat{l}$ big enough. Hence $\hat{l} \geq \frac{\ln u}{2+\omega}$, and $\ln \hat{l} \geq \ln\ln u - \ln(2+\omega) \geq \ln\ln u - \ln 2 - \omega/2$. Also, $1/(2\hat{l})$ can be made arbitrarily small; hence for any $\epsilon > 0$ there exists u_o such that $f(u) \geq \ln\ln u - \ln 2 - \epsilon$ for all $u \geq u_o$. Therefore for any u large enough, and for any k and l such that $N(k,l) \leq u$, the inequality (3.38) implies

$$\ln u - \frac{k}{l} \geq g(l,u) \geq f(u) \geq \ln\ln u - \ln 2 - \epsilon. \tag{3.40}$$

Hence, by Lemma 3.1,

$$\frac{APP}{OPT} \leq \ln m - \ln\ln m + \ln 2 + \epsilon < \ln m - \ln\ln m + 0.69 + \epsilon$$

for m large enough. By actually checking "small" values of m and $\hat{l}$ ($\hat{l} \leq 17$, $m \leq 4 \cdot 10^{15}$), one can show that

$$\frac{APP}{OPT} \leq \ln m - \ln\ln m + 3 + \ln\ln 32 - \ln 32 < \ln m - \ln\ln m + 0.78$$

for all $m \geq 2$. $\square$

Proof [Theorem 3.7]**:** In order to prove the theorem, we will need the following lemma.

Lemma 3.4 *For any $k \geq l \geq 2$*

$$\frac{k}{l} \geq (1 - \frac{1}{2l-1})(\ln N(k,l) - \ln l) + \frac{2}{l} + \frac{l-2}{l}\left(\frac{l-1}{l}\right)^{k-l}. \tag{3.41}$$

Proof: Obviously,

$$N(i+1,l) \le N(i,l)\frac{l}{l-1} + \frac{l-2}{l-1} = N(i,l)\left(1 + \frac{1}{l-1} + \frac{l-2}{(l-1)N(i,l)}\right).$$

Using the fact that $\ln(1+a) < \frac{a}{2}(1+\frac{1}{1+a})$, for any $a > 0$ and inequality (3.23), we have

$$\begin{aligned}
\ln N(i+1,l) - \ln N(i,l) &\le \ln\left(1 + \frac{1}{l-1} + \frac{l-2}{(l-1)N(i,l)}\right) \\
&< \frac{1}{2}\left(\frac{1}{l-1} + \frac{l-2}{(l-1)N(i,l)}\right)\left(1 + \frac{1}{\frac{l}{l-1} + \frac{l-2}{(l-1)N(i,l)}}\right) \\
&< \frac{1}{2}\left(\frac{1}{l-1} + \frac{l-2}{(l-1)N(i,l)}\right)\left(1 + \frac{1}{\frac{l}{l-1}}\right) \\
&\le \frac{2l-1}{2l}\left[\frac{1}{l-1} + \frac{l-2}{l(l-1)}\left(\frac{l-1}{l}\right)^{i-l}\right].
\end{aligned}$$

Adding the above inequalities for $i = l, \ldots, k-1$, and using (3.23) again, we get

$$\begin{aligned}
\ln N(k,l) - \ln l &\le \frac{2l-1}{2l}\cdot\frac{k-l}{l-1} + \frac{2l-1}{2l}\cdot\frac{l-2}{l(l-1)}\cdot\frac{1-\left(\frac{l-1}{l}\right)^{k-l}}{1-\frac{l-1}{l}} \\
&\le \frac{2l-1}{2l(l-1)}\left[(k-l) + (l-2)\left(1-\left(\frac{l-1}{l}\right)^{k-l}\right)\right].
\end{aligned}$$

Now we can multiply the whole inequality by $2(l-1)/(2l-1)$, rearrange the terms and obtain (3.41). □

Proof [Theorem 3.7 - continued]**:** Let m be arbitrary. Taking advantage of the condition (3.39) define l to be the largest integer such that $2l - 1 + \ln l < \ln m$ and let u satisfy $\ln u = 2l - 1 + \ln l$. Thus we have

$$\ln u = 2l - 1 + \ln l < \ln m < 2l + 1 + \ln(l+1), \tag{3.42}$$

therefore

$$\ln m - \ln u < 2 + \ln(l+1) - \ln l < 2 + \frac{1}{l} \tag{3.43}$$

and, for $l \geq 3$ (that is, for $m \geq 446$),

$$\ln m \leq 2l + 1 + \ln(l+1) < 2(2l-1). \tag{3.44}$$

Using (3.42) we have

$$l = \frac{\ln u - \ln l + 1}{2} \leq \frac{\ln u}{2} = \frac{\ln m}{2} - \frac{\ln m - \ln u}{2}.$$

Since $\ln(b-a) \leq \ln b - a/b$ for any $0 < a < b$, we obtain

$$\ln l \leq \ln\left(\frac{\ln m}{2}\right) - \frac{\ln m - \ln u}{\ln m},$$

which by (3.44) can be reduced to

$$\ln l \leq \ln\ln m - \ln 2 - \frac{\ln m - \ln u}{2(2l-1)}. \tag{3.45}$$

Let k be such that $N(k,l) \leq m < N(k+1,l)$. Using inequality (3.41) we have

$$\ln N(k+1,l) - \frac{k+1}{l} \leq \ln l + \frac{\ln N(k+1,l) - \ln l}{2l-1} - \frac{2}{l} - \frac{l-2}{l}\left(\frac{l-1}{l}\right)^{k+1-l}.$$

Now we can add (and subtract) some terms and use the above inequalities to obtain

$$\begin{aligned}
\ln N(k+1,l) - \frac{k}{l} \quad &\leq \quad \ln l + \frac{\ln N(k+1,l) - \ln m + \ln m - \ln u + \ln u - \ln l}{2l-1} \\
&\quad -\frac{2}{l} - \frac{l-2}{l}\left(\frac{l-1}{l}\right)^{k+1-l} + \frac{1}{l} \\
&= \quad \ln l + \frac{\ln u - \ln l}{2l-1} + \frac{\ln m - \ln u}{2l-1} \\
&\quad + \frac{\ln N(k+1,l) - \ln m}{2l-1} - \frac{1}{l} - \frac{l-2}{l}\left(\frac{l-1}{l}\right)^{k+1-l} \\
&< \quad \ln\ln m - \ln 2 - \frac{\ln m - \ln u}{2(2l-1)} + \frac{2l-1}{2l-1} + \frac{\ln m - \ln u}{2l-1} \\
&\quad + \frac{\ln N(k+1,l) - \ln N(k,l)}{2l-1} - \frac{1}{l} - \frac{l-2}{l}\left(\frac{l-1}{l}\right)^{k+1-l} \\
&< \quad \ln\ln m - \ln 2 + 1 + \frac{\ln m - \ln u}{2(2l-1)}
\end{aligned}$$

$$
\begin{aligned}
& + \frac{\ln N(k+1,l) - \ln N(k,l)}{2l-1} - \frac{1}{l} - \frac{l-2}{l}\left(\frac{l-1}{l}\right)^{k+1-l} \\
< \quad & \ln\ln m - \ln 2 + 1 + \frac{2+\frac{1}{l}}{2(2l-1)} + \frac{\frac{1}{l-1} + \frac{l-2}{l(l-1)}\left(\frac{l-1}{l}\right)^{k-l}}{2l} \\
& -\frac{1}{l} - \frac{l-2}{l}\left(\frac{l-1}{l}\right)^{k+1-l} \\
= \quad & \ln\ln m - \ln 2 + 1 + \left[\frac{2+\frac{1}{l}}{2(2l-1)} + \frac{1}{2l(l-1)} - \frac{1}{l}\right] \\
& + \left[\frac{l-2}{2l(l-1)^2}\left(\frac{l-1}{l}\right)^{k+1-l} - \frac{l-2}{l}\left(\frac{l-1}{l}\right)^{k+1-l}\right] \\
< \quad & \ln\ln m - \ln 2 + 1.
\end{aligned}
$$

Thus for each $m \geq 446$ there are l and k, such that $N(k,l) < m$ and

$$
\frac{k}{l} > \ln N(k+1,l) - \ln\ln m + \ln 2 - 1 > \ln m - \ln\ln m + \ln 2 - 1. \tag{3.46}
$$

Checking small values of m, one can show that (3.46) is true for all $m \geq 2$. $\square$

Proof [Theorem 3.8]**:** In order to simplify the notation, let us omit the "*" when referring to N^* and l^*. Hence $l > 1$ is now a real number. Repeating the proof of Lemma 3.2, we get

$$
\ln N(k,l) \geq \ln\lfloor l \rfloor + \frac{2(k - \lfloor l \rfloor)}{2l-1},
$$

hence

$$
\ln N(k,l) \geq \ln l + \frac{2(k-l)}{2l-1} + \omega,
$$

where

$$
\omega = \frac{2(l - \lfloor l \rfloor)}{2l-1} + \ln\lfloor l \rfloor - \ln l.
$$

Set $\alpha = l - \lfloor l \rfloor$, that is $0 \le \alpha < 1$. Then

$$\omega = \frac{2\alpha}{2l-1} + \ln(l-\alpha) - \ln l \ge \frac{2\alpha}{2l-1} - \frac{\alpha}{l-\alpha} = \frac{\alpha(1-2\alpha)}{(2l-1)(l-\alpha)}.$$

Thus for $0 \le \alpha \le 1/2$ and any $k \ge l > 1$, we have $\omega \ge 0$, hence

$$\ln N(k,l) \ge \ln l + \frac{2(k-l)}{2l-1}. \tag{3.47}$$

If $\alpha = l - \lfloor l \rfloor \ge 1/2$, then $l \ge 3/2$, hence $N(\lceil l \rceil, l) = \lfloor l \rfloor + 2$. Therefore

$$\ln N(k,l) \ge \ln(\lfloor l \rfloor + 2) + \frac{2(k - \lfloor l \rfloor - 1)}{2l-1} = \ln l + \frac{2(k-l)}{2l-1} + \epsilon,$$

where

$$\epsilon = \frac{2(l - \lfloor l \rfloor - 1)}{2l-1} + \ln(\lfloor l \rfloor + 2) - \ln l.$$

Similarly as above,

$$\epsilon \ge \frac{2\alpha - 2}{2l-1} + \frac{2-\alpha}{l+1} = \frac{2l + 3\alpha - 4}{(2l-1)(l+1)} \ge 0,$$

hence (3.47) is valid for $1/2 \le \alpha < 1$ as well. Rearranging (3.47) completes the proof. □

3.6 Conclusion

Simple modification of our analysis to Halldórsson's "local improvements" algorithm [45] or Duh's and Fürer's "semi-local improvements" algorithm [28] shows that the performance ratio of these algorithms is also exactly $\ln m - \ln\ln m + \Theta(1)$, with slightly smaller constants than in (3.26) and (3.27). Since Duh's and Fürer's algorithm has the best worst-case performance guarantee among the known polynomial-time approximation algorithms, our analysis suggests a direction in possible further improvement of Feige's hardness result [32].

Chapter 4

Partial Cover

In this chapter, we prove that the classical bounds on the performance of the greedy algorithm for approximating SET COVER with costs are valid for PARTIAL COVER as well, thus lowering, by more than a factor of two, the previously known estimate. In order to do so, we introduce a new simple technique that might be useful for attacking other similar problems.

4.1 Introduction

A natural generalization of the SET COVER problem is to associate a positive cost with each covering set and then look for a cover of minimum total cost. Clearly, this weighted version of SET COVER contains the original problem as a special case—simply assign to each covering set a unit cost. Thus the weighted versions of both SET COVER and d-SET COVER are NP-hard and efficient approximation algorithms are of great interest.

One would hope that some modification of the greedy algorithm from previous chapter would approximate the weighted SET COVER with a reasonable performance. Chvátal [24] showed that this is indeed the case. He considered an algorithm that at each step simply chooses the covering set with the minimum cost per (remaining) element and proved that the performance ratio of his algorithm is no worse than $H(d)$, when approximating d-SET COVER, and does not exceed $H(m)$ in the general case. Thus despite the fact that approximating the weighted version of SET COVER seems more difficult than the unweighted case, the performance bounds are the same for d-SET COVER and only slightly worse for SET COVER. Notice that unlike in the unweighted case, the harmonic bound $H(m)$ on the performance of the greedy algorithm for approximating weighted SET COVER is tight—see Subsection 4.1.2 for more details.

In this chapter, we will consider a weighted version of the PARTIAL COVER problem that further generalizes SET COVER. Given $0 < p \leq 1$ and a collection of sets covering some ground set U, the goal is to find a subcollection of minimum total cost covering at least a p-fraction of the elements of U. We will also consider PARTIAL d-COVER, a restriction of general PARTIAL COVER where the covering sets have at most d elements. Both problems are clearly NP-hard and their optimum solution can again be approximated by a greedy algorithm—see [58] and Subsection 4.1.2. Even though this algorithm is a straightforward modification of the greedy algorithm for complete covers, the methods used by Chvátal, Johnson, or Lovász to establish the harmonic bounds do not readily generalize to the PARTIAL COVER context. Kearns [58, p. 69] proved that for $0 < p \leq 1$ the performance ratio of the greedy algorithm for approximating PARTIAL COVER is at most $2H(m) + 3$. This is of course much worse than the bounds for SET COVER and only shows how complicated the behavior of the "greedy" partial cover can be.

The PARTIAL COVER problem and the greedy algorithm for its approximation naturally arise in several learning algorithms. For example, Kearns in [58] used the greedy algorithm and the above upper bound to construct an Occam algorithm tolerating malicious errors for learning monomials.

4.1.1 Our Results

Using a new simple technique, we significantly improve the guarantee on the performance of the greedy algorithm for approximating both PARTIAL COVER and PARTIAL d-COVER. We prove that PARTIAL d-COVER can be approximated with performance ratio of at most $H(d)$ and that for the (general) PARTIAL COVER problem the performance ratio of the greedy algorithm is no worse than $H(\lceil pm \rceil)$. Hence we establish classical bounds for non-classical problems. Our results clearly contain, as a special case, the bounds obtained by Chvátal, Johnson, and Lovász for complete covers, and significantly improve Kearns's estimate.

4.1.2 Formal Definitions and Related Results

Let us first state the weighted versions of the Set Cover and d-Set Cover problems. As before, we let U be a finite set and $\mathbf{S} = \{S_1, S_2, \ldots, S_n\}$ be a cover of U. We identify the cover $\mathbf{S}$ with the set of indices $\mathbf{J} = \{1, 2, \ldots, n\}$ and any subset $\mathbf{S}^*$ of $\mathbf{S}$ with the corresponding set $\mathbf{J}^* \subset \mathbf{J}$. As before set $m = |U|$. Let $\mathbf{c} = \{c_1, c_2, \ldots, c_n\}$ be positive costs of the sets in $\mathbf{S}$. If $\mathbf{S}^*$ ($\mathbf{J}^*$) is a cover of U we call $c(\mathbf{S}^*) = c(\mathbf{J}^*) = \sum_{j \in \mathbf{J}^*} c_j$ the *cost* of the cover. We denote by OPT the minimum possible cost of a subcover of U.

Set Cover—weighted version
Instance: Finite set U, finite cover $\mathbf{S}$ of U, positive costs $\mathbf{c}$.
Goal: Find a cover $\mathbf{S}^* \subset \mathbf{S}$ of U of minimum total cost.

d-Set Cover—weighted version
Instance: Finite set U, finite cover $\mathbf{S}$ of U with covering sets of at most d elements, positive costs $\mathbf{c}$.
Goal: Find a cover $\mathbf{S}^* \subset \mathbf{S}$ of U of minimum total cost.

As mentioned above, these two problems can be approximated by greedy heuristic. We will use a slight modification of the algorithm given by Chvátal [24].

Algorithm "Weighted_Greedy"

Input: A cover $\mathbf{S} = \{S_1, \ldots, S_n\}$ of a finite set U and positive costs $\mathbf{c} = \{c_1, c_2, \ldots, c_n\}$ of the covering sets.

Output: A subcover $\hat{\mathbf{S}} = \{\hat{S}_1, \ldots, \hat{S}_k\} = \{S_{i_1}, \ldots, S_{i_k}\} \subset \mathbf{S}$ of U.

1. Set $\hat{\mathbf{J}} = \emptyset$.

2. If $S_j = \emptyset$ for all j, then STOP and output $\hat{\mathbf{J}}$.

3. Find $i \in \mathbf{J} \setminus \hat{\mathbf{J}}$ that minimizes the quotient $\frac{c_j}{|S_j|}$, for $j \in \mathbf{J} \setminus \hat{\mathbf{J}}$ and $S_j \neq \emptyset$. In case of a tie, take the smallest such i.

4. Add i to $\hat{\mathbf{J}}$. For each $j \in \mathbf{J} \setminus \hat{\mathbf{J}}$ set $S_j = S_j \setminus S_i$. Return to Step 2.

