Approximation Algorithms

Vijay V. Vazirani

College of Computing
Georgia Institute of Technology

Preliminary and Incomplete

(acknowledgements, credits, references missing)

Contents

Chapter 1

Introduction

This book deals with designing polynomial time approximation algorithms for **NP**-hard optimization problems. Typically[1], the decision versions of these problems are in **NP**, and are therefore **NP**-complete. From the viewpoint of exact solutions, all **NP**-complete problems are equally hard, since they are inter-reducible via polynomial time reductions. Typically, such a reduction maps optimal solutions of the given instance to optimal solutions of the transformed instance and preserves the number of solutions. Indeed, the counting versions of all known **NP**-complete problems are #**P**-complete, and typically the proof follows directly from the proof of **NP**-completeness.

The picture looks different when these problems are studied from the viewpoint of efficiently obtaining near-optimal solutions: polynomial time reductions do not preserve near-optimality of solutions, and **NP**-complete problems exhibit a rich set of possibilities, all the way from allowing approximability to any required degree, to essentially not allowing approximability at all.

A problem is polynomial time solvable only if it has the algorithmically relevant combinatorial structure that can be used as "footholds" to efficiently home in on a solution. The process of designing a polynomial time algorithm is a two-pronged attack: unraveling this structure in the given problem, and finding algorithmic techniques that can exploit this structure.

Although **NP**-hard problems do not offer footholds to find optimal solutions efficiently, they may still offer footholds to find near-optimal solutions efficiently. So, at a high level, the process of designing approximation algorithms is not very different: it still involves unraveling relevant structure and finding algorithmic techniques to exploit it. Typically, the structure turns out to be more elaborate, and often, the algorithmic techniques result from generalizing and extending some of the powerful algorithmic tools developed in the study of exact algorithms. On the other hand, looking at this process a little more closely, one can see that it has its own general principles. In this chapter, we illustrate some of these in an easy setting.

Basic definitions

We first formally define the notions of an optimization problem and an approximation algorithm. An *optimization problem*, Π, consists of:

- A set of *instances*, D_Π. We will assume that all numbers specified in an input are rationals, since our model of computation cannot handle infinite precision arithmetic.

[1]In this paragraph, by "typically" we mean that a conservative estimate of the number of exceptions, among the hundreds of problems that have been studied, is 3.

- Each instance $I \in D_\Pi$ has a set of *feasible solutions*, $S_\Pi(I)$. Further, there is polynomial time algorithm that, given a pair (I, s), decides whether $s \in S_\Pi(I)$.
- There is a polynomial time computable *objective function* f_Π, that assigns a non-negative rational number to each pair (I, s), where I is an instance and s is a feasible solution for I. (The objective function is frequently given a physical interpretation, such as cost, length, weight etc.)
- Finally, Π is specified to be either a *minimization problem* or a *maximization problem.*

An *optimal solution* for an instance of a minimization (maximization) problem is a feasible solution that achieves the smallest (largest) objective function value. We will denote by OPT(I) the objective function value of an optimal solution to instance I. In this book, we will shorten this to OPT when it is clear that we are referring to a generic instance of the problem being studied.

The *size* of instance I, denoted by $|I|$, is defined as the number of bits needed to write I under the assumption that all numbers occurring in the instance are written in binary.

Let us illustrate these definitions in the context of the following minimization problem:

Problem 1.1 (Minimum vertex cover) Given an undirected graph $G = (V, E)$, find a minimum cardinality *vertex cover*, i.e., a set $V' \subseteq V$ such that every edge has at least one end point incident at V'.

Instances of this problem are undirected graphs. Given an instance $G = (V, E)$, feasible solutions are all vertex covers for G. The objective function value for a solution $V' \subseteq V$ is the cardinality of V'. Any minimum cardinality vertex cover is an optimal solution.

An approximation algorithm produces feasible solutions that are "close" to the optimal; the formal definition differs for minimization and maximization problems. Let Π be a minimization (maximization) problem, and let δ be a positive real number, $\delta \geq 1$ ($\delta \leq 1$). An algorithm $\mathcal{A}$ is said to be a *δ factor approximation algorithm for* Π if on each instance I, $\mathcal{A}$ produces a feasible solution s for I, such that $f_\Pi(I, s) \leq \delta \cdot \mathrm{OPT}(I)$ ($f_\Pi(I, s) \geq \delta \cdot \mathrm{OPT}(I)$). An important requirement is that $\mathcal{A}$ be a polynomial time algorithm, i.e., its running time should should be bounded by a fixed polynomial in the size of instance I. Clearly, the closer δ is to 1, the better is the approximation algorithm.

On occassion, we will relax this definition and will allow $\mathcal{A}$ to be randomized, i.e., it will be allowed to use the flips of a fair coin. Assume we have a minimization problem. Then, we will say that $\mathcal{A}$ *is a δ factor randomized approximation algorithm for* Π if on each instance I, $\mathcal{A}$ produces a feasible solution s for I, such that

$$\mathbf{Pr}[f_\Pi(I, s) \ \leq \ \delta \cdot \mathrm{OPT}(I)] \geq \frac{1}{2},$$

where the probability is over the coin flips. The definition for a maximization problem is analogous.

Thus, an algorithm is a factor 2 approximation algorithm for the minimum vertex cover problem if it produces a vertex cover whose cardinality is within twice the optimal cover, and its running time is polynomial in the number of vertices in the instance (notice that the size of an instance is bounded by a polynomial in the number of vertices).

Lower bounding OPT

While designing an approximation algorithm for an **NP**-hard problem, one is immediately faced with the following dilemma: For establishing the approximation guarantee, the cost of the solution produced by the algorithm needs to be compared with the cost of an optimal solution. However, it is **NP**-hard not only to find an optimal solution, but also to compute the cost of such a solution. So, how do we establish the approximation guarantee? The answer to this question provides a key step in the design of approximation algorithms. Let us show this in the context of the minimum vertex cover problem.

Designing an approximation algorithm for vertex cover

We will get around the difficulty mentioned above by coming up with a "good" polynomial time computable *lower bound* on the size of the optimal cover. Let us observe that the size of a maximal matching in G provides a lower bound. This is so because *any* vertex cover has to pick at least one end point of each matched edge. This lower bounding scheme immediately suggests the following simple algorithm:

Algorithm 1.2 (Minimum vertex cover)

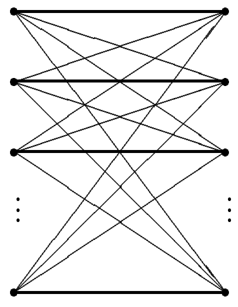
Find a maximal matching in G, and output the set of matched vertices.

Theorem 1.3 *Algorithm 1.2 is a factor 2 approximation algorithm for the minimum vertex cover problem.*

Proof : No edge can be left uncovered by the set of vertices picked – otherwise such an edge could have been added to the matching, contradicting its maximality. Let M be the matching picked. As argued above, $|M| \leq \text{OPT}$. The approximation factor follows from the observation that the cover picked by the algorithm has cardinality $2 \cdot |M|$, which is $\leq 2 \cdot \text{OPT}$. □

As with any approximation algorithm, the following question arises: is our analysis tight? i.e., can we establish a factor better than 2 for our algorithm, or is there an example on which the vertex cover found is twice the optimal cover? It turns out that the analysis presented above is tight, as shown in Example 1.4.

Example 1.4 Consider the infinite family of instances given by the complete bipartite graphs $K_{n,n}$.



When run on $K_{n,n}$ the algorithm will pick all $2n$ vertices, whereas picking one side of the bipartition gives a cover of size n. □

An infinite family of instances of this kind showing that the analysis of an approximation algorithm is tight will be referred to as a *tight example*. The importance of finding tight examples for an approximation algorithm one has designed cannot be over emphasised: they give critical insight into the functioning of the algorithm (the reader is advised to run algorithms on tight examples presented in this book), and have often led to ideas for obtaining algorithms with improved guarantees.

The example given above shows that even if we had picked a maximum matching instead of a maximal matching, we would not have obtained a better approximation guarantee. Indeed, obtaining a better approximation guarantee for vertex cover is currently a major open problem. Perhaps a key step is obtaining a better lower bounding technique. The current method can give a lower bound that is only half the size of the optimal vertex cover, as shown in the following example: Consider the complete graph K_n, where n is odd. Then, the size of any maximal matching is $(n-1)/2$, whereas the size of an optimal cover is $n-1$.

Simple recipes vs. grand theories

Assuming we have a minimization problem at hand, typically a central step in designing an approximation algorithm is studying its combinatorial structure to obtain a good way of lower bounding the cost of the optimal solution (for a maximization problem, we will need a good upper bound). Indeed, as in the vertex cover problem, sometimes the algorithm follows easily once this is done.

The job of a potential algorithm designer, faced with an **NP**-hard problem, could certainly be simplified if one could give a standard set of recipes for coming up with the lower bounding scheme and the algorithmic idea. However, nature is very rich, and we cannot expect a few tricks to help solve the diverse collection of **NP**-hard problems. Indeed, in Part I, we have purposely refrained from categorizing these techniques so as not to trivialize matters. Instead, we have attempted to capture, as accurately as possible, the individual character of each problem. In Part II, we show how LP-duality theory helps provide lower bounds for a large collection of problems. But once again, the exact approximation guarantee obtainable depends on the specific LP-relaxation used, and there is no fixed recipe for discovering good relaxations, just as there is no fixed recipe for proving a theorem in mathematics (readers familiar with complexity theory will recognize this as the philosophical point behind the $\mathbf{P} \neq \mathbf{NP}$ question).

One may wonder then whether there is a *theory* of approximation algorithms. One of the purposes of this book is to show that we have here the beginnings of a promising theory. Of course, confirmation lies only in the future; in the meantime, judgement lies in the eyes of the beholder.

The greedy schema

Perhaps the first strategy one tries when designing an algorithm for an optimization problem is the greedy strategy, in some form. For instance, for the vertex cover problem, such a strategy would have been to pick a maximum degree vertex, remove it from the graph, and iterate until there are no edges left. This strategy is not good for vertex cover – the cover picked by this algorithm can be $\omega(\log n)$ factor larger than the optimal (see Exercise 1.5). However, the greedy schema yields good exact and approximation algorithms for numerous problems. Furthermore, even if it does not work for a specific problem, proving this via a counter-example can provide crucial insights into the structure of the problem.

A maximal matching can be found via a greedy algorithm: pick an edge, remove its two end points, and iterate until there are no edges left. Does this make Algorithm 1.2 a greedy algorithm? This is a moot point; it depends on how generally one defines "greedy". In its extreme definition, any good algorithm is greedy since it is making progress. Under a strict definition of "greedy algorithm" one can show that an optimization problem has a greedy (exact) algorithm iff its underlying structure is a matroid. There is no suitable definition that captures greedy strategies used in this book for obtaining approximation algorithms. The set cover problem, discussed in Chapter 2, provides a very good example of the use of this schema.

Yes and No certificates

Since the design of approximation algorithms involves delicately attacking **NP**-hardness and salvaging from it an efficient approximate solution, it will be useful to review some key concepts from complexity theory. Let us do this in the context of the minimum vertex cover problem.

Let us consider two problems: those of finding a maximum matching and a minimum vertex cover in a graph $G = (V, E)$. Consider the following decision problems (the fact that one is a maximization problem, and the other a minimization problem is reflected in the way these questions are posed):

- Is the size of the maximum matching in G at least k?
- Is the size of the minimum vertex cover in G at most l?

Convincing someone that the answers to these questions are "Yes" involves simply demonstrating a matching of size k and a vertex cover of size l respectively. These answers constitute *Yes certificates* for the problems: a polynomial sized guess (in the size of the problem instance) that can convince one in polynomial time that the answer to the question is "Yes". Indeed, this is a characterization of **NP**; **NP** is the class of problems that have short, i.e., polynomial size bounded, Yes certificates.

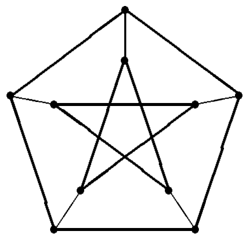
How do we convince someone that the answer to either one of these questions is "No"? We have already observed that the size of a maximum matching is a lower bound on the size of a minimum vertex cover. If G is bipartite, then in fact equality holds; this is the classic König-Egerváry Theorem:

$$\max_{\text{matching } M} \{|M|\} = \min_{\text{v. cover } U} \{|U|\}.$$

So, if the answer to the first question is "No", there must be a vertex cover of size $k-1$ that can be given as a certificate. Similarly, a matching of size $l+1$ must exist in G if the answer to the second question is "No". These constitute *No certificates* for these problems, showing that both problems are in co-**NP** when restricted to bipartite graphs. In fact, both problems are in **P** as well under this restriction. It is easy to see that any problem in **P** has Yes as well as No certificates: an optimal solution, which can be verified to be optimal in polynomial time, provides certificates to both questions. This is equivalent to the statement that **P** is contained in **NP** $\cap$ co-**NP**.