Step 3 of the algorithm is very intuitive, since the more elements a set has and the cheaper it is, the more likely it is included in a minimum subcover.

Theorem 4.1 (Chvátal, [24]) *Algorithm "Weighted_Greedy" approximates the d-*Set Cover *problem with performance ratio*

$$\frac{APP}{OPT} \leq H(d) \tag{4.1}$$

and the Set Cover *problem with performance ratio*

$$\frac{APP}{OPT} \leq H(m). \tag{4.2}$$

Proof: We will briefly outline here the proof of Chvátal [24]. Let $U = \{t_1, \ldots, t_m\}$ and denote by $A = (a_{ij})$ the m by n incidence matrix of the covering sets, that is

$$a_{ij} = \begin{cases} 1 & \text{if } t_i \in S_j, \\ 0 & \text{otherwise.} \end{cases}$$

Consider an LP-relaxation of the weighted version of the Set Cover problem.

Problem LWSC:

$$
\begin{aligned}
OPT^* \;=\; \text{minimize} \quad & \sum_{j=1}^{n} c_j y_j, \\
\text{subject to} \quad (a) \quad & \sum_{j=1}^{n} a_{ij} y_j \geq 1, \quad \text{for all } i = 1, \ldots, m, \\
(b) \quad & 0 \leq y_j, \qquad\quad \text{for all } j = 1, \ldots, n.
\end{aligned}
\tag{4.3}
$$

Chvátal showed that conditions (4.3a) and (4.3b) imply that

$$
\sum_{j=1}^{n} H\left(\sum_{i=1}^{m} a_{ij}\right) c_j y_j \geq \sum_{j \in \hat{\mathbf{J}}} c_j = APP, \tag{4.4}
$$

where $\hat{\mathbf{J}}$ is the cover output by algorithm "Weighted_Greedy". This immediately leads to

$$
\sum_{j=1}^{n} H(d) c_j y_j = H(d) \sum_{j=1}^{n} c_j y_j \geq APP \tag{4.5}
$$

for any feasible solution to (4.3), since $\sum_{i=1}^{n} a_{ij} = |S_j| \leq d$. In particular, when $\mathbf{y}^* = (y_1^*, y_2^*, \ldots, y_n^*)$ is the optimum (fractional) solution to (4.3) with cost $OPT^* = \sum c_j y_j^*$, we have

$$
H(d) \cdot OPT^* \geq APP. \tag{4.6}
$$

And since $OPT \geq OPT^*$, the theorem follows.

To prove (4.4), Chvátal introduced numbers $x_1, x_2, \ldots, x_m$ representing the "price" paid by algorithm "Weighted_Greedy" to cover each of the points $t_1, t_2, \ldots, t_m$. To make this more precise, denote by $S_j^{(r)}$ the set S_j at the beginning of the r^{th} iteration of the greedy algorithm and assume without loss of generality that the sets in the cover $\mathbf{S}$ are ordered in such a way that algorithm "Weighted_Greedy" will choose index r in its r^{th} iteration. Thus $\hat{\mathbf{J}} = \{1, 2, \ldots, k\}$. If point t_i is covered in the r^{th} iteration, then $x_i = c_r / |S_r^{(r)}|$.

Notice that each point $t_i \in U$ is covered exactly once by algorithm "Weighted_Greedy", hence

$$
\sum_{i=1}^{m} x_i = \sum_{r=1}^{k} \sum_{i \in S_r^{(r)}} x_i = \sum_{r=1}^{k} |S_r^{(r)}| (c_r / |S_r^{(r)}|) = \sum_{r=1}^{k} c_r = APP.
$$

Further technical manipulation would show that

$$\sum_{i=1}^{m} a_{ij}x_i \leq H\left(\sum_{i=1}^{m} a_{ij}\right) c_j$$

for each $j = 1, \ldots, n$. This in turn shows that

$$\sum_{j=1}^{n} H\left(\sum_{i=1}^{m} a_{ij}\right) c_j y_j \geq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij}x_i\right) y_j = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij}y_j\right) x_i \geq \sum_{i=1}^{m} x_i = APP$$

and that completes the proof of (4.4). $\square$

The following example shows that the bounds (4.1) and (4.2) are tight.

Example 4.1 (Chvátal, [24]) *Let $U = \{1, \ldots, m\}$. For $i = 1, \ldots, m$, define $S_i = \{i\}$ and $c_i = 1/i$. Set $S_{m+1} = \{1, \ldots, m\}$ and $c_{m+1} = 1 + \varepsilon$. Then $\hat{\mathbf{J}} = \{1, \ldots, m\}$, $APP = H(m) = H(d)$, and $OPT = c_{m+1} = 1 + \varepsilon$. Making ε arbitrarily close to 1 shows that the cost of the greedy cover can exceed the cost of an optimal cover by a factor arbitrarily close to $H(m) = H(d)$.* $\square$

Let us now turn our attention to partial covers.

Definition 4.1 Let U be a finite set, $|U| = m$, $\mathbf{S}$ be a finite cover of U, $\mathbf{c}$ be positive costs of sets in $\mathbf{S}$, and $0 < p \leq 1$. We say that $\mathbf{J}^* \subset \mathbf{J}$ (or alternatively $\mathbf{S}^* \subset \mathbf{S}$) is a *$p$-partial cover* of U (or, if p is clearly understood, a *partial cover* of U), if

$$|\bigcup_{j \in \mathbf{J}^*} S_j| = |\bigcup_{S_j \in \mathbf{S}^*} S_j| \geq pm.$$

We call $c(\mathbf{J}^*) = c(\mathbf{S}^*) = \sum_{j \in \mathbf{J}^*} c_j$ the *cost* of the partial cover $\mathbf{J}^*$ ($\mathbf{S}^*$) and denote by OPT the minimum possible cost of a p-partial cover of U.

Now we can formally state the Partial Cover and Partial d-Cover problems:

PARTIAL COVER

Instance: Finite set U, finite cover $\mathbf{S}$ of U, positive costs $\mathbf{c}$, $0 < p \leq 1$.

Goal: Find a p-partial cover $\mathbf{S}^* \subset \mathbf{S}$ of U of minimum cost.

PARTIAL d-COVER

Instance: Finite set U, finite cover $\mathbf{S}$ of U with covering sets of at most d elements, positive costs $\mathbf{c}$, $0 < p \leq 1$.

Goal: Find a p-partial cover $\mathbf{S}^* \subset \mathbf{S}$ of U of minimum cost.

Both problems are NP-hard, since they contain SET COVER and d-SET COVER as special cases—simply set $p = 1$. To approximate the optimum solution, we will use a minor modification of the greedy algorithm given by Kearns in [58].

Algorithm "Partial_Greedy"

Input: A cover $\mathbf{S} = \{S_1, \ldots, S_n\}$ of a finite set U, positive costs $\mathbf{c} = \{c_1, c_2, \ldots, c_n\}$ of the covering sets, a number p, $0 < p \leq 1$.

Output: A p-partial subcover $\hat{\mathbf{S}} = \{\hat{S}_1, \ldots, \hat{S}_k\} = \{S_{i_1}, \ldots, S_{i_k}\} \subset \mathbf{S}$ of U.

1. Set $\hat{\mathbf{J}} = \emptyset$.

2. Set $r = \lceil pm \rceil - |\bigcup_{j \in \hat{\mathbf{J}}} S_j|$, i.e. r is the number of elements of U yet to be covered in order to obtain a p-partial cover.

3. If $r \leq 0$, then STOP and output $\hat{\mathbf{J}}$.

4. Find $i \in \mathbf{J} \setminus \hat{\mathbf{J}}$ that minimizes the quotient $\frac{c_j}{\min(r, |S_j|)}$, for $j \in \mathbf{J} \setminus \hat{\mathbf{J}}$ and $S_j \neq \emptyset$. In case of a tie, take the smallest such i.

5. Add i to $\hat{\mathbf{J}}$. For each $j \in \mathbf{J} \setminus \hat{\mathbf{J}}$ set $S_j = S_j \setminus S_i$. Return to Step 2.

Step 4 of the algorithm is motivated by the intuition that the more elements a set has and the cheaper it is, the more likely it is included in a minimum partial cover. On the other hand, a set having more than r elements is no better than a set with exactly r elements.

Even though algorithms "Weighted_Greedy" and "Partial_Greedy" are quite similar, it turns out that the approach used by Chvátal, Lovász, or Johnson cannot be used to establish a reasonable bound on the performance of "Partial_Greedy". The reasons are that only a fraction of points of the set U are covered and that the part of U covered by the optimum partial cover can be completely different from the part covered by the greedy partial cover. This makes the analysis of the performance bound for "Partial_Greedy" quite complicated.

Theorem 4.2 (Kearns, [58]) *Given an instance of the* PARTIAL COVER *problem, algorithm "Partial_Greedy" outputs a partial cover of cost APP satisfying*

$$APP \leq (2H(m) + 3)OPT.$$

Proof: We will not present the proof of this theorem here since it is rather long and complicated. It should be noted however, that Kearns's proof exposes the difficulties mentioned above. In the proof, Kearns considers two subsets of the ground set U, namely the set U_{opt} of all elements covered by the optimum partial cover, and the set $\hat{U}$ of all elements covered by the "greedy" partial cover. As in Chvátal's proof, he assigns to every element covered by the greedy partial subcover a price x_i. Then he obtains the performance bound in two steps—first bounding the total price paid by "Partial_Greedy" when covering elements in $\hat{U} \setminus U_{opt}$ and then when covering $\hat{U} \cap U_{opt}$. The bounds are

$$\sum_{\{i | t_i \in \hat{U} \setminus U_{opt}\}} x_i \leq (H(m) + 2)OPT$$

and

$$\sum_{\{i | t_i \in \hat{U} \cap U_{opt}\}} x_i \leq (H(m) + 1)OPT.$$

Adding these two bounds gives the final performance ratio of $2H(m) + 3$. $\square$

4.2 Performance Guarantee

Let $\{j_1, j_2, \ldots, j_l\}$ be a partial cover of U with the minimum cost OPT. To simplify the notation, denote the sets of this partial cover by $A_1, A_2, \ldots, A_l$ and their costs by $\{\alpha_1, \alpha_2, \ldots, \alpha_l\}$, that is $\{A_1, A_2, \ldots, A_l\} = \{S_{j_1}, S_{j_2}, \ldots, S_{j_l}\}$ and $OPT = \alpha_1 + \alpha_2 + \cdots + \alpha_l$. Without loss of generality, we can assume that the sets $A_1, \ldots, A_l$ are disjoint and that

$$\sum_{s=1}^{l} |A_s| = \lceil pm \rceil. \tag{4.7}$$

This can be accomplished by possibly deleting some elements from the sets A_s, thus increasing the cost per element, hence possibly increasing the worst-case cost of the greedy partial cover; the resulting sets A_s still form a p-partial cover, hence the value of the optimal solution does not increase.

Denote by k the number of sets in the partial cover output by the greedy algorithm and by APP the cost of this "greedy" partial cover. We can assume without loss of generality that the sets in the cover $\mathbf{S}$ are ordered in such a way that the greedy algorithm chooses index i in its i-th iteration; that is, $APP = c_1 + c_2 + \cdots + c_k$.

Denote by $r^{(i)}$ the number of elements of U remaining to be covered at the beginning of the i^{th} iteration of algorithm "Partial_Greedy". Let $A_s^{(i)}$ and $S_j^{(i)}$ denote the sets A_s and S_j at the beginning of the i^{th} iteration. Define $u_j^{(i)} = \min(r^{(i)}, |S_j^{(i)}|)$ and $a_s^{(i)} = \min(r^{(i)}, |A_s^{(i)}|)$. Thus $u_j^{(i)}$ is the number of "relevant" elements of the set S_j at the beginning of the i^{th} iteration.

In every iteration i of the greedy method, we choose a subscript j for which $c_j/u_j^{(i)}$ is minimal, thus, in particular,

$$\frac{c_i}{u_i^{(i)}} \leq \frac{\alpha_s}{a_s^{(i)}}$$

for all $s = 1, \ldots, l$ for which $A_s^{(i)} \neq \emptyset$.

Consider all fractions of the form

$$\frac{\alpha_s}{k_s}, \qquad s = 1, \ldots, l, \;\; k_s = 1, \ldots, |A_s|.$$

Note that we have $r^{(1)} = \lceil pm \rceil$-many fractions. Let us rearrange these fractions into a non-increasing sequence $e_1 \geq e_2 \geq \cdots \geq e_{r^{(1)}}$. Then the following inequality holds.

Lemma 4.1 *For each $i = 1, \ldots, k$ we must have*

$$\frac{c_i}{u_i^{(i)}} \leq e_{r^{(i)}}.$$

Proof: At the beginning of the i-th iteration of the greedy algorithm, $i = 1, \ldots, k$, there are exactly $r^{(i)}$ elements to be covered, i.e. there are at least $r^{(i)}$ elements in $\bigcup A_s$ not deleted in the previous steps. Hence $\sum_{s=1}^{l} a_s^{(i)} \geq r^{(i)}$. The greedy condition implies that

$$\frac{c_i}{u_i^{(i)}} \leq \frac{\alpha_s}{a_s^{(i)}}$$

for all $s = 1, \ldots, l$ for which $a_s^{(i)} > 0$. Therefore

$$\frac{c_i}{u_i^{(i)}} \leq \frac{\alpha_s}{k_s},$$

for all $s = 1, \ldots, l$ and all $k_s = 1, \ldots, a_s^{(i)}$, i.e.

$$\frac{c_i}{u_i^{(i)}} \leq e_j$$

for at least $r^{(i)}$ indices j. But $e_1 \geq \cdots \geq e_{r^{(1)}}$, hence

$$\frac{c_i}{u_i^{(i)}} \leq e_{r^{(i)}}$$

for any $i = 1, \ldots, k$. $\square$

Note: It can be shown that the worst possible scenario for the quotient APP/OPT can be obtained by considering only those greedy p-covers of U consisting of single-element sets. Let T_j, $j = 1, \ldots, \lceil pm \rceil$ be a collection of disjoint singletons with costs $C_1 \leq C_2 \leq \cdots \leq C_{\lceil pm \rceil}$ output by the greedy algorithm. In order for T_1 to get selected in the first step of the greedy algorithm, we must have $C_1 \leq e_{\lceil pm \rceil}$, with similar inequalities at the following steps. Thus $C_{\lceil pm \rceil - j + 1} \leq e_j$ for all $j = 1, \ldots, \lceil pm \rceil$. Lemma 4.1 simply modifies these inequalities for general case.

As a straightforward consequence of Lemma 4.1, we easily obtain:

Lemma 4.2

$$c_1 + \cdots + c_k \leq \alpha_1 H(|A_1|) + \cdots + \alpha_l H(|A_l|).$$

Proof: Clearly,

$$\frac{c_i}{u_i^{(i)}} \leq e_{r^{(i)}} \leq e_{r^{(i)}-1} \leq \cdots \leq e_{r^{(i+1)}+1}.$$

Since $r^{(i)} - r^{(i+1)} = u_i^{(i)}$, we have

$$c_i \leq e_{r^{(i)}} + e_{r^{(i)}-1} + \cdots + e_{r^{(i+1)}+1}.$$

Adding the above inequalities for $i = 0, \ldots, k-1$, we have

$$c_1 + \cdots + c_k \leq e_{r^{(1)}} + \cdots + e_1 = \alpha_1 H(|A_1|) + \cdots + \alpha_l H(|A_l|).$$

□

Combining Lemmas 4.1 and 4.2, and the fact that $H(|A_s|) \leq H(d)$ for all $s = 1, \ldots, l$ we easily obtain the following generalization of Chvátal's, Johnson's, and Lovasz's bound.

Theorem 4.3 *Algorithm "Partial_Greedy" approximates* PARTIAL d-COVER *with performance ratio satisfying*

$$\frac{APP}{OPT} \leq H(d). \tag{4.8}$$

□

Equality (4.7) implies that $|A_s| \leq \lceil pm \rceil$ for all $s = 1, \ldots, l$. This and Lemma 4.2 prove the following.

Theorem 4.4 *Algorithm "Partial_Greedy" approximates* PARTIAL COVER *with performance ratio satisfying*

$$\frac{APP}{OPT} \leq H(\lceil pm \rceil). \tag{4.9}$$

□

Theorem 4.4 might seem to be a weaker version of Theorem 4.3. Of course, this is the case when $p = 1$, since $d \leq m$. On the other hand, for $p < 1$, it might be that $d > \lceil pm \rceil$, and in this case, Theorem 4.4 is stronger than Theorem 4.3.

The performance bounds (4.8) and (4.9) are clearly tight. To see that, we can modify Chvátal's Example 4.1.