Edmonds coined the term "well-characterized" to describe problems that have Yes and No certificates, i.e., are in **NP**$\cap$co-**NP**. Indeed, his quest for a polynomial time algorithm for matching started with the observation that it is well-characterized. Min-max relations of the kind given above give a proof that a problem is well-characterized. Such relations are some of the most powerful and beautiful results in combinatorics, and some of the most fundamental polynomial time algorithms have been designed around such relations. Most of these min-max relations are actually special cases of the LP-Duality Theorem; see Chapter 10 for an explanation. Interestingly enough, LP-duality theory plays an even more vital role in the design of approximation algorithms: it provides a unified way of obtaining good lower bounds for several key problems; a good fraction of this book will be devoted to such algorithms.

Returning to the issue of Yes and No certificates, what if G is not restricted to be bipartite? In this case, a maximum matching may be strictly smaller than a minimum vertex cover. For instance, if G is simply an odd length cycle on $2p+1$ vertices, then the size of a maximum matching is p, whereas the size of a minimum vertex cover is $p+1$. This may happen even for graphs having a perfect matching, for instance, the Petersen graph:



This graph has a perfect matching, of cardinality 5; however, the minimum vertex cover has cardinality 6. One can show that there is no vertex cover of size 5 by observing that any vertex cover must pick at least $p+1$ vertices from an odd cycle of length $2p+1$, just to cover all the edges of the cycle, and the Petersen graph has two disjoint cycles of length 5.

Under the widely believed assumption that $\mathbf{NP} \neq$ co-$\mathbf{NP}$, $\mathbf{NP}$-hard problems do not have No certificates. Thus the minimum vertex cover problem in general graphs, which is $\mathbf{NP}$-hard, does not have a No certificate. The maximum matching problem in general graphs is in $\mathbf{P}$(however, the No certificate for this problem is not a vertex cover, but a more general structure: an odd set cover[2]). Under the assumption $\mathbf{NP} \neq$ co-$\mathbf{NP}$, there is no min-max relation for the minimum vertex cover in general graphs. However, the approximation algorithm presented above gives the following approximate min-max relation:

$$\max_{\text{matching } M} |M| \;\leq\; \min_{\text{v.cover } U} |U| \;\leq 2 \max_{\text{matching } M} |M|.$$

Approximation algorithms frequently yield such approximate min-max relations, which can be of independent interest.

Scientific and practical significance

The study of exact algorithms and complexity theory have led to a reasonable understanding of the intrinsic complexity of natural computational problems (modulo some very difficult,

[2]An *odd set cover* C of a graph $G=(V,E)$ is a collection of disjoint odd cardinality subsets of V, $S_1,\ldots,S_k$ and a collection of vertices $v_1,\ldots,v_l$ such that each edge of G is either incident at one of the vertices v_i or has both endpoints in one of the sets S_i. The *weight* of this cover C is defined to be $w(C) = l + \sum_{i=1}^{k}(|S_i|-1)/2$. The following min-max relation holds:

$$\max_{\text{matching } M}\{|M|\} = \min_{\text{odd set cover } C}\{w(C)\}.$$

though strongly believed, conjectures). Since most of these problems are in fact **NP**-hard, charting the landscape of approximability of these problems via efficient algorithms becomes a compelling subject of scientific inquiry in computer science and mathematics. This task involves identifying cornerstone problems, finding relationships among problems, understanding the kinds of combinatorial structures that play the role of "footholds" refererred to earlier, developing appropriate algorithmic techniques, and developing the theory of hardness of approximability so that at least for the key problems we obtain positive and negative approximability results having matching bounds. Exact algorithms have been studied intensively for over three decades, and yet basic insights are still being obtained. It is reasonable to expect the theory of approximation algorithms to take its time.

NP-hard problems abound in practice. On the one hand this fact adds to the importance of the area of approximation algorithms, and on the other hand, it raises our expectations from it. At the very least, we can expect this theory to provide guidelines for use in practice. For instance, simply knowing the limits of approximability of natural problems can be useful in choosing the right problem to model a realistic situation with.

What about the possibility of using algorithms developed directly in practice? Can we expect this theory to have such direct impact on practice? One is inclined to say "No", simply because practitioners are normally looking for algorithms that give solutions that are very close to optimal; say having error within 2% or 5% of the optimal, not within 100%, a common bound in several approximation algorithms. Further, by this token, what is the usefulness of improving the approximation guarantee from say factor 2 to 3/2?

Let us address both issues and point out some fallacies in these assertions. The approximation guarantee only reflects the performance of the algorithm on the most pathological instances. Perhaps it is more appropriate to view the approximation guarantee as a measure that forces us to explore deeper combinatorial structure of the problem (really getting into its 'guts'!) and discover more powerful tools for exploiting this structure. It has been observed that the difficulty of constructing tight examples increases considerably as one obtains algorithms with better guarantees. Indeed, for some recent algorithms [**?**], obtaining a tight example has been a paper by itself! These and other sophisticated algorithms do have error bounds of the desired magnitude, 2% to 5%, on typical instances, even though their worst case error bounds are much higher [**?**]. In addition, the theoretically proven algorithm should be viewed as a core algorithmic idea that needs to be fine-tuned to the types of instances arising in specific applications. All this points to the importance of implementing and experimenting with the algorithms developed.

More than half of this book is based on the surge of progress made in the last eight years, after a long period of lull. Already, in a few instances, sophisticated algorithmic ideas have found their way into industrial products (see for example [**?**]). For more widespread use however, we will have to wait until these ideas diffuse into the right circles – hopefully this book will help speed up the process.

Exercise 1.5 Show that the greedy algorithm for minimum vertex cover achieves an approximation guarantee of $O(\log n)$. Give a tight example for this algorithm.

Exercise 1.6 Give a lower bounding scheme for the weighted version of vertex cover, in which you are trying to minimize the weight of the cover picked.

Part I

COMBINATORIAL ALGORITHMS

Chapter 2

Set cover and its application to shortest superstring

Understanding the area of approximation algorithms involves identifying cornerstone problems: problems whose study leads to discovering techniques that become general principles in the area, and problems that are general enough that other problems can be reduced to them. Problems such as matching, maximum flow, shortest path and minimum spanning tree are cornerstone problems in the design of exact algorithms. In approximation algorithms, the picture is less clear at present. Even so, problems such as minimum set cover and minimum Steiner tree can already be said to occupy this position.

In this chapter, we will first analyse a natural greedy algorithm for the minimum set cover problem. We will then show an unexpected use of set cover to solve the minimum superstring problem. An algorithm with a much better approximation guarantee will be presented in Chapter 7 for the latter problem; the point here is to illustrate the wide applicability of the set cover problem.

Minimum set cover

Problem 2.1 (Minimum set cover) Given a universe U of n elements and a collection of subsets of U, $S_1, \ldots S_k$, with non-negative costs specified, the minimum set cover problem asks for a minimum cost collection of sets whose union is U.

Perhaps the first algorithm that comes to mind for this problem is one based on the greedy strategy of iteratively picking the most cost-effective set and removing the covered elements, until all elements are covered. Let C be the set of elements already covered at the beginning of an iteration. During this iteration, define the *cost-effectiveness* of a set S to be the average cost at which it covers new elements, i.e., $\frac{\text{cost}(S)}{|S \cap \overline{C}|}.$ Define the *price* of an element to be the average cost at which it is covered. Equivalently, when a set S is picked, we can think of its cost being distributed equally among the new elements covered, to set their prices.

Algorithm 2.2 (Greedy set cover algorithm)

1. $C \leftarrow \emptyset$
2. while $C \neq U$ do

   Find the most cost-effective set in the current iteration, say S.

   Let $\alpha = \frac{\text{cost}(S)}{|S \cap \overline{C}|}$, i.e., the cost-effectiveness of S.

   Pick S, and for each $e \in S - C$, $\text{price}(e) \leftarrow \alpha$.
3. Output the picked sets.

Number the elements of U in the order in which they were covered by the algorithm, resolving ties arbitrarily. Let $e_1, \ldots e_n$ be this numbering.

Lemma 2.3 *For each $k \in \{1, \ldots, n\}$, $\text{price}(e_k) \leq \frac{\text{OPT}}{n-k+1}$.*

Proof : In any iteration, the left over sets of the optimal solution can cover the remaining elements at a cost of at most OPT. Therefore, there must be a set having cost-effectiveness at most $\frac{\text{OPT}}{|\overline{C}|}$. In the iteration in which element e_k was covered, $\overline{C}$ contained at least $n-k+1$ elements. Since e_k was covered by the most cost-effective set in this iteration, it follows that

$$\text{price}(e_k) \leq \frac{\text{OPT}}{|\overline{C}|} \leq \frac{\text{OPT}}{n-k+1}.$$

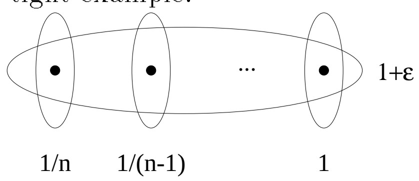
□

From Lemma 2.3, we immediately obtain:

Theorem 2.4 *The greedy algorithm is an H_n factor approximation algorithm for the minimum set cover problem, where $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$.*

Proof : Since the cost of each set picked is distributed among the new elements covered, the total cost of the set cover picked is equal to $\sum_{k=1}^{n} \text{price}(e_k)$. By Lemma 2.3, this is at most $\left(1 + \frac{1}{2} + \cdots + \frac{1}{n}\right) \cdot \text{OPT}$. □

Example 2.5 Following is a tight example:



The greedy algorithm outputs the cover consisting of the n singleton sets, since in each iteration some singleton is the most cost-effective set. So, the algorithm outputs a cover of cost

$$\frac{1}{n} + \frac{1}{n-1} + \cdots + 1 = H_n.$$

On the other hand, the optimal cover has a cost of $1 + \epsilon$. □

Surprisingly enough, the obvious algorithm given above is essentially the best one can hope for for the minimum set cover problem: it is known that an approximation guarantee better than $O(\ln n)$ is not possible, assuming $\mathbf{P} \neq \mathbf{NP}$ (it is easy to see that $\ln n + 1 \leq H_n \leq 1 + \ln n$).

In Chapter 10 we pointed out that finding a good lower bound on OPT is a basic starting point in the design of an approximation algorithm for a minimization problem. At this point, the reader may be wondering whether there is any truth to this claim. We will show in Chapter 12 that the correct way to view the greedy set cover algorithm is in the setting of LP-duality theory – this will not only provide the lower bound on which this algorithm is based, but will also help obtain algorithms for several generalizations of this problem.

Solving shortest superstring via set cover

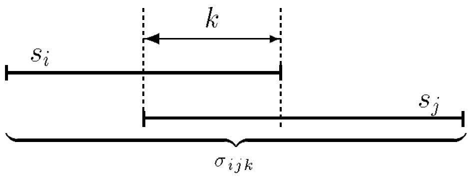
Let us motivate the shortest superstring problem. The human DNA can be viewed as a very long string over a four letter alphabet. Scientists are attempting to decipher this string. Since it is very long, several overlapping short segments of this string are first deciphered. Of course, the locations of these segments on the original DNA are not known. It is hypothesised that the shortest string which contains these segments as substrings is a good approximation to the original DNA string.

Another application of this problem is in data compression: instead of transmitting n strings individually, one can instead transmit their supersting and the starting positions of each string in the superstring.

Problem 2.6 (Shortest superstring) Given a finite alphabet Σ, and a set of n strings, $S = \{s_1, \ldots, s_n\} \subseteq \Sigma^+$, find a shortest string s that contains each s_i as a substring. Without loss of generality, we may assume that no string s_i is a substring of another string s_j, $j \neq i$.

This problem is **NP**-hard. Perhaps the first algorithm that comes to mind for finding a short superstring is the following greedy algorithm. Define the *overlap* of two strings $s, t \in \Sigma^*$ as the maximum length of a suffix of s that is also a prefix of t. The algorithm maintains a set of strings T; initially $T = S$. At each step, the algorithm selects from T two strings that have maximum overlap and replaces them with the string obtained by overlapping them as much as possible. After $n - 1$ steps, T will contain a single string. Clearly, this string contains each s_i as a substring. This algorithm is conjectured to have an approximation factor of 2. To see that the approximation factor of this algorithm is no better than 2, consider an input consisting of 3 strings: ab^k, b^kc, and b^{k+1}. If the first two strings are selected in the first iteration, the greedy algorithm produces the string ab^kcb^{k+1}. This is almost twice as long as the shortest superstring, $ab^{k+1}c$.

We will obtain a $2 \cdot H_n$ factor approximation algorithm by a reduction to the minimum set cover problem. The set cover instance, denoted by $\mathcal{S}$, is constructed as follows. For $s_i, s_j \in S$ and $k > 0$, if the last k symbols of s_i are the same as the first k symbols of s_j, let σ_{ijk} be the string obtained by overlapping these k positions of s_i and s_j. Let set I consist of the strings σ_{ijk} for all valid choices of i, j, k. For a string $\pi \in \Sigma^+$, define $\mathrm{set}(\pi) = \{s \in S | s \text{ is a substring of } \pi\}$. The The universal set of set cover instance $\mathcal{S}$ is S, and the specified subsets of S are $\mathrm{set}(\pi)$ for each string $\pi \in S \cup I$. The cost of the set $\mathrm{set}(\pi)$ is $|\pi|$, i.e., the length of string π.



Let $\mathrm{OPT}_{\mathcal{S}}$ and OPTdenote the cost of an optimal solution to $\mathcal{S}$ and the length of the shortest superstring of S respectively. As shown in Lemma 2.8, $\mathrm{OPT}_{\mathcal{S}}$ and OPTare within a factor of 2 of each other, and so an approximation algorithm for set cover can be used to obtain an approximation algorithm for shortest superstring. The complete algorithm is:

Algorithm 2.7 (Shortest superstring via set cover)

1. Use the greedy set cover algorithm to find a cover for the instance $\mathcal{S}$. Let $\mathrm{set}(\pi_1), \ldots, \mathrm{set}(\pi_k)$ be the sets picked by this cover.
2. Concatenate the strings $\pi_1, \ldots, \pi_k$ in any order.
3. Output the resulting string, say s.

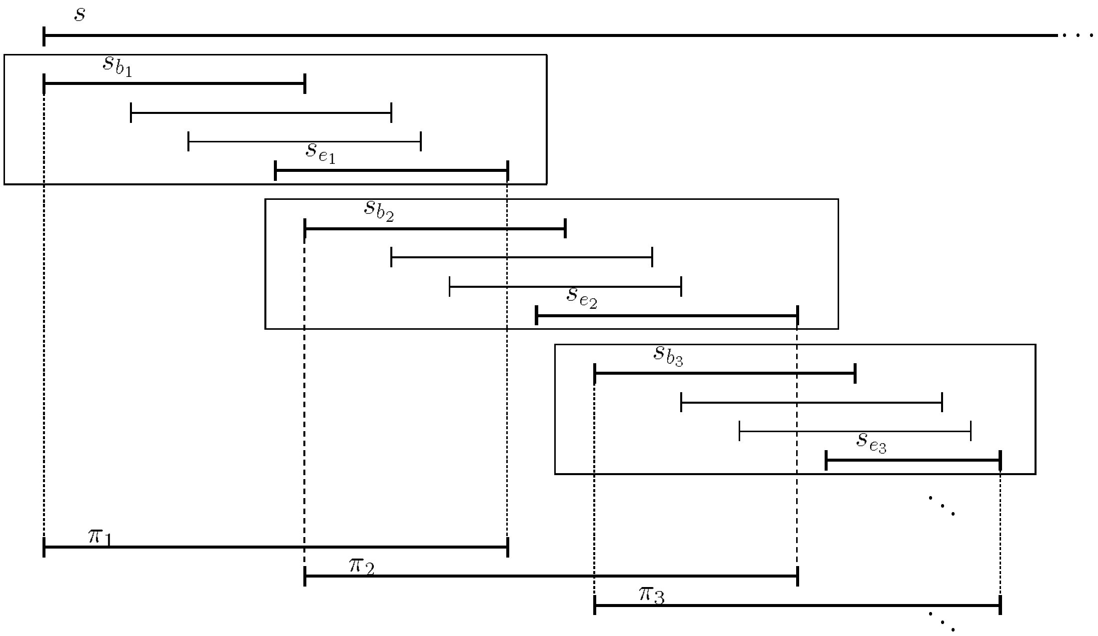
Lemma 2.8

$$\mathrm{OPT} \leq \mathrm{OPT}_{\mathcal{S}} \leq 2 \cdot \mathrm{OPT}.$$

Proof : Consider an optimal set cover, say $\{\mathrm{set}(\pi_i) | 1 \leq i \leq l\}$, and obtain a string, say s, by concatenating the strings π_i, $1 \leq i \leq l$ in any order. Clearly, $|s| = \mathrm{OPT}_{\mathcal{S}}$. Since each string of S is a substring of some π_i, $1 \leq i \leq l$, it is also a substring of s. Hence $\mathrm{OPT}_{\mathcal{S}} = |s| \geq \mathrm{OPT}$.

To prove the second inequality, let s be a shortest superstring of $s_1, \ldots, s_n$, $|s| = \mathrm{OPT}$. It suffices to produce *some* set cover of cost at most $2 \cdot \mathrm{OPT}$.

Assume that $s_1, \ldots, s_n$ are numbered in order of their leftmost occurrence in s. For the rest of the proof, we will only consider the leftmost occurrences of $s_1, \ldots, s_n$ in S. For any $i < j$, the occurrence of s_i in s must end before the occurrence of s_j (otherwise s_j would be a substring of s_i).



We will partition the ordered list of strings $s_1, \ldots, s_n$ in groups. Each group will consist of a contiguous set of strings from this list; b_i and e_i will denote the index of the first and last string in the i^{th} group ($b_i = e_i$ is allowed). Let $b_1 = 1$, and let e_1 be the largest index of a string that overlaps with s_1 (there exists at least one such string, namely s_1 itself). In general, if $e_i < n$ we set $b_{i+1} = e_i + 1$ and denote by e_{i+1} the largest index of a string that overlaps with $s_{b_{i+1}}$. Eventually, we will get $e_t = n$ for some $t \leq n$.

For each pair of strings (s_{b_i}, s_{e_i}), let $k_i > 0$ be the length of the overlap between their leftmost occurrences in s (this may be different from their maximum overlap). Let $\pi_i = \sigma_{b_i e_i k_i}$. Clearly, $\{\text{set}(\pi_i) | 1 \leq i \leq t\}$ is a solution for $\mathcal{S}$, of cost $\sum_i |\pi_i|$.

The critical observation is that π_i does not overlap π_{i+2}. We will prove this claim for $i = 1$; the same argument applies to arbitrary i. Assume for a contradiction that π_1 overlaps π_3. Then the occurrence of s_{b_3} in s overlaps the occurrence of s_{e_1}. Because $e_1 < b_2 < b_3$, it follows that the occurrence of s_{b_3} overlaps the occurrence of s_{b_2}. Ths contradicts the fact that e_2 is the highest indexed string that overlaps with s_{b_2}.

Because of this observation, each symbol of s is covered by at most two of the π_i's. Hence $\text{OPT}_{\mathcal{S}} \leq \sum_i |\pi_i| \leq 2 \cdot \text{OPT}$. □

Lemma 2.8 immediately gives:

Theorem 2.9 *Algorithm 2.7 is a $2 \cdot H_n$ factor algorithm for the shortest superstring problem.*

Exercise 2.10 A more elaborate argument shows that in fact Algorithm 2.2 achieves an approximation factor of H_m, where m is the cardinality of the largest specified subset of U. Prove this approximation guarantee.

Exercise 2.11 Show that a similar greedy strategy achieves an approximation guarantee of H_n for set multi-cover, a generalization of set cover in which an integral coverage requirement is also specified for each element, and sets can be picked multiple number of times to satisfy all coverage requirements. Assume that the cost of picking $alpha$ copies of set S_i is $\alpha \cdot \text{cost}(S_i)$.

Exercise 2.12 By giving an appropriate tight example, show that the analysis of Algorithm 2.2 cannot be improved even if all specified sets have unit cost. Hint: Consider running the greedy algorithm on a vertex cover instance.

Exercise 2.13 The *maximum coverage problem* is the following: Given a universe U of n elements, with non-negative weights specified, a collection of subsets of U, $S_1, \ldots, S_l$, and an integer k, pick k sets so as to maximize the weight of elements covered. Show that the obvious algorithm, of greedily picking the best set in each iteration until k sets are picked, achieves an approximation factor of

$$(1 - (1 - \frac{1}{k})^k) \;>\; (1 - \frac{1}{e}).$$

Exercise 2.14 Using set cover, obtain approximation algorithms for the following variants of the shortest superstring problem (here s^R is the reverse of string s):

1. Find the shortest string s that contains for each string $s_i \in S$, both s_i and s_i^R as subtrings.
2. Find the shortest string s that contains for each string $s_i \in S$, either s_i or s_i^R as a subtring. (Notice that this suffices for the data compression application given earlier.)

Chapter 3

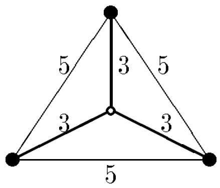
Metric Steiner tree and TSP

The origin of the Steiner tree problem goes back to Gauss, who posed it in a letter to Schumacher. This problem and its generalizations will be studied extensively in this monograph.

Problem 3.1 (Metric Steiner tree) Given a graph $G = (V, E)$ whose edge costs satisfy triangle inequality and whose vertices are partitioned into two sets, *required* and *Steiner*, find a minimum cost tree containing all the required vertices and any subset of the Steiner vertices.

Remark: There is no loss of generality in requiring that the edge costs satisfy triangle inequality: if they don't satisfy triangle inequality, construct the *metric closure* of G, say G', which has the same vertex set as G and edge costs given by shortest distances in G. Clearly, the cost of the optimal Steiner tree in both graphs must be the same. Now, obtaining a Steiner tree in G', and replacing edges by paths wherever needed, gives a Steiner tree in G of at most the same cost.

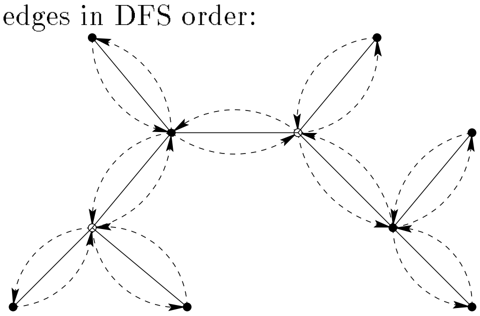
Let R denote the set of required vertices. Clearly, a minimum spanning tree (MST) on R is a feasible solution for this problem. Since the problem of finding an MST is in **P**and the metric Steiner tree problem is **NP**-hard, we cannot expect the MST on R to always give an optimal Steiner tree; below is an example in which the MST is strictly costlier.



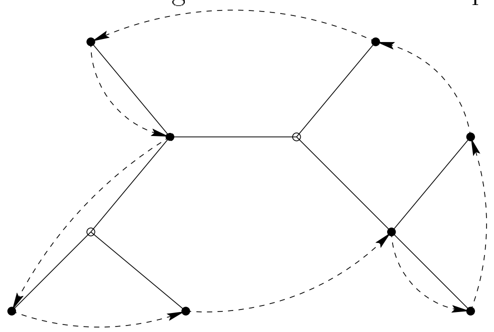
Even so, an MST on R is not much more costly than an optimal Steiner tree:

Theorem 3.2 *The cost of an MST on R is within* $2 \cdot \mathrm{OPT}$.

Proof : Consider a Steiner tree of cost OPT. By doubling its edges we obtain an Eulerian graph connecting all vertices of R and, possibly, some Steiner vertices. Find an Euler tour of this graph, for example by traversing the edges in DFS order:



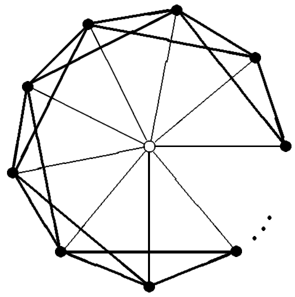
The cost of this Euler tour is $2 \cdot \mathrm{OPT}$. Next obtain a Hamilton tour on the vertices of R by traversing the Euler tour and "short-cutting" Steiner vertices and previously visited vertices of R:



Because of triangle inequality, the shortcuts do not not increase the cost of the tour. If we delete one edge of this Hamilton tour, we obtain a path that spans R and has cost at most $2 \cdot \mathrm{OPT}$. This path is also a spanning tree on R. Hence, the MST on R has cost at most $2 \cdot \mathrm{OPT}$. □

Theorem 3.2 gives a straightforward factor 2 algorithm for the metric Steiner tree problem: simply find an MST on the set of required vertices. As in the case of set cover, the "correct" way of viewing this algorithm is in the setting of LP-duality theory. This will provide the lower bound on which this algorithm is based, and will also help solve generalizations of this problem.