Example 4.2 Given $U = \{1, \ldots, m\}$ and $0 < p \leq 1$ we can construct a cover of U as follows. Set $u = \lceil pm \rceil$. For $i = 1, \ldots, m$, define $S_i = \{i\}$. Set $S_{m+1} = \{1, \ldots, u\}$. Define $c_i = 1/(u - i + 1)$ for $i = 1, \ldots, u$, and $c_i = 1$ for $i = u + 1, \ldots, m + 1$. Then $\hat{\mathbf{J}} = \{1, \ldots, u\}$, $APP = \frac{1}{u} + \cdots + \frac{1}{1} = H(u)$, and OPT=$c_{m+1} = 1$. Therefore $APP/OPT = H(u) = H(\lceil pm \rceil) = H(d)$. □

Chapter 5

Generalized Traveling Salesman and Group Steiner Tree Problems

In this chapter, we give the first algorithm with a non-trivial performance bound for approximating the GENERALIZED TRAVELING SALESMAN PROBLEM—a generalization of the classical TSP well-known among the Operations Research community; and significantly improve the known results for the GROUP STEINER TREE PROBLEM.

Given a collection of cities grouped into possibly intersecting clusters, the GENERALIZED TRAVELING SALESMAN PROBLEM (GTSP) is to find a shortest tour through some of the cities such that at least one city from each cluster gets visited. The closely related GROUP STEINER TREE PROBLEM (GROUP-STEINER) is to find a tree of minimum length that intersects all clusters. Both problems generalize a number of well-known problems in combinatorial optimization and have a wide range of applications including job scheduling on multi-state CNC machines and VLSI design.

We first focus on the case in which the graph is a weighted tree; this trivially generalizes the SET COVER problem. Under the assumption that the number of cities from the same cluster in certain subtrees is bounded, we obtain an algorithm that asymptotically matches the best possible approximation ratio for SET COVER and approximates both GTSP and GROUP-STEINER within $O(\log m)$, where m is the total number of clusters. Our algorithm is based on "tree-stripping"—a technique specifically designed to round the solution to the LP relaxation of the Group Steiner Tree problem, but hopefully useful for approximating other related problems. In the second part of this chapter, we discuss the Generalized Traveling Salesman and Group Steiner Tree problems on a general weighted graph with clusters of size at most ρ. We show that in this case, GTSP can be approximated to within $3\rho/2$ and Group-Steiner to within 2ρ.

5.1 Introduction

A different generalization of the Set Cover problem considers "distances between sets". In this case, the sets we choose as well as the order in which we choose them are important. The following example (modified from [47] and [17]) describes this in more detail.

Example 5.1 Professor Peter Perfect decides to run a few errands during his lunch break. He needs to *have lunch, buy some groceries, buy flowers for his fiancee, mail a letter*, *get some cash*, and then return back to his office. Each errand can be done at several different locations. In particular, he can *have lunch* at *McDonald's* or at the *supermarket*, *buy groceries* at *the local convenience store* or at *the supermarket*, *buy flowers* at *the flower shop* or at *the convenience store*, *mail a letter* at *the supermarket* or at *the post office*, and *get cash* at *the supermarket*, at *a nearby bank*, or at *a cash machine*. Since he always tries to do things in the best possible way, he decides to first create a plan how to run these errands in minimum time. Figure 5.1 shows two (solid versus dashed) of many possible ways of completing all errands.

Assuming that each errand takes the same time at any given location, and that there is no delay between two errands performed at the same location, Professor Perfect "simply" has to choose a partial tour through some of the stores starting and ending at his office, such that all errands can be completed and the total traveling time is minimal.

Another motivation for our problem is plan developing for manufacturing metal parts on CNC machines—see [47] for details. Here we start with a rectangular block of metal and use a machining center to cut a variety of features into the block. The machining center

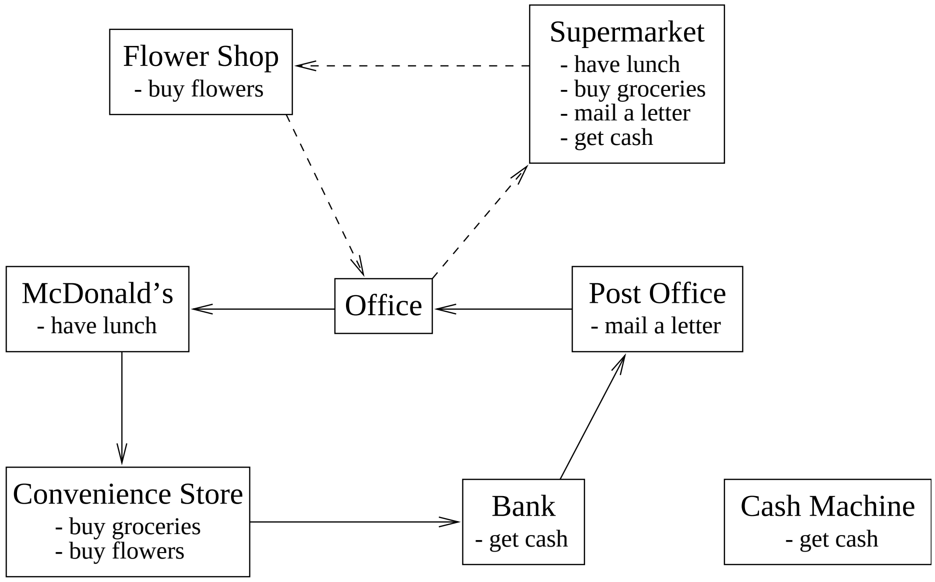


Figure 5.1: Errands of Professor Perfect with two possible solutions

can perform a large variety of different types of operations such as drilling, milling, etc. but it can only do one operation at a time. In classical planing terms, each geometric feature of the metal part is a *goal*. For each goal, there may be a number of methods (*operators*) that can achieve that goal.

The time it takes to complete each goal can be divided into two parts—the time needed for setting up the operator and the time used for actually performing the operation. If we can accomplish two or more goals using the same operator, we encounter the set-up time only the first time we use that operator. We can assume that the actual operations take roughly the same time to complete regardless of the operators used; moreover the set-up usually accounts for a large portion of the time cost of each operator—sometimes as much as 90% of the total time. Thus, in order to make an efficient plan, one has to overlap the operators as much as possible.

The set-up times are different and depend not only on the current operator but also on the previous operator used—it certainly takes less time to simply change the tool in the machining center than to change the whole *fixturing* (that is the way the part is clamped, its orientation, etc.). Hence the set-up time can be regarded as a distance between the operators.

Thus, similar to the first example of errand scheduling, in order to manufacture the metal part in shortest time possible, we have to design a partial tour through the operators, such that all the goals can be accomplished and the total set-up time (total distance) is minimum.

This can be generalized in a straightforward way to the case where we have several *machines* and a set of *tasks* which can be performed one at a time on these machines.

More formally, let U be a set (of tasks/errands) of size m, and let G be an edge-weighted graph (of locations). Associated with each vertex i in G is a set $S_i \subset U$ (the errands that can be performed at location i). The ERRAND SCHEDULING problem is to find a shortest partial tour through G such that each element of U is contained in at least one "visited" subset (that is each task can be completed). We allow repetitions of vertices in the partial tour. Closely related is the TREE COVER problem where the goal is to find a subtree of G of minimum length such that each element of U is contained in at least one "visited" subset.

An equivalent formulation of the ERRAND SCHEDULING problem is known in the Operations Research community as the GENERALIZED TRAVELING SALESMAN problem (GTSP). It can be informally described as follows: Given a set of cities grouped into (possibly intersecting) clusters, find a shortest partial tour through the cities that visits at least one city from each cluster. Besides errand and task scheduling, GTSP has a host of other practical applications, among them routing of welfare clients through governmental agencies (see e.g. [83]) or routing of postal vans (see [59]).

The TREE COVER problem can be equivalently formulated as the GROUP STEINER TREE problem (GROUP-STEINER). Given a graph with vertices grouped into clusters, find a shortest tree which contains at least one node from each cluster. The GROUP STEINER TREE problem is used to model connections between terminals with multiple ports in VLSI design. Here each vertex corresponds to a different port and clusters correspond to the ports from the same terminal—see [80] for details.

5.1.1 Formal Definitions

Definition 5.1 Let $G = (V, E)$ be an undirected graph, and let $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m\}$ be a collection of (possibly intersecting) clusters of vertices in V, that is $\mathcal{C}_j \subset V$. A closed path going through some of the vertices of V is called a *generalized tour (GT)* if it visits at least one vertex from each cluster. Similarly, a subtree of G is called a *group-Steiner tree (GST)* if it contains at least one vertex from each cluster.

GENERALIZED TRAVELING SALESMAN PROBLEM

Instance: Edge-weighted graph $G = (V, E)$ and a (not necessarily exhaustive) clustering $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m\}$ of V.

Goal: Find a generalized tour of minimum length.

GROUP STEINER TREE PROBLEM

Instance: Edge-weighted graph $G = (V, E)$ and a (not necessarily exhaustive) clustering $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m\}$ of V.

Goal: Find a group-Steiner tree of minimum length.

5.1.2 Our Results

In this chapter, we give the first approximation algorithms with nontrivial performance guarantees for the GENERALIZED TRAVELING SALESMAN problem and significantly improve the results of Ihler in [51, 53] for the GROUP STEINER TREE problem in two special cases.

In Section 5.2 we consider the case where the graph of locations is a tree; here GTSP and GROUP-STEINER are equivalent. This trivially contains the SET COVER problem as a very special case, thus the approximation of even such a restricted version of GTSP is quite challenging. Under the general assumption that the number of vertices of certain subtrees belonging to the same cluster is bounded, we give an approximation algorithm with performance ratio $\Theta(\ln m)$, where m is the number of clusters. Considering the fact that

SET COVER itself cannot be approximated better than $\Theta(\ln m)$, our algorithm asymptotically matches the best possible performance guarantee. Our approach is based on solving the LP relaxation of the corresponding integer programming problem and then using our "tree-stripping" technique (see Section 5.2 for details) to transform the whole problem into finding a minimum-cost set cover. We believe that the technique of tree-stripping, though specifically designed for solving GROUP-STEINER on trees, can be used for approximating other similar problems.

In Sections 5.3 and 5.4 we discuss GTSP and GROUP-STEINER on general graphs with the restriction that the size of each cluster is at most ρ. This models the quite realistic situation where each errand can be completed at at most ρ locations or each terminal has at most ρ ports. Using again the optimal solution to the LP relaxation of the general GTSP problem (or GROUP-STEINER) plus the techniques developed for solving the PRIZE-COLLECTING TSP (see, for example, [18] or [85]), we show that both GTSP and GROUP-STEINER can be approximated with a constant performance bound. In particular, we prove that the GTSP problem can be approximated within $3\rho/2$ and the GROUP-STEINER problem within 2ρ. These contain as special cases the above mentioned result of Christofides [23] for approximating TSP and the early results for approximating STEINER TREE.

5.1.3 Previous Results

The GROUP STEINER TREE problem was originally introduced by Reich and Widmayer in [80] and then considered by Ihler and also by Garg and Ravi in [40]. Reich and Widmayer in [80] discussed the running time of two heuristics for approximating GROUP-STEINER, but were "unable to give tight bounds on the qualities of the approximate solutions." Their first heuristic first constructed an (approximate) Steiner tree with terminals $W = \bigcup \mathcal{C}_j$ and then one by one removed excessive vertices, i.e. those vertices which were not needed for the group-Steiner tree. Their second heuristic began with some starting vertex and successively built a tree, at each iteration adding the closest vertex from a group not represented in the tree.

Ihler in [51] proved that the second heuristic of Reich and Widmayer has an $m-1$ performance bound, where m is the number of clusters, and presented another heuristic with the same performance ratio that simply selects from each cluster the point closest to the starting vertex. In [52] he showed that the Group Steiner Tree problem is at least as hard to approximate as Set Cover, which is a rather trivial observation in our "dual" formulation of Group-Steiner as the Tree Cover problem (see Subsection 5.1.6). In [53], Ihler discussed the rectilinear version of the Group-Steiner problem in a special case when all required points lie on two parallel lines. He showed that even with such a restriction, the problem remains NP-hard. But he gave a linear time algorithm for the exact solution to Group-Steiner with the additional requirement that the groups on the two parallel lines consist of non-overlapping clusters. Some further implementation results for the above heuristics and some additional hardness results for restricted cases of Group-Steiner are discussed in [54].

Very recently, the Group Steiner Tree problem acquired large attention among theory researchers. In fact, after the original appearance of our results, there have been two papers which improve on our bounds in the most general cases. Bateman, Helvig, Robins, and Zelikovsky in [14] approximated the Group-Steiner within $(1+\ln\frac{m}{2})\sqrt{m}$. They use the fact that the optimal group-Steiner tree can be approximated reasonably well by an optimal group-Steiner 2-star (a tree of depth at most 2). Their algorithm then approximates this optimal group-Steiner 2-star. Garg, Konjevod, and Ravi in [39] give a randomized algorithm with performance ratio of $O(\ln^4 n)$ where n is the number of vertices. Their algorithm first randomly approximates the complete edge-weighted graph with a tree, then solves the LP-relaxation of Group-Steiner on this tree, and then uses a version of randomized rounding to find the integer solution. Even though these results are stronger than ours for general graphs with no assumptions about the number or size of clusters, in cases where there is some a priori knowledge about the number or size of groups, our bounds provide stronger approximation guarantees. The above results of [14] and [39] for Group-Steiner can be easily modified to provide similar approximation bounds for GTSP.

Even though the Generalized Traveling Salesman problem is a relatively old and quite natural problem, there had not been any worst-case performance bounds established for this problem (with the exception of [14] and [39]), most likely because of the inherent difficulties in providing any reasonable approximations. Laporte and Nobert in [60] give an (exponential-time) algorithm for finding the optimal solution to GTSP using an integer programming formulation of the problem. Noon and Bean in [71] and Saksena in [83] discuss other methods for finding an optimum solution. The only polynomial-time approximation algorithm for this problem is due to Malandraki—see [68]. His heuristic is a restriction of the exact algorithm based on dynamic programming and does not seem to exhibit a good worst-case behavior.

Several papers have been published constructing transformations of the GENERALIZED TSP into the standard TSP (see for example [63] or [72]). These transformations usually preserve the lengths of both optimal and approximate solutions, but they transform an instance of GTSP on a (possibly undirected) graph where the distance function satisfies the triangle inequality into an instance of standard TSP on a much larger directed graph **without** the triangle inequality. Thus the resulting problem is in fact more difficult than the original GTSP.

The GENERALIZED TRAVELING SALESMAN problem is a special case of the TRAVELING PURCHASER PROBLEM discussed by Ong in [73]. Ong proposed several natural heuristics for approximating this problem and discussed their performance in terms of experimental computations on selected instances. His algorithms are combinations of greedy strategies and local improvements and do not have a good worst-case performance. In fact, one can construct instances where his heuristics perform as badly as $\Theta(m)$.

5.1.4 Related Results

A problem somewhat similar to GTSP and GROUP-STEINER is discussed by Arkin and Hassin in [5]. Given the same set-up as for GTSP, their goal was to find a subgraph G' of G of *minimum diameter* containing at least one vertex from each cluster. This problem is of course much simpler than GROUP-STEINER and hence their algorithm cannot provide any reasonable approximations to either GTSP or GROUP-STEINER.

In [17], Bhatia, Khuller, and J. Naor consider a problem somewhat related to GTSP. Their Loading Time Scheduling problem is also a generalization of Set cover problem, but in a different direction. In their problem, there are precedence constraints limiting which location can be chosen at any given time but, on the other hand, instead of considering distances *between* locations they assign a weight to *each* location. Because of these principal differences, their approach to approximating the Loading Time Scheduling problem is not useful for us.

Recently a class of optimization problems including k-TSP, k-MST, and other related problems acquired much attention among theory researchers and this led to significant improvements in both performance guarantees and complexity of approximation algorithms for these problems. See, for example, [7], [12], [13], [18], [19], [20], [38], [43], [77], [78], etc. In the k-TSP problem, the goal is to find a shortest partial tour visiting at least k vertices. In the k-MST problem, one tries to find a shortest tree spanning at least k vertices. The difference between these two problems and GTSP (Group-Steiner) is the fact that feasible solutions to k-TSP or k-MST have to satisfy some *quantitative* requirements—a partial tour has to visit at least k vertices or a tree has to span at least k nodes; whereas the condition on feasible solutions to GTSP or Group-Steiner is of a *qualitative* nature—each cluster has to be visited. Thus, in any of the above mentioned problems, the more vertices we choose, the closer we are to satisfying the constraints; on the other hand, in the GTSP(Group-Steiner) problem, it might happen that we select many "seemingly good" vertices but still remain far from visiting *all* clusters. Thus k-TSP and k-MST are also quite different from GTSP and Group-Steiner.