Example 3.3 A tight example is provided by a graph with n required vertices and one Steiner vertex. Each edge between the Steiner vertex and a required vertex has cost 1, and all other edges have cost $(2-\epsilon)$, where $\epsilon > 0$ is a small number (not all edges of cost $(2-\epsilon)$ are shown below). In this graph, an MST on R has cost $(2-\epsilon)(n-1)$, while $\mathrm{OPT} = n$.



□

Exercise 3.4 Let $G = (V, E)$ be a graph with non-negative edge costs. The vertices of G are partitioned into two sets, *senders* and *receivers*. The problem is to find a minimum cost subgraph of G that has a path connecting each receiver to a sender. Give a good approximation algorithm for this NP-hard problem.

Approximation algorithms for TSP

The following is a well-studied problem in combinatorial optimization.

Problem 3.5 (Traveling salesman problem (TSP)) Given a complete graph with non-negative edge costs, find a minimum cost cycle visiting every vertex exactly once.

Not only is it **NP**-hard to solve this problem exactly, but also approximately:

Theorem 3.6 *For any polynomial time computable function $\alpha(n)$, TSP cannot be approximated within a factor of $\alpha(n)$, unless $\mathbf{P} = \mathbf{NP}$.*

Proof : Assume for a contradiction that for any graph on n vertices, we can find in polynomial time a salesman tour whose cost is within a factor of $\alpha(n)$ from the optimum. We show that this implies a polynomial time algorithm for deciding whether a given graph has a Hamiltonian cycle.

Let G be a graph on n vertices. We extend G to the complete graph on n vertices, assigning unit cost to edges of G, and a cost of $n\alpha(n)$ to edges not in G. Clearly, the optimal salesman tour in the new graph has a cost of n if and only if G has a Hamiltonian cycle. Moreover, any tour that contains a new edge costs more than $n\alpha(n)$. So, an $\alpha(n)$ approximation algorithm finds a tour of cost n whenever one exists. □

Notice that in order to obtain such a strong non-approximability result, we had to assign edge costs that violate triangle inequality. If we restrict ourselves to graphs in which edge costs satisfy triangle inequality, i.e., the *metric traveling salesman problem*, the problem remains **NP**-complete, but it is no longer hard to approximate.

We will first present a simple factor 2 algorithm. The lower bound we will use for obtaining this factor is the cost of an MST in G. This is a lower bound because deleting any edge from an optimal solution to TSP we get a spanning tree of G.

Algorithm 3.7 (Metric TSP – factor 2)

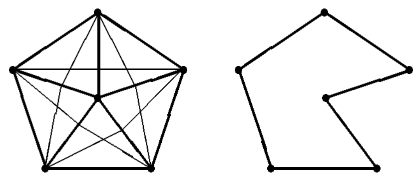
1. Find an MST, T, of G.
2. Double every edge of the MST to obtain an Eulerian graph.
3. Find an Euler tour, $\mathcal{T}$, on this graph.
4. Output the tour that visits vertices of G in order of their first appearence in $\mathcal{T}$. Let $\mathcal{C}$ be this tour.

Notice that Step 4 is similar to the "short-cutting" step in Theorem 3.2.

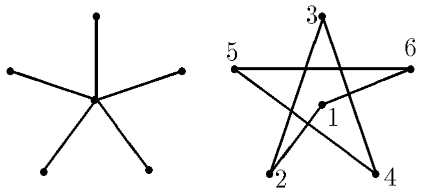
Theorem 3.8 *Algorithm 3.7 is a factor 2 algorithm for metric TSP.*

Proof : As noted above, $\text{cost}(T) \leq \text{OPT}$. Since $\mathcal{T}$ contains each edge of T twice, $\text{cost}(\mathcal{T}) = 2 \cdot \text{cost}(T)$. Because of triangle inequality, after the "short-cutting" step, $\text{cost}(\mathcal{C}) \leq \text{cost}(\mathcal{T})$. Combining these inequalities we get that $\text{cost}(\mathcal{C}) \leq 2 \cdot \text{OPT}$. □

Example 3.9 A tight example for this algorithm is given by the complete graph on n vertices with edges of cost 1 and 2. We present the graph for $n = 6$ below. The thick edges have cost 1 and the remaining edges have cost 2. On n vertices, the graph will have $2n - 2$ edges of cost 2, and the remaining edges of cost 1, with the cost 2 edges forming the union of a star and an $n - 1$ cycle. The optimal TSP tour has cost n as shown below.



Suppose that the MST found by the algorithm is the spanning star created by edges of cost 1. Moreover, suppose that the Euler tour constructed in Step 3 visits vertices in order shown below:



Then the tour obtained after short-cutting contains $n-2$ edges of cost 2, and has a total cost of $2n-2$. This is almost twice the cost of the optimal TSP tour. $\square$

Essentially, this algorithm first finds a low cost Euler tour spanning the vertices of G, and then short-cuts this tour to find a travelling salesman tour. Is there a cheaper Euler tour than that found by doubling an MST? Notice that we only need to be concerned about the vertices of odd degree in the MST; let V' denote this set of vertices. $|V'|$ must be even since the sum of degrees of all vertices in the MST is even (it is $2n-2$). Now, if we add to the MST a minimum cost perfect matching on V', every vertex will have even degree, and we get an Eulerian graph. With this modification, the algorithm achieves an approximation guarantee of $\frac{3}{2}$.

Algorithm 3.10 (Metric TSP – factor $\frac{3}{2}$)

1. Find an MST of G, say T.
2. Compute a minimum cost perfect matching, M, on the set of odd vertices of T. Add M to T and obtain an Eulerian graph.
3. Find an Euler tour, $\mathcal{T}$, of this graph.
4. Output the tour that visits vertices of G in order of their first appearence in $\mathcal{T}$. Let $\mathcal{C}$ be this tour.

Interestingly, the proof of this algorithm is based on a second lower bound on OPT.

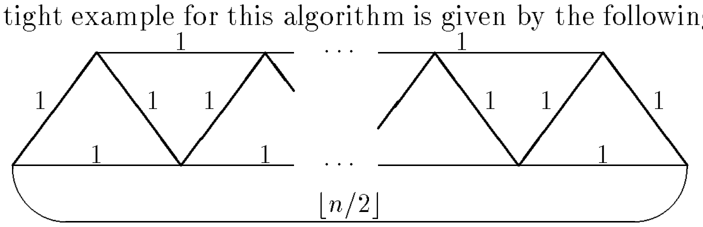
Lemma 3.11 *Let $V' \subseteq V$, such that $|V'|$ is even, and let M be a minimum cost perfect matching on V'. Then,* $\text{cost}(M) \leq \text{OPT}/2$.

Proof : Consider an optimal TSP tour of G, say τ. Let τ' be the tour on V' obtained by short-cutting τ. By triangle inequality, $\text{cost}(\tau') \leq \text{cost}(\tau)$. Now, τ' is the union of two perfect matchings on V', each consisting of alternate edges of τ. So, the cheaper of these matchings has cost $\leq \frac{\text{cost}(\tau')}{2} \leq \frac{\text{OPT}}{2}$. Hence the optimal matching also has cost $\leq \frac{\text{OPT}}{2}$. $\square$

Theorem 3.12 *Algorithm 3.10 achieves an approximation guarantee of $\frac{3}{2}$ for metric TSP.*

Proof : The proof follows by putting together the two lower bounds on OPT. $\square$

Example 3.13 A tight example for this algorithm is given by the following graph on n vertices:



Thick edges represent the MST found in step 1. This MST has only two odd vertices, and by adding the edge joining them we obtain a traveling salesman tour of cost $(n-1)+\lfloor n/2 \rfloor$. In contrast, the optimal tour has cost n. □

Finding a better approximation algorithm for metric TSP is currently one of the outstanding open problems in this area. Many researchers have conjectured that an approximation factor of 4/3 may be achievable.

Exercise 3.14 Consider the following variant of metric TSP: given vertices $u, v \in V$, find a minimum cost simple path from u to v that visits all vertices. First give a factor 2 approximation algorithm for this problem, and then improve it to factor $\frac{3}{2}$.

Exercise 3.15 Give a factor 2 approximation algorithm for: Given an undirected graph $G = (V, E)$, with non-negative edge costs, and a partitioning of V into two sets Senders and Recievers, find a minimum cost subgraph such that every Receiver vertex has a path to a Sender vertex.

Chapter 4

Multiway cuts and k-cuts

The theory of cuts occupies a central place in the study of exact algorithms. In this chapter, we will present approximation algorithms for natural generalizations of the minimum cut problem that are **NP**-hard.

Given a connected, undirected graph $G = (V, E)$ with an assignment of weights to edges, $w : E \to \mathbf{R}^+$, a *cut* is defined by a partition of V into two sets, say V' and $V - V'$, and consists of all edges that have one endpoint in each partition. Clearly, the removal of the cut from G disconnects G. Given *terminals* $s, t \in V$, consider a partition of V that separates s and t. The cut defined by such a partition will be called an *s–t cut*. The removal of such a cut from G will disconnect s and t. The problems of finding a minimum weight cut and a minimum weight s–t cut can be efficiently solved using a maximum flow algorithm. Let us generalize these two notions:

Problem 4.1 (Minimum k-cut) A set of edges whose removal leaves k connected components is called a *k-cut*. The minimum k-cut problem asks for a minimum weight k-cut.

Problem 4.2 (Multiway cut) Given a set of terminals $S = \{s_1, s_2, \ldots, s_k\} \subseteq V$, a *multiway cut* is a set of edges whose removal disconnects the terminals from each other. The multiway cut problem asks for the minimum weight such set.

The problem of finding a minimum weight multiway cut is **NP**-hard for any fixed $k \geq 3$. The minimum k-cut problem is polynomial time solvable for fixed k (though with a prohibitive running time of $O(n^{k^2/2})$); however, it is **NP**-hard if k is specified as part of the input. Interestingly enough, both problems admit approximation algorithms with the same guarantee, of $(2 - \frac{2}{k})$.

The multiway cut problem

Define an *isolating cut for s_i* to be a set of edges whose removal disconnects s_i from the rest of the terminals.

Algorithm 4.3 (Multiway cut)

1. For each $i = 1, \ldots, k$, compute a minimum weight isolating cut for s_i, say C_i.
2. Discard the heaviest of these cuts, and output the union of the rest, say C.

Each computation in Step 1 can be accomplished by identifying the terminals in $S - \{s_i\}$ into a single node, and finding a minimum cut separating this node from s_i; this takes one max-flow computation. Clearly, removing C from the graph disconnects every pair of terminals, and so is a multiway cut.

Theorem 4.4 *Algorithm 4.3 achieves an approximation guarantee of* $(2 - \frac{2}{k})$.

Proof : Let A be an optimal multiway cut in G. We can view A as the union of k cuts as follows: The removal of A from G will create k connected components, each having one terminal (since A is a minimum weight multiway cut, no more than k components will be created). Let A_i be the cut separating the component containing s_i from the rest of the graph. Then $A = \bigcup_{i=1}^{k} A_i$.

Since each edge of A is incident at two of these components, each edge will be in two of the cuts A_i. Hence,

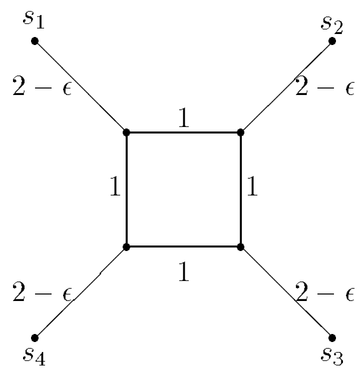
$$\sum_{i=1}^{k} w(A_i) = 2w(A).$$

Clearly, A_i is an isolating cut for s_i. Since C_i is a minimum weight isolating cut for s_i, $w(C_i) \leq w(A_i)$. Finally, since C is obtained by discarding the heaviest of the cuts C_i,

$$w(C) \leq \left(1 - \frac{1}{k}\right) \sum_{i=1}^{k} w(C_i) \leq \left(1 - \frac{1}{k}\right) \sum_{i=1}^{k} w(A_i) = 2\left(1 - \frac{1}{k}\right) w(A).$$

□

Example 4.5 A tight example for this algorithm is given by a graph on $2k$ vertices consisting of a k-cycle and a distinct terminal attached to each vertex of the cycle. The edges of the cycle have weight 1 and edges attaching terminals to the cycle have weight $2 - \epsilon$ for a small fraction $\epsilon > 0$. For example, the graph corresponding to $k = 4$ is:



For each terminal s_i, the minimum weight isolating cuts for s_i is given by the edge incident to s_i. So, the cut C returned by algorithm has weight $(k - 1)(2 - \epsilon)$. On the other hand, the optimal multiway cut is given by the cycle edges, and has weight k. □

Exercise 4.6 Show that the algorithm presented above can be used as a subroutine for finding a k-cut within a factor of $2 - 2/k$ of the minimum k-cut. How many subroutine calls are needed?

Exercise 4.7 A natural greedy algorithm for computing a multiway cut is the following: starting with G, compute minimum s_i–s_j cuts for all pairs s_i, s_j that are still connected and remove the lightest of these cuts; repeat this until all pairs s_i, s_j are disconnected. Prove that this algorithm also achieves a guarantee of $2 - 2/k$.

The minimum k-cut problem

A natural algorithm for finding a k-cut follows along the lines of the above-stated exercise: starting with G, compute a minimum cut in each currently connected component and remove the lightest one; repeat until there are k connected components. This algorithm does achieve a guarantee of $2 - 2/k$; however, the proof is quite involved. Instead we will use the *Gomory-Hu tree representation of minimum cuts* to give a simpler algorithm achieving the same guarantee.

Minimum cuts, as well as sub-optimal cuts, in undirected graphs have several interesting structural properties (as opposed to cuts in directed graphs). The existence of Gomory-Hu trees is one of the remarkable consequences of these properties.

Let T be a tree on vertex set V; the edges of T need not be in E. Each edge (u, v) in T defines a partition of V given by the two connected components obtained by removing (u, v). Consider the cut defined in G by this partition. We will be say that this is the *cut associated with* (u, v) *in* G. Define a weight function w' on the edges of T. Tree T will be said to be a Gomory-Hu tree for G if

1. for each edge $e \in T$, $w'(e)$ is the weight of the cut associated with e in G, and
2. for each pair of vertices $u, v \in V$, the weight of a minimum u–v cut is G is the same as that in T.

Clearly, a minimum u–v cut in T is given by a minimum weight edge on the unique path from u to v in T, say e, and the cut associated with e in G must separate u and v.

Intuitively, finding a light k-cut requires picking light cuts that create lots of components. Notice that the $n - 1$ cuts associated with the edges of T contain a minimum u–v cut for each of the $\binom{n}{2}$ pairs of vertices $u, v \in V$. This fact, together with the following lemma, justifies the use of Gomory-Hu trees for finding a light k-cut.

Lemma 4.8 *Let S be the union of cuts in G associated with l edges of T. Then, removal of S from G leaves a graph with at least $l + 1$ components.*

Proof : Removing the corresponding l edges from T leaves exactly $l + 1$ connected components, say with vertex sets $V_1, V_2, \ldots, V_{l+1}$. Clearly, removing S from G will disconnect each pair V_i and V_j. Hence we must get at least $l + 1$ connected components. □

A Gomory-Hu tree can be computed using $n - 1$ max-flow computations; see [?] for details. We will use this in the k-cut algorithm:

Algorithm 4.9 (Minimum k-cut)

1. Compute a Gomory-Hu tree T for G.
2. Output the union of the lightest $(k - 1)$ cuts of the $(n - 1)$ cuts associated with edges of T in G; let C be the union.

As shown in Lemma 4.8, the removal of C from G will leave at least k components. If more than k components are created, throw back some of the removed edges until there are exactly k components. The following theorem establishes the promised approximation guarantee.

Theorem 4.10 *Algorithm 4.9 achieves an approximation factor of* $(2 - \frac{2}{k})$.

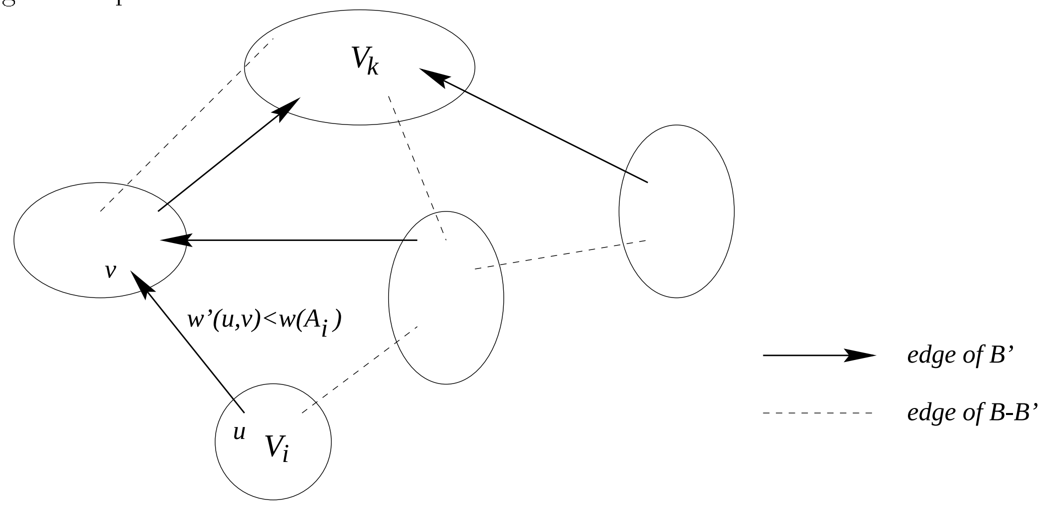
Proof : Let A be an optimal k-cut in G. As in Theorem 4.4, we can view A as the union of k cuts: Let $V_1, V_2, \ldots, V_k$ be the k components formed by removing A from G, and let A_i denote the cut separating V_i form the rest of the graph. Then $A = A_1 \cup \cdots \cup A_k$, and, since each edge of A lies in two of these cuts,

$$\sum_{i=1}^{k} w(A_i) = 2w(A).$$

Without loss of generality assume that A_k is the heaviest of these cuts. The idea behind the rest of the proof is to show that there are $k-1$ cuts defined by the edges of T whose weights are dominated by the weight of the cuts $A_1, A_2, \ldots, A_{k-1}$. Since the algorithm picks the lightest $k-1$ cuts defined by T, the theorem follows.

The $k-1$ cuts are identified as follows. Let B be the set of edges of T that connect across two of the sets $V_1, V_2, \ldots, V_k$. Consider the graph on vertex set V and edge set B, and shrink each of the sets $V_1, V_2, \ldots, V_k$ to a single vertex. This shrunk graph must be connected (since T was connected). Throw edges away until a tree remains. Let $B' \subseteq B$ be the left over edges, $|B'| = k-1$. The edges of B' define the required $k-1$ cuts.

Next, root this tree at V_k (recall that A_k was assumed to be the heaviest cut among the cuts A_i). This helps in defining a correspondence between the edges in B' and the sets $V_1, V_2, \ldots, V_{k-1}$: each edge corresponds to the set it comes out of in the rooted tree.



Suppose edge $(u, v) \in B'$ corresponds to set V_i in this manner. Now, since A_i is a u–v cut in G,

$$w(A_i) \geq w'(u, v).$$

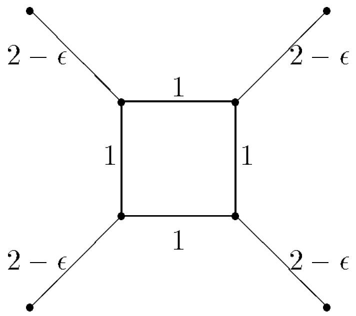
Thus each cut among $A_1, A_2, \ldots, A_{k-1}$ is at least as heavy as the cut defined in G by the corresponding edge of B'. This together with the fact that C is the union of the lightest $k-1$ cuts defined by T gives:

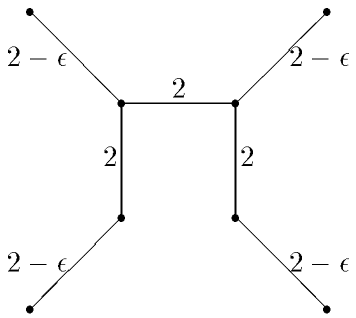
$$w(C) \leq \sum_{e \in B'} w'(e) \leq \sum_{i=1}^{k-1} w(A_i) \leq \left(1 - \frac{1}{k}\right) \sum_{i=1}^{k} w(A_i) = 2\left(1 - \frac{1}{k}\right) w(A).$$

□

Example 4.11 The tight example given above for multiway cuts on $2k$ vertices also serves as a tight example for the k-cut algorithm (of course, there is no need to mark vertices as terminals).

Below we give the example for $k = 4$, together with its Gomory-Hu tree.





The lightest $k - 1$ cuts in the Gomory-Hu tree have weight $2 - \epsilon$ each, corresponding to picking edges of weight $2 - \epsilon$ of G. So, the k-cut returned by the algorithm has weight $(k - 1)(2 - \epsilon)$. On the other hand, the optimal k-cut picks all edges of weight 1, and has weight k. □

Chapter 5

Facility location problems

In this chapter, we will study approximation algorithms for facility location problems. The flavour of these problems is illustrated by the following problem: Given a set of cities with inter-city distances specified, pick k cities for locating warehouses in, so as to minimize the maximum distance of a city from its closest warehouse. Formally,

Problem 5.1 (Minimum k-center problem) Given a complete undirected graph $G = (V, E)$, with costs on edges satisfying the triangle inequality, and an integer k, find a set $S \subseteq V$, $|S| = k$, so as to minimize

$$\max_{v \in V}\{\min_{u \in S}\{\mathrm{cost}(u, v)\}\}.$$

Let us first restate the problem in more convenient terms. Suppose that edges are indexed in non-decreasing order of cost, i.e., $\mathrm{cost}(e_1) \leq \mathrm{cost}(e_2) \leq \ldots \leq \mathrm{cost}(e_m)$, and let $G_i = (V, E_i)$, where $E_i = \{e_1, e_2, \ldots, e_i\}$. A *dominating set* of G is a subset $S \subseteq V$ such that every vertex in $V - S$ is adjacent to a vertex in S. Let $\mathrm{dom}(G)$ denote the size of a minimum cardinality dominating set in G; computing $\mathrm{dom}(G)$ is **NP**-hard. The k-center problem is then equivalent to finding the smallest index i such that G_i has a dominating set of size at most k, i.e., G_i contains k stars spanning all vertices, where a *star* is the graph $K_{1,p}$, with $p \geq 1$. If i^* is the smallest such index, then $\mathrm{cost}(e_{i^*})$ is the cost of an optimal k-center; this will be denoted by OPT.

Define the *square of graph* G, G^2, as the graph containing an edge (u, v) whenever G has a path of length at most two between u and v, $u \neq v$. The following structural result gives a method for lower bounding OPT:

Lemma 5.2 *Given a graph H, let I be an independent set in H^2. Then,*

$$|I| \leq \mathrm{dom}(H).$$

Proof : Let D be a minimum dominating set in H. Then, H contains $|D|$ stars spanning all vertices. Since each of these stars will be a clique in H^2, H^2 contains $|D|$ cliques spanning all vertices. Clearly, I can pick at most one vertex from each clique, and the lemma follows. □

The k-center algorithm is:

Algorithm 5.3 (Minimum k-center)

1. Construct $G_1^2, G_2^2, \ldots, G_m^2$.
2. Compute a maximal independent set, M_i, in each graph G_i^2.
3. Find the smallest index i such that $|M_i| \le k$, say j.
4. Return M_j.

The lower bound on which this algorithm is based is:

Lemma 5.4 *For j as defined in the algorithm,*

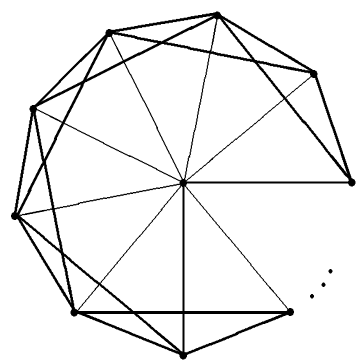
$$\mathrm{cost}(e_j) \le \mathrm{OPT}.$$

Proof : For every $i < j$ we have that $|M_i| > k$. Now, by Lemma 5.2, $\mathrm{dom}(G_i) > k$, and so $i^* > i$. Hence, $j \le i^*$. □

Theorem 5.5 *Algorithm 5.3 achieves an approximation factor of 2 for the minimum k-center problem.*

Proof : The key observation is that a maximal independent set, I, in a graph is also a dominating set (for, if some vertex v is not dominated by I, then $I \cup \{v\}$ must also be an independent set, contradicting the fact that I is a maximal independent set). So, there exist stars in G_j^2, centered on the vertices of M_j, covering all vertices. By triangle inequality, each edge used in constructing these stars has cost at most $2 \cdot \mathrm{cost}(e_j)$. The theorem follows from Lemma 5.4. □

Example 5.6 A tight example for the previous algorithm is given by a wheel graph on $n+1$ vertices, where all edges incident to the center vertex have cost 1, and the rest of the edges have cost 2:



(Here, thin edges have cost 1 and thick edges have cost 2; not all edges of cost 2 are shown.)