5.1.5 Hardness of Approximation

The Generalized Traveling Salesman and Group Steiner Tree problems generalize and contain as special cases several well-known optimization problems, among them the Set Cover and Traveling Salesman problems. As mentioned in Subsection 3.1.1, no polynomial-time algorithm can approximate Set Cover better than $\Omega(\ln m)$ where m is the size of the ground set, unless some unlikely collapses in the polynomial hierarchy occur. Thus, when approximating GTSP or GROUP-STEINER, we cannot hope for a better approximation ratio than $\Theta(\ln m)$. On the other hand, the structure of GTSP or GROUP-STEINER is more complicated than that of SET COVER hence performance ratios worse than $\Theta(\ln m)$ for approximating general GTSP and GROUP-STEINER should be expected.

5.1.6 Preliminaries

Let $G = (V, E)$ be an undirected graph with vertex set $V = \{0, 1, \ldots, n\}$ (of locations). For each edge $e = (i, j) \in E$ let $l_e = l_{(i,j)}$ be the distance between vertices i and j. We assume that the distance function is symmetric and satisfies the triangle inequality.

For simplicity we assume that the starting vertex of the tour or the root of the subtree (the office, in our example) is given. This assumption can be easily removed by running the approximation algorithms several times, each time choosing a different starting vertex from the smallest cluster. To make this assumption explicit, we add one more cluster, namely $\mathcal{C}_o = \{0\}$ to $\mathbf{C}$. This will guarantee that vertex 0 is always visited and we can use it as our starting vertex. Note that the clusters can possibly intersect and it is not necessary that every vertex belong to some cluster. Also, since vertex 0 is always part of a feasible solution, we can assume without loss of generality that $\mathcal{C}_j \subset V \setminus \{0\}$ for all $j = 1, \ldots, m$. (If not, we could simply disregard such cluster from our considerations.)

It is convenient for our purposes to formulate "dual" versions of GTSP and GROUP-STEINER.

Definition 5.2 Let $G = (V, E)$ be an undirected graph (of locations). Let $U = \{t_1, \ldots, t_m\}$ be a "ground set" (of errands). Associated with each vertex $i \in V$ is a set $S_i \subset U$ (the set of errands which can be performed at the i^{th} location). We assume that $S_0 = \emptyset$ and $\bigcup S_i = U$. A tour through the vertices of some set $V' \subset V$ is called a *covering tour* if $0 \in V'$ and $\bigcup_{i \in V'} S_i = U$. Similarly, let T be a subgraph of G which is a tree. We call T a *covering tree* if $0 \in V(T)$ and $\bigcup_{i \in V(T)} S_i = U$.

Errand Scheduling Problem (ESP)
Instance: Edge-weighted graph $G = (V, E)$, a set U, and subsets S_i of U associated with each vertex of V.
Goal: Find a covering tour of minimum length.

Tree Cover Problem (TCP)
Instance: Edge-weighted graph $G = (V, E)$, a set U, and subsets S_i of U associated with each vertex of V.
Goal: Find a covering tree of minimum length.

Given $G = (V, E)$ and clusters $\mathcal{C}_o, \ldots, \mathcal{C}_m$, one can set $U = \{t_1, \ldots, t_m\}$ and $S_i = \{t_j \,|\, i \in \mathcal{C}_j\}$. Conversely, given $G = (V, E)$, $U = \{t_1, \ldots, t_m\}$, and the sets S_i, one can set $\mathcal{C}_o = \{0\}$ and $\mathcal{C}_j = \{i \,|\, t_j \in S_i\}$. In either case, any generalized tour is a covering tour and vice versa; hence GTSP and ESP are equivalent. Similarly, any group-Steiner tree is a covering tree and vice versa, hence Group-Steiner and TCP are equivalent.

Similarly as before, we denote by $A = (a_{ji})$ the m by n incidence matrix of the clusters $\mathcal{C}_j$, $j = 1, \ldots, m$ (or alternately sets S_i, $i = 1, \ldots, n$), that is

$$a_{ji} = \begin{cases} 1 & \text{if } i \in \mathcal{C}_j \text{ (that is } t_j \in S_i) \\ 0 & \text{otherwise.} \end{cases}$$

We will denote by OPT the cost of an optimal solution to a problem under discussion and by APP the cost of the approximate solution.

5.2 GTSP and Group-Steiner on Trees

Let us now consider the case when the graph $G = (V, E)$ is a tree. Any partial tour in G is clearly at least twice as long as the subtree T determined by the vertices visited (the tour passes through each edge of T at least twice). On the other hand, given a subtree T of G, we can use depth-first traversal to construct a tour through vertices of T

twice as long as T. Therefore $length$(minimum GT) = 2 $length$(minimum GST) (that is $length$(minimum generalized tour) = 2$length$(minimum covering tree)) and solving GTSP (ESP) is equivalent to solving GROUP-STEINER (TCP). Hence, in what follows, we will seek an approximation to the optimal solution of GROUP-STEINER (TCP) only, under the restriction that $G = (V, E)$ is a tree.

Even though this problem is simpler than the general GROUP-STEINER (TCP), it of course remains NP-hard. In fact, any instance of the (weighted) SET COVER problem can be reduced to an instance of the TCP on trees. To see this, given any collection $\{S_1, S_2, \ldots, S_n\}$ of subsets of U with costs $\{c_1, c_2, \ldots, c_n\}$, simply set G to be a star with center 0 and leaves $1, 2, \ldots, n$. Then set $S_o = \emptyset$, and for all $i = 1, \ldots, n$ associate each leaf i with a subset S_i and define the length of an edge from 0 to i to be c_i—see Figure 5.2. Thus even in the simple case when G is a tree, we cannot in general approximate the length of a minimum covering tree (minimum group-Steiner tree) better than $\Omega(\log m)$.

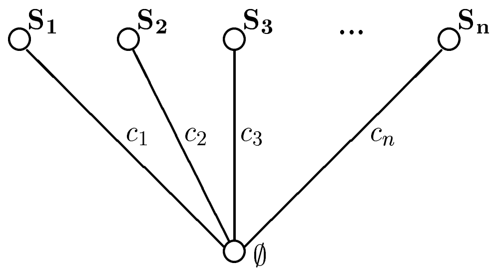


Figure 5.2: SET COVER is a special case of TCP on trees

We will regard G as a rooted tree with root 0. Let $\pi(i)$ be the parent of node $i \in V$, $\pi(0) = \text{NIL}$. Since each edge joins some node i and its parent, we can set $e_i = (i, \pi(i))$ and $c_i = l_{e_i} = l_{(i,\pi(i))}$ for all $i = 1, 2, \ldots, n$. For simplicity, set $c_o = 0$.

5.2.1 Polynomially Solvable Cases of Group-Steiner (TCP) on Trees

Consider first the case when graph G is a "line-like" tree, that is all nodes have either degree 1 or 2—see Figure 5.3.

Node 0 splits G into two parts. Without loss of generality, we can enumerate the nodes in one part as $\{1, 2, \ldots, n_R\}$ and in the other part as $\{-1, -2, \ldots, -n_L\}$ where $n_R + n_L = n$.

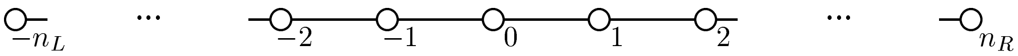


Figure 5.3: TCP on line

Any subtree of G containing 0 is uniquely determined by its two end-points, n_1 and n_2, where $n_1 \leq 0 \leq n_2$, and contains all nodes from the interval $[n_1, n_2]$. We have only $(n_L+1)(n_R+1) = O(n^2)$ such intervals, hence we can check all of them and determine which one corresponds to the covering tree (GST) of minimum cost. Using a simple algorithm, this can be done in $O(nm)$ time.

The case of a line-like tree can be generalized in a straightforward way to the case when tree G consists of K "line segments" with 0 as one end-point. A little more thought might be necessary to convince oneself that even in the case of a general tree having constant number K of leaves, one can enumerate all $O(n^K)$ possible subtrees containing 0, check each one of them (with possible shortcuts) to determine whether all clusters are visited (all elements of U are covered), and then choose a group-Steiner tree (covering tree) of minimum length. Using a simple algorithm, this can be done in $O(n^{K-1}m)$ time.

5.2.2 Integer Program and LP Relaxation

For any tree $T \subset G$, set $y_i = 1$ if $i \in T$ and 0 otherwise. We can alternately formulate Group-Steiner (TCP) on trees as the following integer program.

Problem $\mathbf{ITCP_1}$:

$$
\begin{aligned}
OPT_1 \;=\; \text{minimize} \quad & \sum_{i \in V \setminus \{0\}} c_i y_i, \\
\\
\text{subject to} \quad & \text{(a)} \quad y_o = 1, \\
& \text{(b)} \quad \sum_{i=1}^{n} a_{ji} y_i \geq 1, \quad \text{for all } j = 1, \ldots, m, \\
& \text{(c)} \quad y_i \leq y_{\pi(i)}, \qquad \text{for all } i \in V \setminus \{0\}, \\
& \text{(d)} \quad y_i \in \{0, 1\}, \qquad \text{for all } i \in V \setminus \{0\}.
\end{aligned}
\tag{5.1}
$$

Clearly, the original formulation of TCP (Group-Steiner) and the integer program ITCP_1 are equivalent. Condition (5.1a) guarantees that node 0 is always selected. Condition (5.1b) assures that the sets corresponding to the selected nodes cover the ground set (the subtree contains at least one vertex from each cluster). Condition (5.1c) guarantees that the selected nodes form a tree. The structure of (5.1) is particularly simple because of the fact that graph G is a tree. The general case (see Section 5.3) is more complicated.

Relaxing condition (5.1d) yields the following linear program.

Problem $\mathbf{LTCP_1}$:

$$
\begin{aligned}
OPT_1^* \;=\; \text{minimize} \quad & \sum_{i \in V \setminus \{0\}} c_i y_i, \\
\\
\text{subject to} \quad & \text{(a)} \quad y_o = 1, \\
& \text{(b)} \quad \sum_{i=1}^{n} a_{ji} y_i \geq 1, \quad \text{for all } j = 1, \ldots, m, \\
& \text{(c)} \quad y_i \leq y_{\pi(i)}, \qquad \text{for all } i \in V \setminus \{0\}, \\
& \text{(d)} \quad 0 \leq y_i \leq 1, \qquad \text{for all } i \in V \setminus \{0\}.
\end{aligned}
\tag{5.2}
$$

We can solve Problem LTCP_1 using any of the known polynomial time algorithms for solving linear programming problems (ellipsoid method, Karmarkar's method, etc.) and obtain the optimal solution $(\mathbf{y}^*; OPT_1^*) = (y_o^*, y_1^*, \ldots, y_n^*; OPT_1^*)$. But can we use $(\mathbf{y}^*; OPT_1^*)$ to obtain a reasonable approximation of the minimum covering tree (minimum GST)? In what follows we will show that this is indeed the case.

Let $\mathbf{y} = (y_o, y_1, \ldots, y_n) \in \mathbf{R}_+^{n+1}$. Define $V_{\mathbf{y}} = \{i \in V \mid y_i > 0\}$. For any $t_j \in U$ define $cov_{\mathbf{y}}(t_j)$, the "covering number" of t_j with respect to $\mathbf{y}$, as

$$cov_{\mathbf{y}}(t_j) = \sum_{i: t_j \in S_i} y_i = \sum_{i=1}^{n} a_{ji} y_i.$$

Intuitively, $cov_{\mathbf{y}}(t)$ indicates how many times t is covered by fractions y_i of the sets in $V_{\mathbf{y}}$. If $\mathbf{y}$ is a feasible solution to (5.2) then conditions (5.2b) and (5.2c) imply that $cov_{\mathbf{y}}(t) \geq 1$ for all $t \in U$ and subgraph $T \subset G$ generated by $V_{\mathbf{y}}$ is a tree containing 0. In that case, we call the pair $(T, \mathbf{y})$ a *"fractional" covering tree* and define the "length" of $(T, \mathbf{y})$ as $length^*(T, \mathbf{y}) = \sum_{i=1}^{n} c_i y_i$. Thus $(T, \mathbf{y}^*)$ is a fractional covering tree of minimum length.

5.2.3 Tree Sets

Assume without loss of generality that the vertices of tree $G = (V, E)$ are numbered in a breadth-first fashion starting at vertex 0, that is the vertices adjacent to root 0 are $1, 2, \ldots, K$. Removing edges $e_1, e_2, \ldots, e_K$ results in isolating vertex 0 and creating a forest of K subtrees $G_1, G_2, \ldots, G_K$ with roots $1, 2, \ldots, K$.

Let $X \subset U$. We say that X is a "tree set" if $X = \emptyset$ or there is an index $k \in \{1, \ldots, K\}$ and a subtree $Z \subset G_k$ with $k \in Z$ such that $X = \bigcup_{i \in Z} S_i$. We define $C(X)$, the cost of tree set X, to be $C(X) = \sum_{i \in Z} c_i$. Set $C(\emptyset) = 0$.

For any subtree T of G containing 0, set $T_k = T \cap G_k$ and $Y_k = \bigcup_{i \in T_k} S_i$. Then T_k is either empty or is a subtree of G_k with $k \in T_k$; hence Y_k is a tree set, and the cost of tree T_k satisfies $cost(T_k) = \sum_{i \in T_k} c_i = C(Y_k)$ for $k = 1, \ldots, K$.

If T is a covering tree then

$$\bigcup_{k=1}^{K} Y_k = \bigcup_{k=1}^{K} \bigcup_{i \in T_k} S_i = \bigcup_{i \in T} S_i = U,$$

hence $\{Y_1, Y_2, \ldots, Y_K\}$ covers U. Also,

$$cost(\{Y_1, Y_2, \ldots, Y_K\}) = \sum_{k=1}^{K} C(Y_k) = \sum_{k=1}^{K} cost(T_k) = length(T).$$

Hence we have proved the following.

Lemma 5.1 *Any covering tree $T \subset G$ induces a cover $\mathbf{Y} = \{Y_1, Y_2, \ldots, Y_K\}$ of U by tree sets such that*

$$length(T) = cost(\mathbf{Y}).$$

□

We also have the following counterpart of Lemma 5.1.

Lemma 5.2 *Any cover $\mathbf{X} = \{X_1, X_2, \ldots, X_R\}$ of U by tree sets induces a covering tree $T \subset G$, such that*

$$length(T) \leq cost(\mathbf{X}).$$

Proof: Let $\mathbf{X} = \{X_1, \ldots, X_R\}$ be a collection of tree sets covering U and let $\{Z_1, \ldots, Z_R\}$ be the corresponding subtrees, that is for each $r = 1, \ldots, R$, there is an index k_r such that $Z_r \subset G_{k_r}$, $k_r \in Z_r$, and $X_r = \bigcup_{i \in Z_r} S_i$. Tree $T = \bigcup Z_r$ can be obtained in a straightforward way using the following algorithm.

Algorithm "Cover_to_Tree"

Input: A cover $\mathbf{X} = \{X_1, \ldots, X_R\}$ of U by tree sets corresponding to subtrees $\{Z_1, \ldots, Z_R\}$.

Output: A covering tree T such that $length(T) \leq cost(\mathbf{X})$.

begin
  $T \leftarrow \{0\}$;
  for $r \leftarrow 1$ **to** R **do**
    $T \leftarrow T \cup Z_r$;
  while some leaf $i^* \in T$ can be removed without changing the set $\bigcup_{i \in T} S_i$ **do**
    $T \leftarrow T \setminus \{i^*\}$;
  return T;
end

Clearly, the algorithm runs in time polynomial in m, n, and R; the exact time would depend on the implementation. For each $k = 1, \ldots, K$, $T_k = T \cap G_k = \bigcup_{r:k_r=k} Z_r$ is either empty or a subtree of G_k containing k. If $\{X_1, X_2, \ldots, X_R\}$ is a cover of U, then

$$\bigcup_{i \in T} S_i = \bigcup_{k=1}^{K} \bigcup_{i \in T_k} S_i = \bigcup_{k=1}^{K} \bigcup_{r:k_r=k} \bigcup_{i \in Z_r} S_i = \bigcup_{k=1}^{K} \bigcup_{r:k_r=k} X_r = \bigcup_{r=1}^{R} X_r = U,$$

thus T is a covering tree. Moreover

$$cost(T_k) = \sum_{i \in T_k} c_i \leq \sum_{r:k_r=k} \sum_{i \in Z_r} c_i = \sum_{r:k_r=k} C(X_r),$$

hence

$$length(T) = \sum_{k=1}^{K} cost(T_k) \leq \sum_{k=1}^{K} \sum_{r:k_r=k} C(X_r) = \sum_{r=1}^{R} C(X_r) = cost(\{X_1, \ldots, X_R\}).$$

Therefore T is a covering subtree of G containing 0 and

$$length(T) \leq cost(\mathbf{X}).$$

□

As an easy consequence of Lemmas 5.1 and 5.2 we have

Proposition 5.1 *A subtree $T \subset G$ containing vertex 0 is a minimum-length covering tree if and only if the collection $\mathbf{Y} = \{Y_1, Y_2, \ldots, Y_K\}$ of tree sets associated with T is a minimum-cost cover of U by tree sets. Moreover the length of a minimum covering tree equals the cost of a minimum cover.* □

5.2.4 Multi-set Cover Problem

Proposition 5.1 transforms the Tree Cover problem into the Set Cover problem. There is a major difficulty though—the number of tree sets is in general exponential! Hence, at this stage, we cannot use any of the well-known algorithms for approximating Set Cover. Fortunately, we can overcome this obstacle.