For $k = 1$, the optimal solution is the center of the wheel, and $\mathrm{OPT} = 1$. The algorithm will compute index $j = n$. Now, G_n^2 is a clique, and, if a peripheral vertex is chosen as the maximal independent set, then the cost of the solution found is 2. □

Exercise 5.7 Perhaps a more obvious scheme would have been finding a minimal dominating set, instead of a maximal independent set, in G_i^2 in Step 2. Show that this does not lead to a factor

2 algorithm for the k-center problem. In particular, notice that with this modification, the lower bounding method does not work, since Lemma 5.2 does not hold if I is picked to be a minimal dominating set in H^2.

Next, we will show that 2 is essentially the best approximation factor achievable for the minimum k-center problem.

Theorem 5.8 *Assuming* $\mathbf{P} \neq \mathbf{NP}$*, there is no polynomial time algorithm achieving a factor of* $2 - \epsilon$*,* $\epsilon > 0$*, for the minimum* k*-center problem.*

Proof : We will show that such an algorithm can solve the dominating set problem in polynomial time. Let an instance of the dominating set problem be specified by graph $G = (V, E)$ and integer k. Construct a complete graph $G' = (V, E')$ with edge costs given by

$$\mathrm{cost}(u, v) = \begin{cases} 1, & \text{if } (u, v) \in E, \\ 2, & \text{if } (u, v) \notin E. \end{cases}$$

Clearly, if $\mathrm{dom}(G) \leq k$, then G' has a k-center of cost 1. On the other hand, if $\mathrm{dom}(G) > k$, then the optimum cost of a k-center in G' is 2. In the first case, when run on G', the $(2 - \epsilon)$-approximation algorithm must give a solution of cost 1, since it cannot use an edge of cost 2. Hence, using this algorithm, we can distinguish between the two possibilities, thus solving the dominating set problem. □

The weighted k-center problem

Let us generalize the k-center problem as follows:

Problem 5.9 (Weighted k-center) In addition to a cost function on edges, we are given a weight function on vertices, $w : V \to R^+$, and a bound $W \in R^+$. The problem is to pick $S \subseteq V$ of total weight at most W, minimizing the same objective function as before, i.e.,

$$\max_{v \in V}\{\min_{u \in S}\{\mathrm{cost}(u, v)\}\}.$$

Let $\mathrm{wdom}(G)$ denote the weight of a minimum weight dominating set in G. Then, with respect to the graphs G_i defined above, we need to find the smallest index i such that $\mathrm{wdom}(G_i) \leq W$. If i^* is this index, then the cost of the optimal solution is $\mathrm{OPT} = \mathrm{cost}(e_{i^*})$.

Given a vertex weighted graph H, let I be an independent set in H^2. For each $u \in I$, let $s(u)$ denote a lightest neighbor of u in H, where u is also considered a neigbor of itself. (Notice that the neighbor is picked in H and not in H^2.) Let $S = \{s(u) | \; u \in H\}$. The following fact, analogous to Lemma 5.2, will be used to derive a lower bound on OPT:

Lemma 5.10 $w(S) \leq \mathrm{wdom}(H)$.

Proof : Let D be a minimum weight dominating set of H. Then, there exist a set of disjoint stars in H, centered on the vertices of D and covering all the vertices. Since each of these stars becomes a clique in H^2, I can pick at most one vertex from each of them. So, each vertex in I has the center of the corresponding star available as a neighbor in H. Hence, $w(S) \leq w(D)$. □

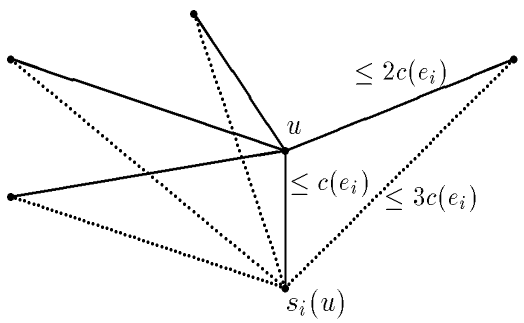
The algorithm is given below. In it, $s_i(u)$ will denote a lightest neighbor of u in G_i; for this definition, u will be considered a neigbor of itself as well.

Algorithm 5.11 (Weighted k-center)

1. Construct $G_1^2, G_2^2, \ldots, G_m^2$.
2. Compute a maximal independent set, M_i, in each graph G_i^2.
3. Compute $S_i = \{s_i(u) |\ u \in M_i\}$.
4. Find the minimum index i such that $w(S_i) \leq W$, say j.
5. Return S_j.

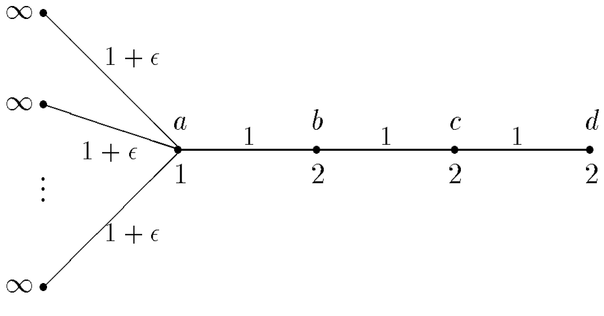
Theorem 5.12 *Algorithm 5.11 achieves an approximation factor of 3 for the weighted k-center problem.*

Proof : Lemma 5.10 gives a lower bound on OPT: $\mathrm{cost}(e_j) \leq \mathrm{OPT}$; the argument is identical to that in Lemma 5.4 and is omitted. Since M_j is a dominating set in G_j^2, we can cover V with stars of G_j^2 centered in vertices of M_j. By triangle inequality these stars use edges of cost at most $2 \cdot \mathrm{cost}(e_j)$.



Each star center is adjacent to a vertex in S_j, using an edge of cost at most $\mathrm{cost}(e_j)$. Move each of the centers to the adjacent vertex in S_j, and redefine the stars. Again, by triangle inequality, the largest edge cost used in constructing the final stars is at most $3 \cdot \mathrm{cost}(e_j)$. $\square$

Example 5.13 A tight example is provided by the following graph on $n + 4$ vertices. Vertex weights and edge costs are as marked; all missing edges have cost given by the shortest path.



It is not difficult to see that for $W = 3$, the optimum cost of a k-center is $1 + \epsilon$: a k-center achieving this cost is $\{a, c\}$. For any $i < n + 3$, the set S_i computed by the algorithm will contain a vertex of infinite weight. Suppose that, for $i = n + 3$, the algorithm chooses $M_{n+3} = \{b\}$ as a maximal independent set. Then $S_{n+3} = \{a\}$, and this is the output of the algorithm. The cost of this solution is 3. □

The *k-median* problem is: Given a complete undirected graph $G = (V, E)$, with costs on edges satisfying the triangle inequality, and an integer k, find a set $S \subseteq V$, $|S| = k$, and a mapping $f : V \to S$ so as to minimize $\sum_{v \in V} \mathrm{cost}(v, f(v))$. Surprisingly enough, no approximation algorithm achieving a non-trivial factor is currently known for this problem.

Chapter 6

Feedback vertex set

Problem 6.1 (Feedback vertex set) Given an undirected graph $G = (V, E)$ and a function w assigning non-negative weights to its vertices, find a minimum weight subset of V whose removal leaves an acyclic graph.

We present a factor 2 approximation algorithm for this **NP**-hard problem. An interesting feature of the algorithm is that we first derive a lower bound on OPT for special vertex-weight functions; the given vertex weights are then decomposed into these special weights.

Order the edges of G in an arbitrary order. The *characteristic vector* of a simple cycle C in G is a vector in $GF[2]^m$, $m = |E|$, which has 1's in components corresponding to edges of C and 0's in the remaining components. The *cycle space* of G is the subspace of $GF[2]^m$ that is spanned by the characteristic vectors of all simple cycles of G, and the *cyclomatic number of* G, denoted $\mathrm{cyc}(G)$, is the dimension of this space.

Theorem 6.2 $\mathrm{cyc}(G) = |E| - |V| + \kappa(G)$, *where* $\kappa(G)$ *denotes the number of connected components of* G.

Proof : The cycle space of a graph is the direct sum of the cycle spaces of its connected components, and so its cyclotomic number is the sum of the cyclotomic numbers of its connected components. Therefore, it is sufficient to prove the theorem for a connected graph G.

Let T be a spanning tree in G. For each non-tree edge e, define its *fundamental cycle* to be the unique cycle formed in $T \cup \{e\}$. The set of characteristic vectors of all such cycles is linearly independent (each cycle includes an edge that is in no other fundamental cycle). So, $\mathrm{cyc}(G) \geq |E| - |V| + 1$.

Each edge e of T defines a *fundamental cut* $(S, \overline{S})$ in G, $S \subset V$ (S and $\overline{S}$ are the vertex sets of two connected components formed by removing e from T). Define the *characteristic vector* of a cut to be a vector in $GF[2]^m$, that has 1's in components corresponding to the edges of G in the cut and 0's in the remaining components. Consider the $|V| - 1$ vectors defined by edges of T. Since each cycle must cross each cut an even number of times, these vectors are orthogonal to the cycle space of G. Furthermore, these $|V| - 1$ vectors are linearly independent, since each cut has an edge (the tree edge defining this cut) that is not in any of the other $|V| - 2$ cuts. Therefore the dimension of the orthogonal complement to the cycle space is at least $|V| - 1$. Hence, $\mathrm{cyc}(G) \leq |E| - |V| + 1$. Combining with the previous inequality we get $\mathrm{cyc}(G) = |E| - |V| + 1$. □

Denote by $\delta_G(v)$ the decrease in the cyclomatic number of the graph on removing vertex v. Since the removal of a feedback vertex set $F = \{v_1, \ldots, v_f\}$ decreases the cyclomatic number of G

down to 0,

$$\mathrm{cyc}(G) = \sum_{i=1}^{f} \delta_{G_{i-1}}(v_i),$$

where $G_0 = G$ and, for $i > 0$, $G_i = G - \{v_1, \ldots, v_i\}$. By Lemma 6.4 below, we get:

$$\mathrm{cyc}(G) \leq \sum_{v \in F} \delta_G(v). \tag{6.1}$$

Let us say that a function assigning vertex weights is *cyclomatic* if there is a constant $c > 0$ such that the weight of each vertex v is $c \cdot \delta_G(v)$. By inequality (6.1), for such a weight function, $c \cdot \mathrm{cyc}(G)$ is a lower bound on OPT. The importance of cyclotomic weight functions is established in Lemma 6.5 below, which shows that for such a weight function, any minimal feedback vertex set has weight within twice the optimal.

Let $\deg_G(v)$ denote the degree of v in G, and $\mathrm{comps}(G - v)$ denote the number of connected components formed by removing v from G. The claim below follows in a straightforward way by applying Theorem 6.2 to G and $G - v$:

Claim 6.3 *For a connected graph G, $\delta_G(v) = \deg_G(v) - \mathrm{comps}(G - v)$.*

Lemma 6.4 *Let H be a subgraph of G (not necessarily vertex induced). Then, $\delta_H(v) \leq \delta_G(v)$.*

Proof : It is sufficient to prove the lemma for the connected components of G and H containing v. So, we may assume w.l.o.g. that G and H are connected (H may be on a smaller set of vertices). By Claim 6.3, proving the following inequality is sufficient:

$$\deg_H(v) - \mathrm{comps}(H - v) \leq \deg_G(v) - \mathrm{comps}(G - v).$$

We will show that edges in $G - H$ can only help this inequality. Let $c_1, c_2, \ldots, c_k$ be components formed by removing v from H. Edges of $G - H$ not incident at v can only help merge some of these components (and of course, they don't change the degree of v). An edge of $G - H$ that is incident at v can lead to an additional component, but this is compensated by the additional degree it provides to v. □

Lemma 6.5 *If F is a minimal feedback vertex set of G, then*

$$\sum_{v \in F} \delta_G(v) \leq 2 \cdot \mathrm{cyc}(G).$$

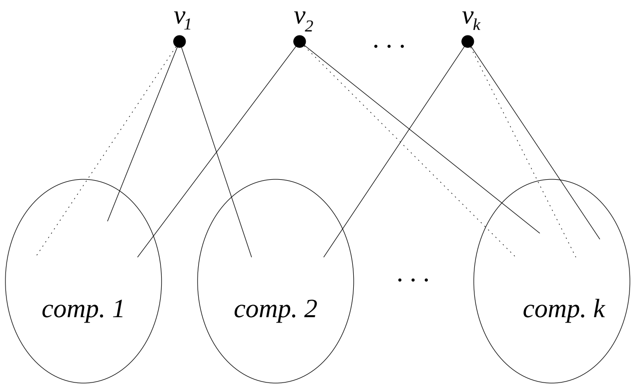
Proof : Since the cycle space of G is the direct sum of the cycle spaces of its connected components, it suffices to prove the lemma for a connected graph G.

Let $F = \{v_1, \ldots, v_f\}$, and let k be the number of connected components obtained by deleting F from G. Partition these components into two types: those that have edges incident to only one of the vertices of F, and those that have edges incident to two or more vertices of F. Let t and $k - t$ be the number of components of the first and second type respectively. We will prove that

$$\sum_{i=1}^{f} \delta_G(v_i) = \sum_{i=1}^{f} (\deg_G(v_i) - \mathrm{comps}(G - v_i)) \leq 2(|E| - |V|),$$

thereby proving the lemma. Clearly, $\sum_{i=1}^{f} \text{comps}(G - v_i) = f + t$. Therefore, we are left to prove

$$\sum_{i=1}^{f} \deg_G(v_i) \leq 2(|E| - |V|) + f + t.$$



Since F is a feedback vertex set, each of the k components is acyclic, and is therefore a tree. So, the number of edges in these components is $|V| - f - k$. Next, we put a lower bound on the number of edges in the cut $(F, V - F)$. Since F is minimal, each $v_i \in F$ must be in a cycle that contains no other vertices of F. So, each v_i must have at least two edges incident at one of the components. For each v_i, arbitrarily remove one of these edges from G, thus removing a total of f edges. Now, each of the t components must still have at least one edge and each of the $k - t$ components must still have at least two edges incident at F. Therefore, the number of edges in the cut $(F, V - F)$ is at least $f + t + 2(k - t) = f + 2k - t$.

These two facts imply that

$$\sum_{i=1}^{f} \deg_G(v_i) \leq 2|E| - 2(|V| - f - k) - (f + 2k - t).$$

The lemma follows. $\square$

Corollary 6.6 *Let w be a cyclotomic weight function on the vertices of G, and let F be a minimal feedback vertex set in it. Then $w(F) \leq 2 \cdot \text{OPT}$.*

Finally, let us deal with arbitrary weights. Consider the following basic operation: Given graph $G = (V, E)$ and a weight function w, let

$$c = \min_{v \in V} \left\{ \frac{w(v)}{\delta_G(v)} \right\}.$$

The weight function $t(v) = c\delta_G(v)$ is the *largest cyclotomic weight function in w*. Define $w'(v) = w(v) - t(v)$ to be the *residual weight function*. Finally, let V' be the set of vertices having positive residual weight (clearly, $V' \subset V$), and let G' be the subgraph of G induced on V'.

Using this basic operation, decompose G into a "telescoping" sequence of induced subgraphs, until an acyclic graph is obtained, each time finding the largest cyclotomic weight function in the current residual weight function. Let these graphs be $G = G_0 \supset G_1 \supset \cdots \supset G_k$, where G_k is acyclic; G_i is the induced subgraph of G on vertex set V_i, where $V = V_0 \supset V_1 \supset \cdots \supset V_k$. Let $t_i, i = 0, \ldots, k-1$ be the cyclotomic weight function defined on graph G_i. Thus, $w_0 = w$ is the residual weight function for G_0, t_0 is the largest cyclotomic weight function in w_0, $w_1 = w_0 - t_0$ is the residual weight function for G_1, and so on. Finally, w_k is the residual weight function for G_k. For convenience, define $t_k = w_k$. Since the weight of a vertex v has been decomposed into the weights $t_0, t_1, \dots t_k$, we have

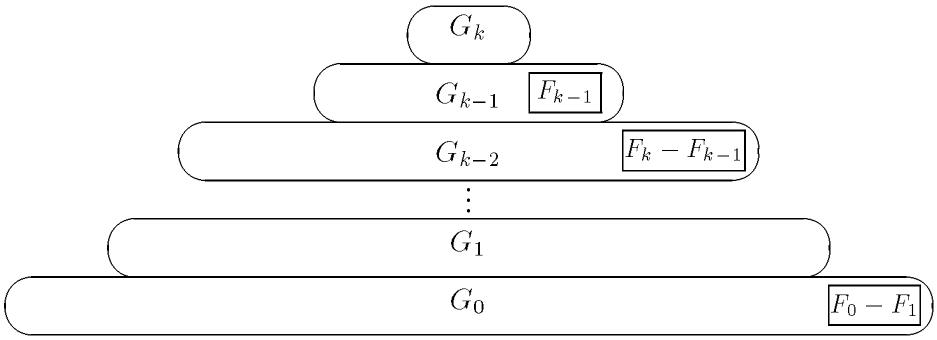
$$\sum_{i:\, v \in V_i} t_i(v) = w(v).$$

The next fact suggests an algorithm for constructing a feedback vertex set on which Lemma 6.5 can be applied:

Lemma 6.7 *Let H be a subgraph of $G = (V, E)$, induced on vertex set $V' \subset V$. Let F be a minimal feedback vertex set in H, and let $F' \subseteq V - V'$ be a minimal set such that $F \cup F'$ is a feedback vertex set for G. Then $F \cup F'$ is a minimal feedback vertex set for G.*

Proof : Since F is minimal for H, for each $v \in F$, there is a cycle, say C, in H that does not use any other vertex of F. Since $F' \cap V' = \emptyset$, C uses only one vertex, v, from $F \cup F'$ as well, and so v is not redundant. □

After the entire decomposition, $F_k = \emptyset$ is a minimal feedback vertex set of G_k. For $i = k, k-1, \dots, 1$, the minimal feedback vertex set F_i found in G_i is extended in a minimal way using vertices of $V_{i-1} - V_i$ to yield a minimal feedback vertex set, say F_{i-1}, for G_{i-1}. The last set, F_0, is a feedback vertex set for G.



The algorithm is described in detail below. For future reference, let us call this process of decomposing G into a telescoping sequence of subgraphs and building a set F from the smallest graph outwards as *layering*.

Algorithm 6.8 (Feedback vertex set)

1. Decomposition phase:

   $H \leftarrow G$, $w' \leftarrow w$, $i \leftarrow 0$

   While H is not acyclic,

   $c \leftarrow \min_{u \in H} \left\{ \frac{w'(u)}{\delta_H(u)} \right\}$

   $G_i \leftarrow H$, $t_i \leftarrow c \cdot \delta_{G_i}$, $w' \leftarrow w' - t_i$

   $H \leftarrow$ the subgraph of G_i induced by vertices u with $w'(u) > 0$

   $i \leftarrow i + 1$,

   $k \leftarrow i$, $G_k \leftarrow H$

2. Extension phase

   $F_k \leftarrow \emptyset$

   For $i = k, \ldots, 1$, extend F_i to a feedback vertex set F_{i-1} of G_{i-1} by adding a minimal set of vertices from $V_{i-1} - V_i$.

   Output F_0

Theorem 6.9 *Algorithm 6.8 achieves an approximation guarantee of factor 2 for the feedback vertex set problem.*

Proof : Let F^* be an optimal feedback vertex set for G. Since G_i is an induced subgraph of G, $F^* \cup V_i$ must be a feedback vertex set for G_i (not necessarily optimal). Since the weights of vertices have been decomposed into the functions t_i, we have

$$\mathrm{OPT} = w(F^*) = \sum_{i=0}^{k} t_i(F^* \cap V_i) \geq \sum_{i=0}^{k} \mathrm{OPT}_i,$$

where OPT_i is the weight of an optimal feedback vertex set of G_i with weight function t_i.

Decomposing the weight of F_0, we get:

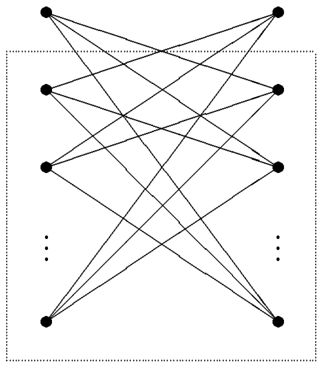
$$w(F_0) = \sum_{i=0}^{k} t_i(F_0 \cap V_i) = \sum_{i=0}^{k} t_i(F_i).$$

By Lemma 6.7, F_i is a minimal feedback vertex set in G_i. Since for $0 \leq i \leq k-1$, t_i is a cyclotomic weight function, by Lemma 6.5, $t_i(F_i) \leq 2\mathrm{OPT}_i$; recall that $F_k = \emptyset$. Therefore,

$$w(F_0) \leq 2 \sum_{i=0}^{k} \mathrm{OPT}_i \leq 2 \cdot \mathrm{OPT}.$$

□

Example 6.10 A tight example for the algorithm is given by the graph obtained by removing a perfect matching from a complete bipartite graph and duplicating every edge. (Note that the algorithm works for parallel edges as well. If a tight example without parallel edges is desired, then a vertex with very high weight can be placed on every edge.):



Assuming that the graph is cyclomatic weighted, each vertex receives the same weight. The decomposition obtained by the algorithm consists of only one non-trivial graph, G itself, on which the algorithm computes a minimal feedback vertex set. A possible output of the algorithm is the set shown above; this set contains $2n-2$ vertices as compared with the optimum of n given by one side of the bipartition. $\square$

Remark: We can reduce the weighted vertex cover problem to the minimum feedback vertex set problem by duplicating every edge and then placing a very high weight vertex on each edge. Hence, any improvement to the approximation factor for the minimum feedback vertex set problem will carry over to the weighted vertex cover problem.

Exercise 6.11 A natural greedy algorithm for finding a minimum feedback vertex set is to repeatedly pick and remove the most cost-effective vertex, i.e., a vertex minimizing $\frac{w(v)}{\delta_H(v)}$, where H is the current graph, until there are no more cycles left. Give examples to show that this is not a constant factor algorithm. What is the approximation guarantee of this algorithm?

Exercise 6.12 Obtain a factor 2 approximation algorithm for the weighted vertex cover problem using the technique of layering. Use the following obvious fact to construct the layering: if the weight of each vertex is equal to its degree in G, i.e., the graph is *degree weighted*, then the weight of any vertex cover is within twice the optimal.

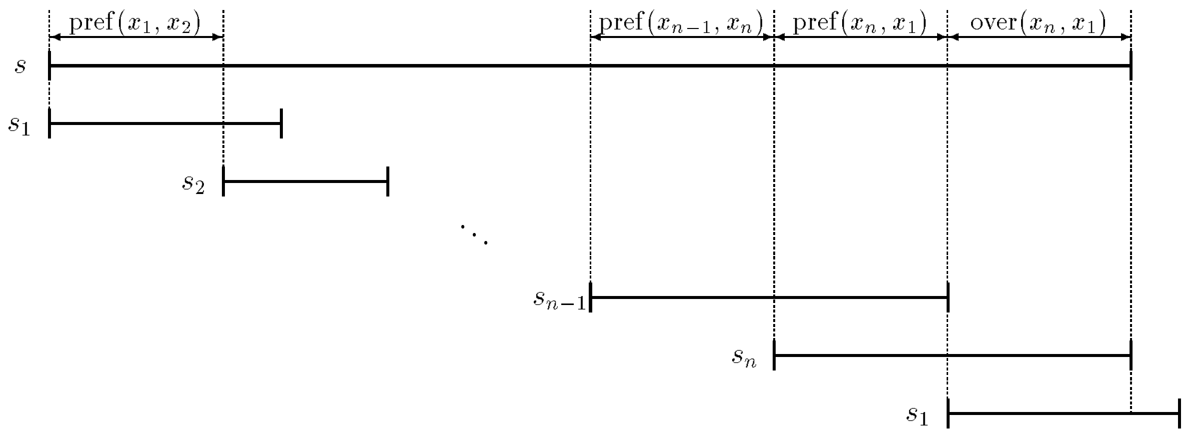
Chapter 7

Shortest superstring

In Chapter 2 we defined the shortest superstring problem and gave a $2 \cdot H_n$ factor algorithm for it by reducing to the set cover problem. In this chapter, we will first give a factor 4 algorithm, and then we will improve this to factor 3.

A factor 4 algorithm

We begin by developing a good lower bound on OPT. Let us assume that $s_1, s_2, \ldots, s_n$ are numbered in order of leftmost occurrence in the shortest superstring, s.