Let $\{Z_\alpha\}_{\alpha \in A}$ be some enumeration of all nonempty subtrees of trees G_k with $k \in Z_\alpha$, for $k = 1, \ldots, K$. Notice that we have, in general, exponentially many subtrees. Let $\{X_\alpha\}_{\alpha \in A}$ be the corresponding tree sets (there might be possible repetitions of identical sets obtained from different subtrees) and $\{C_\alpha\}_{\alpha \in A}$ be the costs of the tree sets. Hence $X_\alpha = \bigcup_{i \in Z_\alpha} S_i$ and $C_\alpha = \sum_{i \in Z_\alpha} c_i$.

Define $b_{j\alpha}$, the multiplicity of t_j in X_α, as

$$b_{j\alpha} = \sum_{i \in Z_\alpha} a_{ji}.$$

Thus $b_{j\alpha}$ indicates how many times an element t_j is covered by the sets S_i corresponding to vertices of Z_α.

Consider the following integer program that we call the MULTI-SET COVER problem.

Problem ITCP$_2$:

$$\begin{aligned} OPT_2 \;=\; \text{minimize} \quad & \sum_{\alpha \in A} C_\alpha x_\alpha, \\ \text{subject to} \quad (a) \quad & \sum_{\alpha \in A} b_{j\alpha} x_\alpha \geq 1, \quad \text{for all } j = 1, \ldots, m, \\ (b) \quad & x_\alpha \in \{0, 1\}, \qquad \text{for all } \alpha \in A. \end{aligned} \tag{5.3}$$

In view of Lemmas 5.1 and 5.2 and Proposition 5.1, ITCP$_2$ is clearly a different formulation of the TCP (GROUP-STEINER) problem and can be considered a (non-standard!) formulation of the SET COVER problem in terms of tree sets. The following proposition is an immediate consequence of Proposition 5.1.

Proposition 5.2 *The values OPT_1 and OPT_2 of optimal solutions to $ITCP_1$ and $ITCP_2$ are the same.* □

The LP relaxations of problem ITCP$_2$ is the following.

Problem $\mathbf{LTCP_2}$:

$$
\begin{aligned}
OPT_2^* &= \text{minimize} \quad \sum_{\alpha \in A} C_\alpha x_\alpha, \\
&\text{subject to} \quad \text{(a)} \quad \sum_{\alpha \in A} b_{j\alpha} x_\alpha \geq 1, \quad \text{for all } j = 1, \ldots, m, \\
&\qquad\qquad\quad \text{(b)} \quad x_\alpha \in [0,1], \qquad\quad \text{for all } \alpha \in A.
\end{aligned}
\tag{5.4}
$$

Let $\{x_\alpha\}_{\alpha \in A} \in \mathbf{R}_+^{|A|}$. Define $\mathbf{X} = \{X_\alpha \mid x_\alpha > 0\} = (X_1, X_2, \ldots, X_L)$ and $\mathbf{x} = \{x_\alpha \mid x_\alpha > 0\} = (x_1, x_2, \ldots, x_L)$ for some $L \in \mathbf{N}$. Here, we wrote X_l instead of X_{α_l} and x_l instead of x_{α_l} to simplify the notation. We will use this shortcut for the rest of the section.

For any $t_j \in U$, define $cov_{\mathbf{x}}^m(t_j)$, the "multi-covering number" of t_j with respect to $\mathbf{x}$, as

$$
cov_{\mathbf{x}}^m(t_j) = \sum_{l=1}^{L} b_{jl} x_l = \sum_{\alpha \in A} b_{j\alpha} x_\alpha.
$$

Thus $cov_{\mathbf{x}}^m(t)$ indicates how many times t is covered by fractions x_l of tree sets $X_1, \ldots, X_L$, *including* multiplicities.

If $\{x_\alpha\}_{\alpha \in A}$ is a feasible solution to (5.4) then condition (5.4a) implies that $cov_{\mathbf{x}}^m(t_j) \geq 1$ for all $t \in U$. We call the pair $(\mathbf{X}, \mathbf{x}) = ((X_1, \ldots, X_L), (x_1, \ldots, x_L))$ a *"fractional multi-cover"* of U and define the "cost" of $(\mathbf{X}, \mathbf{x})$ as $cost^*(\mathbf{X}, \mathbf{x}) = \sum_{l=1}^{L} C(X_l) x_l$.

Note: Since $b_{j\alpha}$ can, in general, be greater than 1, problem LTCP_2 is quite different from the standard LP relaxation of the Set Cover problem. This has no effect on the integer solutions but it can decrease the value of the optimal fractional solution, thus increasing the "integrality gap". This makes our analysis more complicated and requires the introduction of additional restrictions on the magnitude of $b_{j\alpha}$.

5.2.5 Tree Stripping

Consider the following algorithm.

Algorithm "Tree-Stripping"

Input: A fractional covering tree $(T, \mathbf{y})$.

Output: A collection of tree sets $\mathbf{X} = \{X_1, \ldots, X_L\}$ with costs $\mathbf{C} = \{C_1, \ldots, C_L\}$ and a vector $\mathbf{x} = (x_1, \ldots, x_L)$ of positive rational numbers such that $length^*(T, \mathbf{y}) = \sum C_l x_l$.

begin
    $l \leftarrow 1$;
    for $k \leftarrow 1$ **to** K **do**
        if $T_k \neq \emptyset$ **then**
            $Z_l \leftarrow T_k$;
            while $(y_k > 0)$ **do**
                $X_l \leftarrow \bigcup_{i:i \in Z_l} S_i$;
                $C_l \leftarrow length(Z_l)$;
                $x_l \leftarrow$ the smallest value y_i among the nodes of Z_l;
                for all $i \in Z_l$ **do** $y_i \leftarrow y_i - x_l$;
                $l \leftarrow l + 1$;
                $Z_l \leftarrow$ subtree generated by those nodes $i \in Z_{l-1}$
                    for which $y_i > 0$;
    $L \leftarrow l - 1$;
    return $(\mathbf{X}, \mathbf{C}, \mathbf{x})$;
end

For each k, the algorithm splits tree T_k with some rational "weights" associated with its nodes into a "sum" of subtrees Z_l such that the weights of the nodes in each of these trees are the same (they all are equal to x_l)—see Figure 5.4. At each iteration l, the algorithm finds the maximum possible weight which can be subtracted from all nodes of the current tree Z_l without making any weight negative and "subtracts" a tree of this weight from Z_l. Since at each step, $y_i \leq y_{\pi(i)}$, this results either in a smaller tree (containing root k)

having at least one node less, or in an empty set. Therefore the algorithm "Tree-Stripping" terminates after at most $|T| - 1 \leq n$ iterations, and as such, runs in time polynomial in m and n and outputs $L \leq n$ tree sets. Since each Z_l is a subtree of some G_k and $k \in Z_l$, each of the sets X_l is a tree set. Clearly, $C_l = C(X_l)$.

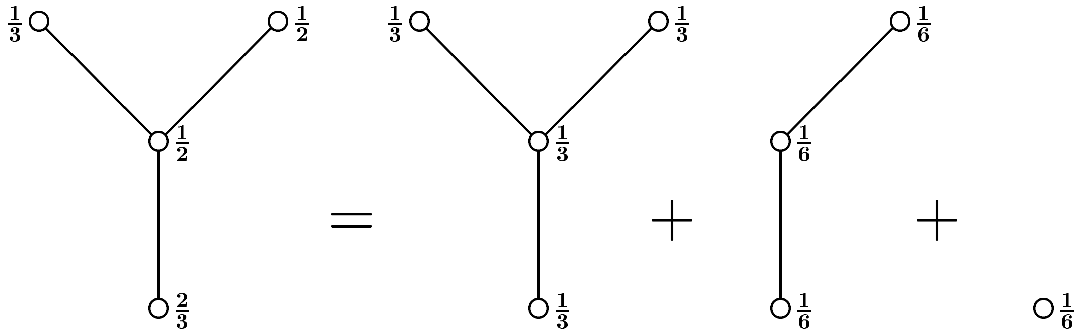


Figure 5.4: Tree-stripping

For any $t_j \in U$, we have

$$\begin{aligned} cov_{\mathbf{x}}^m(t_j) &= \sum_{l=1}^{L} b_{jl} x_l = \sum_{l=1}^{L} \sum_{i \in Z_l} a_{ji} x_l = \sum_{i \in T} a_{ji} \sum_{l : i \in Z_l} x_l \\ &= \sum_{i \in T} a_{ji} y_i = \sum_{i=1}^{n} a_{ji} y_i = cov_{\mathbf{y}}(t_j). \end{aligned}$$

Also,

$$\begin{aligned} cost^*(\mathbf{X}, \mathbf{x}) &= \sum_{l=1}^{L} C(X_l) x_l = \sum_{l=1}^{L} \sum_{i \in Z_l} c_i x_l = \sum_{i \in T} c_i \sum_{l : i \in Z_l} x_l \\ &= \sum_{i \in T} c_i y_i = \sum_{i=1}^{n} c_i y_i = length^*(T, \mathbf{y}). \end{aligned}$$

Thus we have proved the following theorem, which plays a pivotal role in developing the results of this section.

Theorem 5.1 *Given a fractional covering tree $(T, \mathbf{y})$, algorithm "Tree-Stripping" outputs a fractional multi-cover $(\mathbf{X}, \mathbf{x})$ of U of size $L \leq n$ by tree sets such that*

$$cost^*(\mathbf{X}, \mathbf{x}) = length^*(T, \mathbf{y}).$$

Moreover, for each $t \in U$,

$$cov_{\mathbf{x}}^{m}(t) = cov_{\mathbf{y}}(t).$$

□

Any fractional multi-cover $(\mathbf{X}, \mathbf{x})$ corresponds to a feasible solution of LTCP$_2$ with the same total cost—just set $x_\alpha = x_l$ if $\alpha = \alpha_l$ for some $l = 1, \ldots, L$ and $x_\alpha = 0$ otherwise. Thus for any feasible solution $\mathbf{y}$ of LTCP$_1$ there is a corresponding feasible solution $\{x_\alpha\}_{\alpha \in A}$ of LTCP$_2$ of the same value, that is $\sum C_\alpha x_\alpha = \sum c_i y_i$.

By reversing the procedure of tree-stripping, that is simply "adding" trees Z_l corresponding to tree sets with nonzero value of x_l, one can easily show that for any feasible solution $\{x_\alpha\}_{\alpha \in A}$ of LTCP$_2$, that is for any fractional multi-cover $(\mathbf{X}, \mathbf{x})$, there is a corresponding tree cover $(T, \mathbf{y})$, that is a feasible solution $\mathbf{y}$ of LTCP$_1$ such that $\sum c_i y_i = \sum C_\alpha x_\alpha$ and $cov_{\mathbf{y}}(t) = cov_{\mathbf{x}}^{m}(t)$ for any $t \in U$.

Thus we have the following.

Proposition 5.3 $(T, \mathbf{y})$ *is a minimum-length fractional covering tree if and only if the corresponding fractional multi-cover* $(\mathbf{X}, \mathbf{x})$ *is a minimum-cost fractional multi-cover.* □

Note: The idea of tree-stripping is somewhat similar to path-stripping introduced by Raghavan and Thompson in [76].

5.2.6 Approximation Algorithm and Performance Bounds

Example 5.2 Let $U = \{t\}$. Consider the tree in Figure 5.5 where $S_i = U$ for each vertex $i \in V \setminus \{0\}$. Let $\epsilon > 0$ be a very small number. We have n nonempty tree sets $X_{-1} = X_1 = \cdots = X_{n-1} = U$ determined by subtrees $T_{-1}, T_1, \ldots, T_{n-1}$ where $T_i = [1, i]$ for $i = 1, \ldots, n-1$ and $T_{-1} = \{-1\}$. The costs of tree sets are $C_i = n + (i-1)\varepsilon$ for $i = 1, \ldots, n-1$ and $C_{-1} = \sqrt{n}$. Also, $b_{1i} = b_i = i$ for $i = 1, \ldots, n-1$ and $b_{-1} = 1$.

The optimal solution to LTCP$_1$ is clearly $y_1^* = y_2^* = \cdots = y_{n-1}^* = \frac{1}{n-1}$, $y_{-1} = 0$ with value $OPT_1^* = \frac{n+(n-1)\varepsilon}{n-1} \approx 1$. Similarly, the optimal solution to LTCP$_2$ is $x_{n-1}^* = \frac{1}{n-1}$ and $x_i^* = 0$ otherwise, and its value is $OPT_2^* = \frac{n+(n-1)\varepsilon}{n-1} \approx 1$.

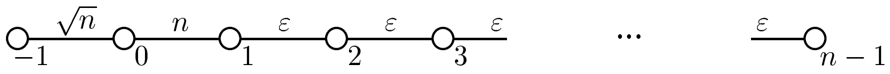


Figure 5.5: Tree from Example 1

On the other hand, the optimal solution to ITCP$_1$ is $\hat{y}_{-1} = 1$ and $\hat{y}_i = 0$ otherwise with value $OPT_1 = \sqrt{n}$. This corresponds to the optimal solution to ITCP$_2$ for which $\hat{x}_{-1} = 1$ and $\hat{x}_i = 0$ otherwise with value $OPT_2 = OPT_1 = \sqrt{n}$.

Thus the values of optimal solutions to (5.2) and (5.4) are the same (which was already guaranteed by Proposition 5.3), but the value of the optimal integer solution differs significantly from the value of the fractional solution. In fact, we can make the integrality gap as bad as $\Omega(n)$ by simply removing the node -1. Therefore the performance guarantee of any algorithm for approximating the solutions to ITCP$_1$ and ITCP$_2$ based on linear programming (randomized rounding, the primal-dual method, etc.) is, in general, poor.

And there is yet another difficulty. Assume that we have some algorithm (such as randomized rounding) which uses an optimal fractional solution to LTCP$_1$ or LTCP$_2$ to construct an approximation to the optimal integer solution of ITCP$_1$ or ITCP$_2$. This algorithm would necessarily output an approximation having cost $APP_1 \approx APP_2 \approx n$, since it would completely disregard the node -1 (the set X_{-1}). Hence, in general, not only the fractional and integer optimal solutions to TCP$_1$ or TCP$_2$ differ significantly, but also the optimal integer solution and the approximation based on fractional optimal solution can be quite different. In fact, changing the cost c_{-1} to say 2 makes the performance ratio $\Omega(n)$. $\square$

In the remainder of this section we will need the following notation. Set $Y_k = \bigcup_{i \in G_k} S_i$. Define $B = \max_{j,\alpha} b_{j\alpha} = \max\{\text{multiplicity of } t_j \text{ in } Y_k \,|\, j, k\}$ and $d = \max_\alpha |X_\alpha| = \max_k |Y_k|$. Thus B is the maximum number of locations in any subtree G_k at which any fixed errand can be completed and d is an upper bound on the size of any tree set. B can be also viewed as an upper bound on the size of $\mathcal{C}_j \cap G_k$ (that is the number of vertices in subtree G_k belonging to cluster $\mathcal{C}_j$) for all $j = 1, \ldots, m$ and $k = 1, \ldots, K$.

The previous example shows that the optimal solutions to ITCP and LTCP can have quite different values. In order for us to get reasonable worst-case performance bounds on any algorithm based on LP relaxation of the original TCP, B has to be bounded by a constant, or be a slowly growing function of n and m. In what follows, we will show that with these restrictions on B we can approximate TCP quite well.

Consider the following algorithm for approximating the TCP (Group-Steiner) problem on trees.

Algorithm "Approximate_TCP"

Input: A tree $G = (V, E)$ of "locations" with nonnegative cost on each edge and a collection $\{S_1, \ldots, S_n\}$ of subsets of U.

Output: A subtree $\hat{T} \subset G$ approximating an optimal solution to TCP on G.

begin

$(T, \mathbf{y}^*) \leftarrow$ Solve LTCP$_1$ on $(G, \mathbf{c})$;

$(\mathbf{X}, \mathbf{C}, \mathbf{x}^*) \leftarrow$ Tree-Stripping $(T, \mathbf{y}^*)$;

$\hat{\mathbf{X}} \leftarrow$ Weighted_Greedy $(\mathbf{X}, \mathbf{C})$;

$\hat{T} \leftarrow$ Cover_to_Tree $(\hat{\mathbf{X}})$;

return $\hat{T}$;

end

The algorithm Weighted_Greedy is the greedy algorithm for approximating weighted Set Cover introduced in Subsection 4.1.2. Before we state the main theorem, let us prove the following lemma.

Lemma 5.3 *Let $((X_1, \ldots, X_L), (x_1^*, \ldots, x_L^*), (C_1, \ldots, C_L))$ be a minimum-cost fractional multi-cover of U by tree sets of cost OPT_2^*. The cover $\hat{\mathbf{X}}$ output by the greedy algorithm* *with input* $((X_1, \ldots, X_L), (C_1, \ldots, C_l))$ *satisfies*

$$cost(\hat{\mathbf{X}}) \leq OPT_2^* BH(d).$$

Proof: A special case of this lemma for $B = 1$ was proved by Chvátal in [24] and we briefly discussed it in Subsection 4.1.2. Let us restate Chvátal's result in a form convenient for this section. Let $(\bar{a}_{jl})$ be the $m \,\mathrm{x}\, L$ incidence matrix of the set cover $(X_1, \ldots, X_L)$, that is $\bar{a}_{jl} = 1$ if $t_j \in X_l$, and $\bar{a}_{jl} = 0$ otherwise. Let $\{l_1, l_2, \ldots, l_R\}$ be the indices of the sets in the subcover output by algorithm Weighted_Greedy. Chvátal proved that for any L-tuple $(z_1, \ldots, z_L)$ of nonnegative numbers satisfying $\sum_{l=1}^{L} \bar{a}_{jl} z_l \geq 1$ for all $j = 1, \ldots, m$, it must be the case that

$$\sum_{l=1}^{L} H(\sum_{j=1}^{m} \bar{a}_{jl}) C_l z_l \geq \sum_{r=1}^{R} C_{l_r} = cost(\hat{\mathbf{X}}).$$

Since $\sum_j \bar{a}_{jl} = |X_l| \leq d$, we have

$$H(d) \sum_{l=1}^{L} C_l z_l \geq cost(\hat{\mathbf{X}}).$$

We can use Chvátal's result to prove the lemma. In our case, $\bar{a}_{jl} = 1$ if $b_{jl} \geq 1$ and $\bar{a}_{jl} = 0$ if $b_{jl} = 0$. Hence $B\bar{a}_{jl} \geq b_{jl}$. Set $z_l = Bx_l^*$. Then for any $j = 1, \ldots, m$

$$\sum_{l=1}^{L} \bar{a}_{jl} z_l = \sum_{l=1}^{L} \bar{a}_{jl} B x_l^* \geq \sum_{l=1}^{L} b_{jl} x_l^* \geq 1$$

since $(x_1^*, \ldots, x_L^*)$ is a feasible solution to LTCP_2. Thus Chvátal's result implies that

$$H(d) \sum_{l=1}^{L} C_l z_l \geq cost(\hat{\mathbf{X}})$$

that is

$$BH(d) \sum_{l=1}^{L} C_l x_l^* \geq cost(\hat{\mathbf{X}}).$$

Therefore

$$BH(d) OPT_2^* \geq cost(\hat{\mathbf{X}}).$$

□

Theorem 5.2 *The algorithm "Approximate_TCP" runs in time polynomial in m and n, and its performance ratio is no worse than $BH(d)$. Therefore, for any instance of the* TCP *(*GROUP-STEINER*) problem on trees, the length APP of the approximate covering tree (group-Steiner tree) output by the algorithm satisfies*

$$\frac{APP}{OPT} \leq BH(d).$$

Proof: The problem LTCP$_1$ can be solved in time polynomial in m and n using any of the well-known polynomial-time algorithms (ellipsoid method, Karmarkar's method, etc.). Since $R \leq L \leq n$, algorithms Weighted_Greedy and Cover_to_Tree also run in time polynomial in m and n. Thus the total running time of Approximate_TCP is bounded by a polynomial in m and n.

Combining the above results, we easily obtain

$$APP = length(\hat{T}) \leq cost(\hat{\mathbf{X}}) \leq OPT_2^* BH(d) = OPT_1^* BH(d) \leq OPT BH(d).$$

□

We now give performance bounds for some specific cases of TCP (GROUP-STEINER).

Corollary 5.1 *Let tree G be such that the sets S_i in each of the subtrees G_k are disjoint; that is, each cluster $\mathcal{C}_j$ intersects each subtree G_k in at most one point. Then the performance ratio of the algorithm Approximate_*TCP *for approximating* GROUP-STEINER *(*TCP*) satisfies*

$$\frac{APP}{OPT} \leq H(m).$$

Proof: The fact that for each $k = 1, \ldots, K$ the sets $\{S_i \mid i \in G_k\}$ are disjoint implies that $B = 1$. □

Corollary 5.2 *Let $B = O(1)$. Then we can approximate* GROUP-STEINER *(*TCP*) within $O(\log m)$.* □

Corollary 5.3 *Let G be a tree such that the number of nodes in each subtree G_k is bounded by some β (possibly a function of m and n). Then the performance ratio of algorithm Approximate*_TCP *is no worse than $\beta H(d) \leq \beta H(m)$. In particular, when $\beta = 1$, we have*

$$\frac{APP}{OPT} \leq H(d).$$

Thus, in this special case of TCP *corresponding to weighted set cover, our algorithm has the same worst-case performance guarantee as the usual greedy algorithm for weighted* SET COVER. □

Note: One can further improve the performance of our algorithm in the following way. Choose a subtree $Z \subset G$ with $0 \in Z$ having $O(\log n)$ nodes. The graph $G \setminus Z$ is a forest of trees G'_k, $k = 1 \ldots K'$. Now, for each of $O(n)$ possible subtrees $Z_s \subset Z$, $0 \in Z_s$, do the following. Create a "super-root" R_s consisting of all the nodes of Z_s. Then consider a new graph G' obtained as a union of the trees G'_i adjacent to Z_s and the super-root R_s. This graph is again a tree. Set $U' = U \setminus \bigcup_{i \in Z_s} S_i$ and $S'_i = S_i \cap U'$ for all $i \in G'$. Thus we have a simpler instance of the TCP problem and can use our algorithm Approximate_TCP to find a good approximation. We have a total of $O(n)$ subproblems, hence the total running time will be $O(n)$-times the time for Approximate_TCP. This method is particularly useful when the degree of root 0 is small.

Note: As discussed at the beginning of this section, algorithm Approximate_TCP can be also used to approximate GTSP (ERRAND SCHEDULING) on trees with exactly the same performance bounds.

5.3 GTSP on Graphs with Bounded Cluster Size

In this and the following sections we consider GTSP and GROUP-STEINER on an edge-weighted general graph $G = (V, E)$. We will assume that each cluster $\mathcal{C}_j$ has size at most ρ, that is $\rho = \max_j\{\sum_i a_{ji}\}$. We first concentrate on approximating the GENERALIZED TRAVELING SALESMAN problem. Here, the fact that each cluster is of size at most ρ simply means that there are at most ρ cities in each group or (for ESP) that each task can be completed at at most ρ locations.

Let G' be a complete graph on V with a shortest-path metric inherited from G. Then for any partial tour with repeated vertices (sometimes called a "closed partial walk") through some $V' \subset V$ in G, there is a "simple" partial tour (without repeated vertices) through V' in G' at most that long. On the other hand, given a simple partial tour through some $V' \subset V$ in G', one can easily construct a partial tour (possibly with repeated vertices) in G of the same length visiting all V' (and possibly some more vertices). Hence, without loss of generality, we will assume that $G = (V, E)$ is a complete undirected graph and the distance function $l_e = l_{(i,j)}$ is symmetric and satisfies the triangle inequality. Also, all the tours considered are simple tours.

Denote by OPT_{3+} the length of a minimum generalized tour consisting of at least three vertices, and OPT_2 the minimum length of a generalized "tour" consisting of vertex 0 and one other vertex (provided that such a "tour" exists), that is $OPT_2 = \min\{2l_{(0,i)} \mid S_i = U\}$. Then, clearly, $OPT = \min\{OPT_2, OPT_{3+}\}$. We can find the exact value of OPT_2 in $O(nm)$ time. Thus, in what follows, we will concentrate on approximating the value OPT_{3+}. Set $\tilde{V} = \bigcup_{j=0}^{m} \mathcal{C}_j$ and $N = |\tilde{V}|$. Thus $\tilde{V}$ is the set of all points belonging to some cluster. Since the clustering is not necessarily exhaustive, in general $\tilde{V} \subset V$.

For a given partial tour, let $y_i = 1$ if vertex i is in the tour and $y_i = 0$ otherwise. Similarly, let $x_e = 1$ if the edge e is in the tour and $x_e = 0$ otherwise. For any $S \subset V$ let $\delta(S)$ be the edges from the cut $(S, V \setminus S)$, that is all the edges with one endpoint in S and the other endpoint in $V \setminus S$. Also, denote by $E(S)$ the set of all edges whose both end-points belong to S.

The problem of finding OPT_{3+} can be formulated as the following integer program:

Problem $\mathbf{IP}_1$:

$$
\begin{aligned}
Z_1 \quad = \quad \text{minimize} \quad & \sum_{e \in E} l_e x_e, \\
\text{subject to} \quad (a) \quad & \sum_{e \in \delta(\{i\})} x_e = 2y_i, \quad \text{for all } i \in V, \\
(b) \quad & \sum_{e \in \delta(S)} x_e \geq 2y_i, \quad \text{for all } S \subset V,\ 0 \notin S, \text{ and all } i \in S, \\
(c) \quad & \sum_{i=1}^{n} a_{ji} y_i \geq 1, \quad \text{for all } j = 1, \ldots, m, \\
(d) \quad & y_o = 1, \\
(e) \quad & y_i \in \{0,1\}, \quad \text{for all } i \in V, \\
(f) \quad & x_e \in \{0,1\}, \quad \text{for all } e \in E.
\end{aligned}
\tag{5.5}
$$

Condition (a) guarantees that for every vertex in the tour, there are exactly two edges incident on it. Condition (b) prevents subtours, that is the tour cannot be made of two disjoint subtours. Condition (c) guarantees that each cluster gets visited by the tour at least once. Condition (d) simply assures that the vertex 0 is in the tour. Thus it is easy to see that Problem IP_1 is equivalent to GTSP—assuming that the optimum subtour consists of at least three vertices.

5.3.1 Approximation Algorithm

First, let us formulate problem LP_1, the LP relaxation of problem IP_1. In order to do that, we simply replace conditions (e) and (f) in IP_1 by new conditions (e') and (f'), where

$$
\begin{aligned}
(e') \quad & 0 \leq y_i \leq 1, \quad \text{for all } i \in V, \\
(f') \quad & 0 \leq x_e \leq 1, \quad \text{for all } e \in E.
\end{aligned}
$$

The algorithm for approximating the optimal solution to GTSP is as follows:

Algorithm "Approximate_GTSP"

Input: A complete graph $G = (V, E)$ of "locations" with non-negative symmetric distance function on edges satisfying the triangle inequality, and clusters $\mathcal{C}_o, \ldots, \mathcal{C}_m$ such that $\mathcal{C}_o = \{0\}$.

Output: A partial tour through some set $W \subset V$ approximating the optimal solution to GTSP.

1. Find $APP_2 = \min\{2l_{0,i} \mid i \in V,\ S_i = U\}(= OPT_2)$.

2. Solve problem LP_1 - the LP relaxation of problem IP_1. Let $(\mathbf{y}, \mathbf{x}, Z_1^*) = ((y_i^*)_{i=1}^n, (x_e^*)_{e\in E}, Z_1^*)$ be the optimal solution.

3. Use the Christofides algorithm for approximating TSP to find a tour through all the vertices of the set $W = \{i \in V \mid y_i^* \geq 1/\rho\}$. Set APP_{3+} to be the length of such a tour.

4. Output $APP = \min\{APP_2, APP_{3+}\}$, and the shorter tour.

Step 1 can be done in polynomial time. Step 2 can be done in polynomial time using the ellipsoid method. Step 3 can also be done in polynomial time. We can also assume that $W \subset \tilde{V}$.

5.3.2 Auxiliary Results

In order to establish upper bounds on the performance ratio of the above algorithm, we now present some auxiliary results.

Consider the following problem.

Problem IP$_2$:

$$
\begin{aligned}
Z_2 \quad = \quad \text{minimize} \quad & \sum_{e \in E} l_e x_e, \\
\text{subject to} \quad (a) \quad & \sum_{e \in \delta(\{i\})} x_e = 2, \quad \text{for all } i \in V, \\
(b) \quad & \sum_{e \in \delta(S)} x_e \geq 2, \quad \text{for all } S \subset V,\ S \neq \emptyset,\ S \neq V, \\
(c) \quad & x_e \in \{0,1\}, \quad \text{for all } e \in E.
\end{aligned}
\tag{5.6}
$$

Clearly, it is the integer programming formulation of the classical TSP. Let LP$_2$ be the LP relaxation of IP$_2$, that is we simply replace the constraint (c) by a new constraint (c'), where

$$(c') \quad 0 \leq x_e \leq 1, \quad \text{for all } e \in E.$$

Denote by Z_2^* the value of the optimal solution to LP$_2$.

For any subset S of V, denote by $L(S)$ the length of the optimal traveling salesman tour through vertices in S and by $L^C(S)$ the length of the tour passing through all vertices of S obtained by the well-known Christofides algorithm. Also denote by $L^T(S)$ the length of the minimum spanning tree on S and for $|S|$ even denote by $L^M(S)$ the length of the minimum-weight perfect matching on S. Thus $L(V) = Z_2$ and $L^C(V) \leq L^T(V) + L^M(V')$, where V' is the set of odd degree vertices of the minimum spanning tree.

Proposition 5.4 (Wolsey, [95], Shmoys and Williamson, [85])

$$L(V) \leq L^C(V) \leq \frac{3}{2} Z_2^*$$

Proof: We will outline the proof given by Wolsey. Since $L(V) \leq L^C(V)$, it is enough to show the second inequality. The proof consists of two parts. We will only show that $L^T(V) \leq Z_2^*$. The fact that $L^M(V') \leq Z_2^*/2$ given that V' is a set of odd degree vertices of some minimum spanning tree on V can be proved using similar techniques. The result then follows.

Let S be a proper nonempty subset of V. Denote by $\bar{S}$ the complement of S in V, that is $\bar{S} = V \setminus S$. Equation (5.6a) implies that

$$2 \sum_{e \in E(\bar{S})} x_e + \sum_{e \in \delta(S)} x_e = 2|\bar{S}|.$$

Adding condition (5.6b) and dividing by 2 gives us

$$\sum_{e \in E(\bar{S})} x_e + \sum_{e \in \delta(S)} x_e \geq |\bar{S}| + 1.$$

Since $|\bar{S}| + 1 \geq |\bar{S}| = |V| - |S| = n - |S|$, the problem

$$\begin{array}{rcll} Z_R^* & = & \text{minimize} & \displaystyle\sum_{e \in E} l_e x_e, \\ & & \text{subject to} & (a) \quad \displaystyle\sum_{e \notin E(S)} x_e \geq n - |S|, \quad \text{for all } S \subset V,\, S \neq \emptyset,\, S \neq V, \\ & & & (b) \quad x_e \geq 0, \qquad\qquad\quad\;\; \text{for all } e \in E \end{array} \tag{5.7}$$

is a valid relaxation of LP_2. Combining the results of Fulkerson [33, 34], Edmonds [31, 29], and Tutte [94], we see that the extremal points of the polyhedra determined by conditions (5.7a) and (5.7b) are spanning trees. In particular, the optimal solution to (5.7) is a minimum spanning tree on V, hence $L^T(V) = Z_R^*$. And since the polyhedron of the relaxed linear program is in general larger than the original one, it must be that $Z_R^* \leq Z_2^*$. Therefore, $L^T(V) \leq Z_2^*$. □

Let $V_2 \subset V$. Define

$$r_i = \begin{cases} 2 & \text{if } i \in V_2, \\ 0 & \text{otherwise.} \end{cases}$$

Consider the following LP problem.

Problem LP$_3$:

$$
\begin{aligned}
Z_3^* \quad = \quad & \text{minimize} \quad \sum_{e\in E} l_e x_e, \\
& \text{subject to} \quad (a) \quad \sum_{e\in\delta(\{i\})} x_e = r_i, \quad \text{for all } i \in V, \\
& \qquad\qquad\quad (b) \quad \sum_{e\in\delta(S)} x_e \geq 2, \quad \text{for all } S \subset V \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{such that } V_2 \cap S \neq \emptyset \neq V_2 \setminus S, \\
& \qquad\qquad\quad (c) \quad 0 \leq x_e \leq 1, \quad \text{for all } e \in E.
\end{aligned}
\tag{5.8}
$$

We can easily modify Proposition 5.4 to obtain

Proposition 5.5 (Bienstock, Goemans, Simchi-Levi, and Williamson, [18])

$$L(V_2) \leq L^C(V_2) \leq \frac{3}{2} Z_3^*$$

Proof: Since for any feasible solution to (5.8), $e \notin E(V_2)$ implies $x_e = 0$, we can use Proposition 5.4 to prove the inequality. □

Let us consider the following relaxation of problem LP$_3$.

Problem LP$_4$:

$$
\begin{aligned}
Z_4^* \quad = \quad & \text{minimize} \quad \sum_{e\in E} l_e x_e, \\
& \text{subject to} \quad (b) \quad \sum_{e\in\delta(S)} x_e \geq 2, \quad \text{for all } S \subset V \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{such that } V_2 \cap S \neq \emptyset \neq V_2 \setminus S, \\
& \qquad\qquad\quad (c) \quad 0 \leq x_e, \quad \text{for all } e \in E.
\end{aligned}
\tag{5.9}
$$

Thus we omitted constraint (a) and relaxed constraint (c).

Lemma 5.4 (Goemans, Bertsimas, [41], Bienstock, Goemans, Simchi-Levi, Williamson, [18]) *The optimal solution values to problems LP_3 and LP_4 are the same, that is*

$$Z_3^* = Z_4^*.$$

□

The proof of Lemma 5.4 is rather non-trivial and relies on the following result of Lovász [65, exercise 6.51].

Lemma 5.5 (Lovász, [65]) *Let G be an Eulerian multigraph and $s \in V(G)$, such that G is k-connected between any two vertices different from s. Then, for any neighbor u of s, there exists another neighbor w of s, such that the multigraph obtained from G by removing edges (s,u) and (s,w), and adding a new edge (u,w) is also k-connected between any two vertices different from s.* □

5.3.3 Performance Bounds

Let $(\mathbf{y}, \mathbf{x}, Z_1^*) = ((y_i^*)_{i=1}^n, (x_e^*)_{e\in E}, Z_1^*)$ be the optimal solution to problem LP_1. Define

$$\begin{aligned} \hat{x}_e &= \rho x_e^* \\ \hat{y}_i &= \begin{cases} 1 & \text{if } y_i^* \geq \frac{1}{\rho} \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

Then the set W of vertices selected by algorithm Approximate_GTSP can be expressed as

$$W = \{i \in V \mid y_i^* \geq 1/\rho\} = \{i \in V \mid \hat{y}_i = 1\}.$$

Lemma 5.6 *For all $j = 0, \ldots, m$, we have $\mathcal{C}_j \cap W \neq \emptyset$, i.e. set W contains at least one vertex from each cluster.*

Proof: Let j be arbitrary. We know that $i \in \mathcal{C}_j$ iff $a_{ji} = 1$. Since $|\mathcal{C}_j| \leq \rho$, $a_{ji} = 1$ for at most ρ indices i. Condition (5.5c) implies that

$$1 \leq \sum_{i=1}^n a_{ji} y_i^* = \sum_{i \in \mathcal{C}_j} y_i^*,$$

hence $y_{i_o}^* \geq 1/\rho$ for some index $i_o \in \mathcal{C}_j$, thus $\hat{y}_{i_o} = 1$, and $i_o \in W$. □

Since LP_1 is the LP relaxation of IP_1, we have

$$Z_1^* \leq Z_1.$$

Set $V_2 = W$ and $r_i = 2\hat{y}_i$ in problems LP$_3$ and LP$_4$, and use Proposition 5.5 and Lemma 5.4 to obtain

$$L^C(W) \leq \frac{3}{2}Z_3^* = \frac{3}{2}Z_4^*. \tag{5.10}$$

Now let us show that $(\hat{x}_e)_{e\in E}$ is a feasible solution to LP$_4$. Indeed, $\hat{x}_e \geq 0$ for all $e \in E$, hence condition (c) is satisfied. Let $S \subset V$ be such that $W \cap S \neq \emptyset \neq (W \setminus S)$. Without loss of generality assume that $0 \in W \setminus S$ and choose some $i \in W \cap S$. Hence $\hat{y}_i = 1$ and $y_i^* \geq \frac{1}{\rho}$. Then we have

$$\sum_{e\in\delta(S)} \hat{x}_e = \rho \sum_{e\in\delta(S)} x_e^* \geq \rho 2 y_i^* \geq \rho 2\frac{1}{\rho} = 2$$

by the definition of $\hat{x}_e$ and the fact that the x_e^* are a solution to LP$_1$. Hence the $\hat{x}_e$ satisfy constraint (b) in LP$_4$.

Therefore,

$$APP_{3+} = L^C(W) \leq \frac{3}{2}Z_4^* \leq \frac{3}{2}\sum_{e\in E} l_e\hat{x}_e = \frac{3}{2}\rho\sum_{e\in E} l_e x_e^* = \frac{3}{2}\rho Z_1^* \leq \frac{3}{2}\rho Z_1 = \frac{3}{2}\rho \cdot OPT_{3+}.$$

Thus we have proved the following:

Theorem 5.3 *The performance ratio of the algorithm "Approximate_GTSP" for approximating the optimum solution to the* GENERALIZED TRAVELING SALESMAN *problem satisfies*

$$\frac{APP}{OPT} \leq \frac{3}{2}\rho.$$

□

Note: If the sets S_i are disjoint, that is $\rho = 1$, we get the classical TSP performance bound.

Note: One can easily generalize the algorithm and its analysis to the case when, in addition to distances between vertices, there is a cost associated with each vertex. In fact, we can show that, in this case,

$$APP = L_{APP} + C_{APP} \leq \rho(3/2L_{OPT} + C_{OPT}) \leq (3\rho/2)(L_{OPT} + C_{OPT}) \leq (3\rho/2)OPT.$$

Note: We can slightly improve the result of Theorem 5.3. Since

$$|\bar{S}|+1 = \frac{|\bar{S}|+1}{|\bar{S}|}|\bar{S}| \geq \frac{|V|}{|V|-1}|\bar{S}|,$$

one can show that $L^T(V) \leq (1-\frac{1}{|V|})Z_2^*$, i.e. $L^C(V_2) \leq (\frac{3}{2}-\frac{1}{|V_2|})Z_3^*$, and since $W \subset \tilde{V}$, we get

$$\frac{APP}{OPT} \leq (\frac{3}{2}-\frac{1}{N})\rho,$$

where $N = |\tilde{V}| \leq n+1 = |V|$ is the number of vertices belonging to at least one cluster.

5.4 Group-Steiner on Graphs with Bounded Cluster Size

Let us assume again that each cluster $\mathcal{C}_j$ has size at most ρ. The assumption that the cluster size is bounded is quite natural for applications in VLSI design. It simply means that the number of ports from each terminal is bounded by a constant. Without loss of generality, we will assume that $G = (V, E)$ is a complete undirected graph and that the distance function $l_e = l_{(i,j)}$ is symmetric and satisfies the triangle inequality. As before, $\tilde{V}$ will denote the set of all vertices belonging to some cluster, that is $\tilde{V} = \bigcup_{j=0}^{m} \mathcal{C}_j$.

For a given subgraph G' of G let $y_i = 1$ if vertex i is in the subgraph and $y_i = 0$ otherwise. Similarly, let $x_e = 1$ if edge e is in the subgraph and $x_e = 0$ otherwise. Consider now the following integer program:

Problem $\mathbf{IGS_1}$:

$$Z_1 \quad = \quad \text{minimize} \quad \sum_{e \in E} l_e x_e,$$

$$\begin{aligned}
&(a) \quad \sum_{e \in \delta(S)} x_e \geq y_i, \ \text{ for all } S \subset V,\ 0 \notin S, \text{ and all } i \in S, \\
&(b) \quad \sum_{i=1}^{n} a_{ji} y_i \geq 1, \ \text{ for all } j = 1, \ldots, m, \\
&(c) \quad y_o = 1, \\
&(d) \quad y_i \in \{0,1\}, \qquad \text{for all } i \in V, \\
&(e) \quad x_e \in \{0,1\}, \qquad \text{for all } e \in E.
\end{aligned} \tag{5.11}$$

Condition (a) guarantees that any feasible solution is a connected subgraph. Condition (b) guarantees that a feasible solution contains at least one vertex from each cluster. Condition (c) simply assures that a feasible solution contains vertex 0. Thus the optimal solution to IGS_1 is simply a smallest connected subgraph which contains at least one vertex from each cluster. Clearly an optimal group-Steiner tree is also an optimal solution to IGS_1. On the other hand any feasible solution G' to IGS_1 can be turned into a group-Steiner tree of smaller or equal cost by simply breaking up any cycles in G'.

5.4.1 Approximation Algorithm

First, let us formulate problem LGS_1, the LP relaxation of problem IGS_1. In order to do that, we simply replace conditions (d) and (e) in IGS_1 by new conditions (d') and (e'), where

$$\begin{aligned}
&(d') \quad 0 \leq y_i \leq 1, \quad \text{for all } i \in V, \\
&(e') \quad 0 \leq x_e \leq 1, \quad \text{for all } e \in E.
\end{aligned}$$

The algorithm for approximating the optimal solution to GROUP-STEINER is as follows:

Algorithm "Approximate_Group-Steiner"

Input: A complete graph $G = (V, E)$ with non-negative symmetric distance function on edges satisfying the triangle inequality, and clusters $\mathcal{C}_o, \ldots, \mathcal{C}_m$ such that $\mathcal{C}_o = \{0\}$.

Output: A tree $T \subset G$ spanning some set $W \subset V$ that approximates the optimal solution to Group-Steiner.

1. Solve problem LGS_1 - the LP relaxation of problem IGS_1. Let $(\mathbf{y}, \mathbf{x}, Z_1^*) = ((y_i^*)_{i=1}^n, (x_e^*)_{e \in E}, Z_1^*)$ be the optimal solution.

2. Set $W = \{i \in V \mid y_i^* \geq 1/\rho\} \cap \tilde{V}$ and find a minimum spanning tree $T \subset G$ on the subgraph G' generated by W.

3. Output $APP = length(T)$ and the tree T.

Even though the problem LGS_1 has exponentially many constraints, it can still be solved in polynomial time using the ellipsoid method with a *min-cut—max-flow polynomial-time oracle*. The interested reader can find details in [44]. Step 2 can also be done in polynomial time as discussed in Subsection 2.5.2.

5.4.2 Auxiliary Results

In order to establish upper bounds on the performance ratio of the above algorithm, we now present some auxiliary results.

Consider the following problem.

Problem IGS$_2$:

$$
Z_2 = \text{minimize} \sum_{e \in E} l_e x_e,
$$

$$
\begin{aligned}
&(a) \quad \sum_{e \in \delta(S)} x_e \geq 1, \quad \text{for all } S \subset V,\ S \neq \emptyset,\ S \neq V, \\
&(b) \quad x_e \in \{0,1\}, \qquad \text{for all } e \in E.
\end{aligned} \tag{5.12}
$$

Clearly, it is the integer programming formulation of the MST problem (minimum connected subgraph problem). Let LGS$_2$ be the LP relaxation of IGS$_2$, that is we simply replace the constraint (b) by a new constraint (b'), where

$$
(b') \quad 0 \leq x_e \leq 1, \quad \text{for all } e \in E.
$$

Denote by Z_2^* the value of the optimal solution to LGS$_2$.

We can modify Proposition 5.4, to obtain the following.

Proposition 5.6

$$
L^T(V) \leq (2 - \frac{2}{|V|}) Z_2^*
$$

Proof: Similarly as in the proof of Proposition 5.4, we obtain that

$$
2 \sum_{e \in E(\bar{S})} x_e + \sum_{e \in \delta(S)} x_e \geq |\bar{S}|
$$

for any nonempty proper subset S of V. Adding condition (5.12a) gives

$$
2 \sum_{e \in E(\bar{S})} x_e + 2 \sum_{e \in \delta(S)} x_e \geq |\bar{S}| + 1.
$$

Now, $|\bar{S}| + 1 = \frac{|\bar{S}|+1}{|\bar{S}|}|\bar{S}| \geq \frac{|V|}{|V|-1}|\bar{S}| = \frac{|V|}{|V|-1}(|V| - |S|)$, hence the problem

$$
\begin{aligned}
&\text{minimize} \quad \sum_{e \in E} l_e x_e, \\
&\text{subject to} \quad (a) \quad \sum_{e \notin E(S)} x_e \geq \frac{|V|}{2(|V|-1)}(|V| - |S|), \quad \text{for all } S \subset V,\ S \neq \emptyset,\ S \neq V, \\
&\qquad\qquad\quad (b) \quad x_e \geq 0, \qquad\qquad\qquad\qquad\qquad \text{for all } e \in E
\end{aligned} \tag{5.13}
$$

is a valid relaxation of LGS_2. Hence, as in the proof of Proposition 5.4, we can argue that the extremal points of the polyhedra determined by conditions (5.13a) and (5.13b) are $\frac{|V|}{2(|V|-1)}$ multiples of spanning trees. Therefore the minimum spanning tree on V is no longer than $\frac{2(|V|-1)}{|V|} Z_2^*$. □

Let $V_2 \subset V$. Consider the following linear program.

Problem LGS$_3$:

$$Z_3^* = \text{minimize} \quad \sum_{e \in E} l_e x_e,$$

$$\begin{aligned} \text{subject to} \quad (a) \quad & \sum_{e \in \delta(\{i\})} x_e = 0, && \text{for all } i \in V \setminus V_2, \\ (b) \quad & \sum_{e \in \delta(S)} x_e \geq 1, && \text{for all } S \subset V \\ & && \text{such that } V_2 \cap S \neq \emptyset \neq V_2 \setminus S, \\ (c) \quad & 0 \leq x_e \leq 1, && \text{for all } e \in E. \end{aligned} \tag{5.14}$$

We can easily modify Proposition 5.6 to obtain

Proposition 5.7

$$L^T(W) \leq (2 - \frac{2}{|V_2|}) Z_3^*.$$

Proof: Since for any feasible solution to (5.14), $e \notin E(V_2)$ implies $x_e = 0$, we can use Proposition 5.6 to prove the inequality. □

Let us consider the following relaxation of problem LGS_3.

Problem LGS$_4$:

$$Z_4^* = \text{minimize} \quad \sum_{e \in E} l_e x_e,$$

$$\begin{aligned} \text{subject to} \quad (b) \quad & \sum_{e \in \delta(S)} x_e \geq 1, && \text{for all } S \subset V \\ & && \text{such that } V_2 \cap S \neq \emptyset \neq V_2 \setminus S, \\ (c) \quad & 0 \leq x_e, && \text{for all } e \in E. \end{aligned} \tag{5.15}$$

Thus we omitted constraint (a) and relaxed constraint (c). Notice that the optimal V_2-Steiner tree on V (with V_2 being the set of terminals) is a feasible solution to (5.15).

The following is a straightforward consequence of Lemma 5.4

Lemma 5.7 *The optimal solution values to problems LGS_3 and LGS_4 are the same, that is*

$$Z_3^* = Z_4^*.$$

□

5.4.3 Performance Bounds

Let $(\mathbf{y}, \mathbf{x}, Z_1^*) = ((y_i^*)_{i=1}^n, (x_e^*)_{e\in E}, Z_1^*)$ be the optimal solution to problem LGS_1. Define

$$\begin{aligned} \hat{x}_e &= \rho x_e^* \\ \hat{y}_i &= \begin{cases} 1 & \text{if } y_i^* \geq \frac{1}{\rho} \text{ and } i \in \tilde{V}, \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

Then

$$W = \{i \in V \mid y_i^* \geq 1/\rho\} \cap \tilde{V} = \{i \in V \mid \hat{y}_i = 1\}.$$

As in the previous section, we easily obtain the following.

Lemma 5.8 *For all $j = 0, \ldots, m$, we have $\mathcal{C}_j \cap W \neq \emptyset$, i.e. set W contains at least one vertex from each cluster.* □

Since LGS_1 is the LP relaxation of IGS_1, we have

$$Z_1^* \leq Z_1.$$

Now let us show that $(\hat{x}_e)_{e\in E}$ is a feasible solution to LGS_4. Indeed, $\hat{x}_e \geq 0$ for all $e \in E$, hence condition (c) is satisfied. Let $S \subset V$ be such that $W \cap S \neq \emptyset \neq (W \setminus S)$. Without loss of generality assume that $0 \in W \setminus S$ and choose some $i \in W \cap S$. Hence $\hat{y}_i = 1$ and $y_i^* \geq \frac{1}{\rho}$. Then we have

$$\sum_{e\in\delta(S)} \hat{x}_e = \rho \sum_{e\in\delta(S)} x_e^* \geq \rho y_i^* \geq \rho \frac{1}{\rho} = 1$$

by the definition of $\hat{x}_e$ and the fact that the x_e^* are a solution to LGS_1. Hence the $\hat{x}_e$ satisfy constraint (b) in LGS_4.

Therefore,

$$\begin{aligned} APP &= L^T(W) \leq (2 - \frac{2}{|W|})Z_4^* \leq (2 - \frac{2}{|W|}) \sum_{e \in E} l_e \hat{x}_e \\ &= (2 - \frac{2}{|W|})\rho \sum_{e \in E} l_e x_e^* = (2 - \frac{2}{|W|})\rho Z_1^* \\ &\leq (2 - \frac{2}{|W|})\rho Z_1 = (2 - \frac{2}{|W|})\rho\, OPT. \end{aligned}$$

And since $W \subset \tilde{V}$, that is $|W| \leq |\tilde{V}| = N$, we have proved the following:

Theorem 5.4 *The performance ratio of the algorithm "Approximate_GROUP-STEINER" for approximating the optimum solution to the* GROUP STEINER TREE *problem satisfies*

$$\frac{APP}{OPT} \leq (2 - \frac{2}{N})\rho.$$

□

Note: One can easily generalize the algorithm and its analysis to the case when, in addition to distances between vertices, there is a cost associated with each vertex. In fact, we can show that, in this case,

$$\begin{aligned} APP &= L_{APP} + C_{APP} \leq \rho(2 - \frac{2}{N})L_{OPT} + \rho\, C_{OPT}) \\ &\leq \rho(2 - \frac{2}{N})(L_{OPT} + C_{OPT}) \leq \rho(2 - \frac{2}{N})OPT. \end{aligned}$$

Note: If the clusters have size 1, we get the classical STEINER TREE problem. In that case, the set $W = \tilde{V}$ is the set of all terminals. Our analysis shows that the optimum Steiner tree can be approximated by a minimum spanning tree on terminals with performance ratio $(2 - \frac{2}{N})$—a result proved before independently by several authors (see e.g. [42]).

Note: Since we need only one vertex from each cluster, we can modify our algorithm slightly by first deleting extra vertices from W and then constructing the MST. This will assure that $|W| \leq m+1$, hence the GROUP-STEINER problem can be approximated to within $(2-\frac{2}{M})\rho$, where $M = m+1$ is the total number of clusters.

Note: Another possibility for improvement of the performance of our algorithm lies in replacing the MST algorithm by some of the existing algorithms for approximating STEINER TREE. This however would not effect the worst-case performance.

5.5 Open Problems

An obvious open problem is to improve any of the performance bounds obtained in this chapter. In particular, strong performance bounds for approximating GTSP or GROUP-STEINER on trees that would not depend on the number of vertices from the same cluster in the subtrees are of independent interest. Such bounds could be combined with results of Garg, Konjevod, and Ravi [39] to obtain a good algorithm for approximating general GTSP or GROUP-STEINER.

Another interesting direction in further improvement of our results is to reduce the running times of the algorithms. As we mentioned before, our algorithms are not practical, since they heavily rely on optimum solutions to linear programming problems. In particular, the algorithms from Sections 5.3 and 5.4 are very impractical despite their good worst-case performance since the number of constraints in the corresponding linear programs depends exponentially on the size of the problem and hence complicated slow algorithms have to be used. However, as mentioned in Section 5.4, the version of GROUP-STEINER where the size of clusters is bounded is very useful for practical applications in the VLSI design, hence improving the running time of the algorithm "Approximate_Group_Steiner" while retaining its performance is of practical interest.

In practical applications, it is often enough to consider GTSP and GROUP-STEINER in plane with Euclidean (l_2) or Manhattan (l_1) metrics. It would be of independent interest to design algorithms with stronger performance bounds and/or faster running times for approximating these restricted versions of both GTSP and GROUP-STEINER.

An interesting open problem (though, most likely, a very difficult one) would be to generalize our results to the case of directed graphs, that is to consider an asymmetric distance function. This has an interesting application in routing of postal vans in the city with possible one-way streets—see [59] for details. Here one tries to design a route for a postal van that collects mail from mailboxes located at corners of streets. Each mailbox corresponds to a cluster of size 2 since the van can park near the mailbox in exactly two possible directions—the postman is not allowed to cross the street, to park the van on the wrong side of a two-way street, or enter a one-way street in the wrong direction.

Bibliography

[1] *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, Las Vegas, Nevada, 29 May–1 June 1995.

[2] *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, Philadelphia, Pennsylvania, 22–24 May 1996.

[3] *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, El Paso, Texas, 4–6 May 1997.

[4] *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, California, 22–24 Jan. 1990.

[5] E. M. Arkin and R. Hassin. Minimum diameter covering problems. Submitted, Aug. 1995.

[6] S. Arora. Probabilistic checking of proofs and hardness of approximation problems. Technical Report CS-TR-476-94, Princeton University, 1994. Revised version of a dissertation submitted at CS Division, UC Berkeley, August 1994, `http://ftp.cs.princeton.edu/pub/people/arora/thesis.ps`.

[7] S. Arora. Nearly linear time approximation schemes for Euclidean TSP and other geometric problems. In *38th Annual Symposium on Foundations of Computer Science*, pages 554–563, Miami Beach, Florida, 20–22Oct. 1997.

[8] S. Arora and C. Lund. Hardness of approximations. In D. S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 10. PWS Publishing, Boston, MA, July 1996. `http://ieor.berkeley.edu/~hochbaum/html/book-aanp.html`.

[9] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In IEEE [50], pages 14–23.

[10] S. Arora and S. Safra. Probabilistic checking of proofs; a new characterization of NP. In IEEE [50], pages 2–13.

[11] S. Arora and M. Sudan. Improved low-degree testing and its applications. In ACM [3], pages 485–495.

[12] B. Awerbuch, Y. Azar, A. Blum, and S. Vempala. Improved approximation guarantees for minimum-weight k-trees and prize-collecting salesmen. In ACM [1], pages 277–283.

[13] E. Balas. The prize collecting traveling salesman. *Networks*, 19:621–636, 1989.

[14] C. D. Bateman, C. Helvig, G. Robins, and A. Zelikovsky. Provably good routing tree construction with multi-port terminals. In *Proceedings of ACM/SIGDA International Symposium on Physical Design*, Napa Valley, California, Apr. 1997.

[15] M. Bellare, S. Goldwasser, C. Lund, and A. Russell. Efficient probabilistically checkable proofs and applications to approximation. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 294–304, San Diego, California, 16–18 May 1993.

[16] P. Berman and A. Z. Zelikovsky. Improved approximations for the Steiner tree problem. *Journal of Algorithms*, 17(3):381–408, Nov. 1994.

[17] R. Bhatia, S. Khuller, and J. S. Naor. The loading time scheduling problem. In *36th Annual Symposium on Foundations of Computer Science*, pages 72–81, Milwaukee, Wisconsin, 23–25 Oct. 1995.

[18] D. Bienstock, M. X. Goemans, D. Simchi-Levi, and D. P. Williamson. A note on the prize collecting traveling salesman problem. *Mathematical Programming*, 59:413–420, 1993.

[19] A. Blum, P. Chalasani, and S. Vempala. A constant-factor approximation for the k-MST problem in the plane. In ACM [1], pages 294–302.

[20] A. Blum, R. Ravi, and S. Vempala. A constant-factor approximation algorithm for the k-MST problem. In ACM [2], pages 442–448.

[21] A. Caprara, M. Fischetti, and P. Toth. A heuristic algorithm for the set covering problem. *Operations Research*, 1998. To appear.

[22] S. Chakravarty and P. J. Thadikaran. A study of IDDQ subset selection algorithm for bridging faults. In *IEEE International Test Conference*, pages 403–412, 1994. `http://www.cs.buffalo.edu/pub/WWW/faculty/sreejit/CAD/postscript/ITC94.ps`.

[23] N. Christofides. Worst case analysis of a new heuristic for the traveling salesman problem. Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1976.

[24] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, Aug. 1979.

[25] S. A. Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, Ohio, 3–5 May 1971.

[26] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. John Wiley & Sons, New York, NY, 1998.

[27] P. Crescenzi and V. Kann. A compendium of NP optimization problems. Technical Report SI/RR-95/02, Department of Computer Science, University of Rome "La Sapienza", 1995. `http://www.nada.kth.se/~viggo/problemlist/compendium.html`.

[28] R. Duh and M. Fürer. Approximation of k-set cover by semi-local optimization. In ACM [3], pages 256–264.

[29] J. Edmonds. Lehman's switching game and a theorem of Tutte and Nash-Williams. *Journal of Research of the National Bureau of Standards B*, 69B:73–77, 1965.

[30] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards B*, 69B:125–130, 1965.

[31] J. Edmonds. Minimum partition of a matroid into independent subsets. *Journal of Research of the National Bureau of Standards B*, 69B:67–72, 1965.

[32] U. Feige. A threshold of $\ln n$ for approximating set cover. In ACM [2], pages 314–318.

[33] D. R. Fulkerson. Blocking polyhedra. In B. Harris, editor, *Graph Theory and Its Applications*, pages 93–112. Academic Press, 1970.

[34] D. R. Fulkerson. Blocking and anti-blocking pairs of polyhedra. *Mathematical Programming*, 1(2):168–194, Nov. 1971.

[35] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In ACM/SIAM [4], pages 434–443.

[36] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for general graph-matching problems. *J. ACM*, 38:815–853, 1991.

[37] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, New York, 1979.

[38] N. Garg. A 3-approximation for the minimum tree spanning k vertices. In *37th Annual Symposium on Foundations of Computer Science*, pages 302–309, Burlington, Vermont, 14–16 Oct. 1996.

[39] N. Garg, G. Kojenvod, and R. Ravi. A polylogarithmic approximation algorithm for the group Steiner tree problem. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 253–259, Jan. 1998.

[40] N. Garg and R. Ravi. Personal communication, Oct. 1996.

[41] M. X. Goemans and D. J. Bertsimas. On the parsimonious property of connectivity problems. In ACM/SIAM [4], pages 388–396.

[42] M. X. Goemans and D. J. Bertsimas. Survivable networks, linear programming relaxations and the parsimonious property. *Mathematical Programming*, 60:145–166, 1993.

[43] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM J. Comput.*, 24(2):296–317, Apr. 1995.

[44] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, Berlin, 1988.

[45] M. M. Halldórsson. Approximating discrete collections via local improvements. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 160–169, San Francisco, California, 22–24 Jan. 1995.

[46] M. M. Halldórsson. Approximating set cover via local search. Technical Report IS-RR-95-0002F, JAIST, Mar. 1995.

[47] C. C. Hayes. A model of planning for plan efficiency: Taking advantage of operator overlap. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 949–953, 1989.

[48] I. Holyer. The NP-completeness of edge-coloring. *SIAM Journal on Computing*, 10(4):718–720, Nov. 1981.

[49] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[50] *33rd Annual Symposium on Foundations of Computer Science*, Pittsburgh, Pennsylvania, 24–27 Oct. 1992.

[51] E. Ihler. Bounds on the quality of approximate solutions to the group Steiner problem. In *Graph-Theoretic Concepts in Computer Science, WG90*, volume 484 of *Lecture Notes in Computer Science*, pages 109–118. Springer, 1991.

[52] E. Ihler. The complexity of approximating the class Steiner tree problem. In *Graph-Theoretic Concepts in Computer Science, WG91*, volume 570 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 1992.

[53] E. Ihler. The rectilinear class Steiner tree problem for intervals on two parallel lines. *Mathematical Programming, Series B*, 63:281–296, 1994.

[54] E. Ihler, G. Reich, and P. Widmayer. Class Steiner trees and VLSI-design. Draft paper, `http://www.gmd.de/People/Edmund.Ihler/`, 1996.

[55] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, Dec. 1974.

[56] R. M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[57] M. Karpinski and A. Z. Zelikovsky. New approximation algorithms for the Steiner tree problems. *Journal of Combinatorial Optimization*, 1:47–65, 1997.

[58] M. J. Kearns. *The Computational Complexity of Machine Learning.* MIT Press, Cambridge, Massachussetts, 1990.

[59] G. Laporte, H. Mercure, and Y. Nobert. Generalized travelling salesman problem through n sets of nodes: The asymmetrical case. *Discrete Applied Mathematics*, 18:185–197, 1987.

[60] G. Laporte and Y. Nobert. Generalized travelling salesman problem through n sets of nodes: An integer programming approach. *INFOR*, 21(1):61–75, Feb. 1983.

[61] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[62] E. L. Lawler, J. K. Lenstra, A. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley, New York, 1985.

[63] Y.-N. Lien, E. Ma, and B. W.-S. Wah. Transformation of the generalized traveling-salesman problem into the standard traveling-salesman problem. *Information Sciences*, 74(1):177–189, Oct. 1993.

[64] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[65] L. Lovász. *Combinatorial Problems and Exercises.* North Holland, Amsterdam, 1979.

[66] L. Lovász and M. Plummer. *Matching Theory.* North Holland, Amsterdam, 1986. Math QA 164 L695 1986.

[67] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. *J. ACM*, 41(5):960–981, Sept. 1994.

[68] C. Malandraki. A restricted dynamic programming heuristic algorithm for the generalized traveling salesman problem. Working Paper, Nov. 1993.

[69] R. Motwani. Lecture notes on approximation algorithms—volume I. Technical Report STAN-CS-92-1435, Department of Computer Science, Stanford University, 1992. `http://theory.stanford.edu/people/rajeev/postscripts/approximations.ps.gz`.

[70] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[71] C. E. Noon and J. C. Bean. A Lagrangian based approach for the asymmetric generalized traveling salesman problem. *Operations Research*, 39(4):623–632, July–Aug. 1991.

[72] C. E. Noon and J. C. Bean. An efficient transformation of the generalized traveling salesman problem. *INFOR*, 31(1):39–44, Feb. 1993.

[73] H. L. Ong. Approximate algorithms for the traveling purchaser problem. *Operations Research Letters*, 1(5):201–205, Nov. 1982.

[74] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[75] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[76] P. Raghavan and C. D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

[77] S. Rajagopalan and V. V. Vazirani. Logarithmic approximation of minimum weight k trees. Unpublished manuscript, Aug. 1995.

[78] R. Ravi, R. Sundaram, M. V. Marathe, D. J. Rosenkrantz, and S. S. Ravi. Spanning trees short or small. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 546–555, Philadelphia, Pennsylvania, 23–25 Jan. 1994.

[79] R. Raz and S. Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In ACM [3], pages 475–484.

[80] G. Reich and P. Widmayer. Beyond Steiner's problem: A VLSI oriented generalization. In *Graph-Theoretic Concepts in Computer Science, WG89*, volume 411 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 1990.

[81] H. Rogers. *Theory of Recursive Functions and Effective Computability.* MIT Press, Cambridge, Massachussetts, second edition, 1987.

[82] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563–581, Sept. 1977.

[83] J. P. Saksena. Mathematical model of scheduling clients through welfare agencies. *CORS Journal*, 8:185–200, 1970.

[84] D. B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems. Technical Report ORIE TR-1120, Cornell University, School of Operations Research, Ithaca, NY, 1994. `http://www.orie.cornell.edu/~shmoys/`.

[85] D. B. Shmoys and D. P. Williamson. Analyzing the Held-Karp TSP bound: A monotonicity property with application. *Inf. Process. Lett.*, 35(6):281–285, Sept. 1990.

[86] P. Slavík. Improved performance of the greedy algorithm for minimum set cover and minimum partial cover. Technical Report 95-45, Department of Computer Science, SUNY at Buffalo, Oct. 1995.

[87] P. Slavík. A tight analysis of the greedy algorithm for set cover (extended abstract). In ACM [2], pages 435–441.

[88] P. Slavík. The errand scheduling problem. Technical Report 97-2, Department of Computer Science, SUNY at Buffalo, Mar. 1997.

[89] P. Slavík. Improved performance of the greedy algorithm for partial cover. *Inf. Process. Lett.*, 64:251–254, 1997.

[90] P. Slavík. On the approximation of the generalized traveling salesman problem. Submitted, 1997.

[91] P. Slavík. A tight analysis of the greedy algorithm for set cover. *Journal of Algorithms*, 25:237–254, 1997.

[92] A. Srinivasan. Improved approximation of packing and covering problems. In ACM [1], pages 268–276.

[93] H. Takahashi and A. Matsuyama. An approximate solution for the Steiner problem in graphs. *Mathematica Japonicae*, 24:573–577, 1980.

[94] W. T. Tutte. On the problem of decomposing a graph into n connected factors. *Journal of London Mathematical Society*, 36:221–230, 1961.

[95] L. Wolsey. Heuristic analysis, linear programming and branch and bound. *Mathematical Programming Study*, 13:121–134, 1980.

[96] A. Z. Zelikovsky. An 11/6–approximation algorithm for the network Steiner problem. *Algorithmica*, 9(5):463–470, May 1993.

[97] A. Z. Zelikovsky. A faster approximation algorithm for the Steiner tree problem in graphs. *Inf. Process. Lett.*, 46(2):79–83, May 1993.

Index