Let overlap(s_i, s_j) denote the maximum overlap between s_i and s_j, i.e., the longest suffix of s_i that is a prefix of s_j. Also, let prefix(s_i, s_j) be the prefix of s_i obtained by removing its overlap with s_j. The overlap in s between two consecutive s_i's is maximum possible, because otherwise a shorter superstring can be obtained. Hence, assuming that no s_i is substring of another, we get

$$\mathrm{OPT} = |\mathrm{prefix}(s_1, s_2)| + |\mathrm{prefix}(s_2, s_3)| + \ldots + |prefix(s_n, s_1)| + |\mathrm{overlap}(s_n, s_1)|. \tag{7.1}$$

Notice that we have repeated s_1 in the end in order to obtain the last two terms of (7.1). This equality shows the close relation between the shortest superstring of S and the minimum traveling salesman tour on the *prefix graph of* S, defined as the directed graph on vertex set $\{1, \ldots, n\}$ that contains an edge $i \to j$ of weight $|\mathrm{prefix}(s_i, s_j)|$ for each i, j (self loops included). Clearly, $|\mathrm{prefix}(s_1, s_2)| + |\mathrm{prefix}(s_2, s_3)| + \ldots + |prefix(s_n, s_1)|$ represents the weight of the tour

$1 \to 2 \to \ldots \to n \to 1$. Hence, by (7.1), the minimum weight of a traveling salesman tour of the prefix graph gives a lower bound on OPT. As such, this lower bound is not very useful, since we cannot efficiently compute a minimum traveling salesman tour.

The key idea is to lower bound OPT using the minimum weight of a *cycle cover* of the prefix graph (a cycle cover is a collection of disjoint cycles covering all vertices). Since the tour $1 \to 2 \to \ldots \to n \to 1$ is a cycle cover, from (7.1) we get that the minimum weight of a cycle cover lower bounds OPT.

Unlike minimum TSP, a minimum weight cycle cover can be computed in polynomial time. We first construct a bipartite version of the prefix graph: let $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$ be the two sides of the bipartition, for each $i, j \in \{1, \ldots, n\}$ add an edge of weight $|\text{prefix}(s_i, s_j)|$ from u_i to v_j. It is easy to see that each cycle cover of the prefix graph corresponds to a perfect matching of the same weight in this bipartite graph, and vice versa. Hence, finding a minimum weight cycle cover reduces to finding a minimum weight perfect matching in the bipartite graph.

If $c = (i_1 \to i_2 \to \ldots i_l \to i_1)$ is a cycle in the prefix graph, let

$$\alpha(c) = \text{prefix}(s_{i_1}, s_{i_2}) \circ \ldots \circ \text{prefix}(s_{i_{l-1}}, s_{i_l}) \circ \text{prefix}(s_{i_l}, s_{i_1}).$$

Notice that each string $s_{i_1}, s_{i_2}, \cdots s_{i_l}$ is a substring of $(\alpha(c))^\infty$. Next, let

$$\sigma(c) = \alpha(c) \circ s_{i_1}.$$

Then $\sigma(c)$ is a superstring of $s_{i_1}, \ldots, s_{i_l}$[1]. In the above construction, we "opened" cycle c at an arbitrary string s_{i_1}. For the rest of the algorithm, we will call s_{i_1} the *representative string* for c. We can now state the complete algorithm:

Algorithm 7.1 (Shortest superstring – factor 4)

1. Construct the prefix graph corresponding to strings in S.
2. Find a minimum cycle cover of the prefix graph, $\mathcal{C} = \{c_1, \ldots, c_k\}$.
3. Output $\sigma(c_1) \circ \ldots \circ \sigma(c_k)$.

Clearly, the output is a superstring of the strings in S. Notice that if in each of the cycles we can find a representative string of length at most the weight of the cycle, then the string output is within $2 \cdot \text{OPT}$. So, the hard case is when all strings of some cycle c are long. But since they must all be substrings of $(\alpha(c))^\infty$, they must be periodic. This will be used to prove Lemma 7.4, which establishes another lower bound on OPT. This in turn will give:

Theorem 7.2 *Algorithm 7.1 achieves an approximation factor of 4 for the shortest superstring problem.*

Proof : Let $\text{wt}(\mathcal{C}) = \sum_{i=1}^{k} \text{wt}(c_i)$. The output of the algorithm has length

$$\sum_{i=1}^{k} |\sigma(c_i)| = \text{wt}(\mathcal{C}) + \sum_{i=1}^{k} |r_i|,$$

[1]This remains true even for the shorter string $\alpha(c) \circ \text{overlap}(s_l, s_1)$. We will work with $\sigma(c)$, since it will be needed for the factor 3 algorithm presented in next section, where we use the property that $\sigma(c)$ begins and ends with a copy of s_{i_1}.

where r_i denotes the representative string from cycle c_i. We have shown that $\mathrm{wt}(\mathcal{C}) \leq \mathrm{OPT}$. Next, we show that the sum of the lengths of representative strings is at most $3 \cdot \mathrm{OPT}$.

Assume that $r_1, \ldots, r_k$ are numbered in order of their leftmost occurrence in the shortest superstring of S. Using Lemma 7.4, we get the following lower bound on OPT:

$$\mathrm{OPT} \geq \sum_{i=1}^{k} |r_i| - \sum_{i=1}^{k-1} |\mathrm{overlap}(r_i, r_{i+1})| \geq \sum_{i=1}^{k} |r_i| - 2\sum_{i=1}^{k} \mathrm{wt}(c_i).$$

Hence,

$$\sum_{i=1}^{k} |r_i| \leq \mathrm{OPT} + 2\sum_{i=1}^{k} \mathrm{wt}(c_i) \leq 3 \cdot \mathrm{OPT}.$$

□

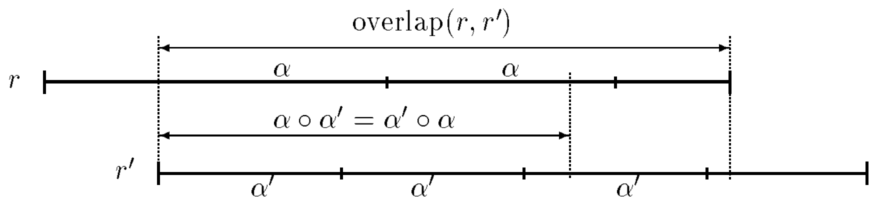
Lemma 7.3 *If each string in $S' \subseteq S$ is a substring of t^∞ for a string t, then there is a cycle of weight at most $|t|$ in the prefix graph covering all the vertices corresponding to strings in S'.*

Proof : For each string in S', locate the starting point of its first occurrence in t^∞. Clearly, all these starting points will be distinct (since no string in S is a substring of another) and will lie in the first copy of t. Consider the cycle in the prefix graph visiting the corresponding vertices in this order. Clearly, the weight of this cycle is at most $|t|$. □

Lemma 7.4 *Let c and c' be two cycles in $\mathcal{C}$, and let r, r' be representative strings from these cycles. Then*

$$|\mathrm{overlap}(r, r')| < \mathrm{wt}(c) + \mathrm{wt}(c').$$

Proof : Suppose, for contradiction, that $|\mathrm{overlap}(r, r')| \geq \mathrm{wt}(c) + \mathrm{wt}(c')$. Denote by α (α') the prefix of length $\mathrm{wt}(c)$ ($\mathrm{wt}(c')$, respectively) of $\mathrm{overlap}(r, r')$.



Clearly, $\mathrm{overlap}(r, r')$ is a prefix of both α^∞ and $(\alpha')^\infty$. In addition, α is a prefix of $(\alpha')^\infty$ and α' is a prefix of α^∞. Since $\mathrm{overlap}(r, r') \geq |\alpha| + |\alpha'|$, it follows that α and α' commute, i.e., $\alpha \circ \alpha' = \alpha' \circ \alpha$. Bur then, $\alpha^\infty = (\alpha')^\infty$. This is so because for any $k > 0$,

$$\alpha^k \circ (\alpha')^k = (\alpha')^k \circ \alpha^k.$$

Hence, for any $N > 0$, the prefix of length N of α^∞ is the same as that of $(\alpha')^\infty$.

Now, by Lemma 7.3, there is a cycle of weight at most $\mathrm{wt}(c)$ in the prefix graph covering all strings in c and c', contradicting the fact that $\mathcal{C}$ is a minimum weight cycle cover. □

Exercise 7.5 Show that Lemma 7.4 cannot be strengthened to

$$|\text{overlap}(r, r')| < \max\{\text{wt}(c), \text{wt}(c')\}.$$

Improving to factor 3

Notice that any superstring of the strings $\sigma(c_i)$, $i = 1, \ldots, k$, is also a superstring of all strings in S. So, instead of simply concatenating these strings, let us make them overlap as much as possible (this may sound circular, but it is not!).

Let us define the *compression* achieved by a superstring as the difference between the total length of the input strings and the length of the resulting superstring. Clearly, maximum compression is achieved by the shortest superstring. Several algorithms are known to achieve at least half the optimal compression. For instance, the greedy superstring algorithm presented in a previous lecture does so, though its proof is based on a complicated case analysis. A less efficient algorithm, based on cycle cover, is presented at the end.

Algorithm 7.6 (Shortest superstring – factor 3)

1. Construct the prefix graph corresponding to strings in S.
2. Find a minimum cycle cover of the prefix graph, $\mathcal{C} = \{c_1, \ldots, c_k\}$.
3. Run the greedy algorithm on $\{\sigma(c_1), \ldots, \sigma(c_k)\}$ and output the resulting string, $\overline{\sigma}$.

Let OPT_σ denote the length of the shortest superstring of the strings in $S_\sigma = \{\sigma(c_1) \ldots \sigma(c_k)\}$, and let r_i be the representative string of c_i.

Lemma 7.7 $|\overline{\sigma}| \leq \text{OPT}_\sigma + \text{wt}(\mathcal{C})$.

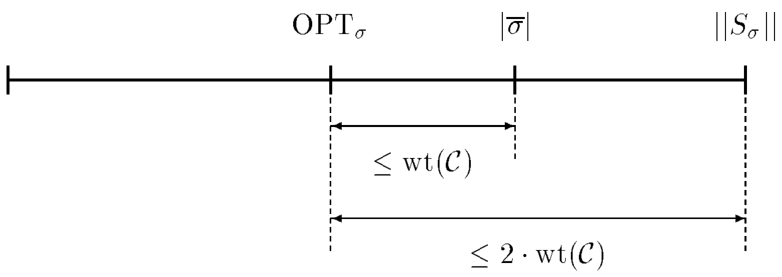
Proof : Assuming that $\sigma(c_1), \ldots, \sigma(c_k)$ appear in this order in a shortest superstring of S_σ, the maximum compression that can be achieved on S_σ is given by

$$\sum_{i=1}^{k-1} |\text{overlap}(\sigma(c_i), \sigma(c_{i+1}))|.$$

Since each string $\sigma(c_i)$ has r_i as a prefix as well as suffix, by Lemma 7.4,

$$|\text{overlap}(\sigma(c_i), \sigma(c_{i+1}))| \leq \text{wt}(c_i) + \text{wt}(c_{i+1}).$$

Hence, the maximum compression achievable on S_σ is at most $2\cdot\text{wt}(\mathcal{C})$, i.e., $||S_\sigma|| - \text{OPT}_\sigma \leq 2\cdot\text{wt}(\mathcal{C})$, where $||X||$ denotes the sum of the lengths of the strings in X.



The compression achieved by the greedy algorithm on S_σ is at least half the maximum compression, hence $|\overline{\sigma}|$ is closer to OPT_σ than to $||S_\sigma||$, and the lemma follows. □

Finally, we relate OPT_σ to OPT.

Lemma 7.8 $\mathrm{OPT}_\sigma \leq \mathrm{OPT} + \mathrm{wt}(\mathcal{C})$.

Proof : Let OPT_r denote the length of the shortest superstring of the strings in $S_r = \{r_1, \ldots, r_k\}$. Because each $\sigma(c_i)$ begins and ends with r_i, the compression achievable on the strings of S_σ is at least as large as that achievable on the strings of S_r. Thus,

$$||S_\sigma|| - \mathrm{OPT}_\sigma \geq ||S_r|| - \mathrm{OPT}_r.$$

Clearly, $||S_\sigma|| = ||S_r|| + \mathrm{wt}(\mathcal{C})$. This gives

$$\mathrm{OPT}_\sigma \leq \mathrm{OPT}_r + \mathrm{wt}(\mathcal{C}).$$

The lemma follows by noticing that $\mathrm{OPT}_r \leq \mathrm{OPT}$. □

Combining the previous two lemmas we get:

Theorem 7.9 *Algorithm 7.6 achieves an approximation factor of 3 for the shortest superstring problem.*

Finally, we present an algorithm achieving at least half the optimal compression. Suppose that the strings to be compressed, $s_1, \cdots, s_k$, are numbered in the order in which they appear in a shortest superstring. Then, the optimal compression is given by

$$\sum_{i=1}^{k-1} |\mathrm{overlap}(\sigma_i, \sigma_{i+1})|.$$

This is the weight of the traveling salesman path $1 \rightarrow 2 \rightarrow \ldots \rightarrow k$ in the *overlap graph* of the strings $s_1, \cdots, s_k$; this graph contains an arc $i \rightarrow j$ of weight $|\mathrm{overlap}(s_i, s_j)|$ for each $i \neq j$ (thus this graph has no self loops). Hence, the optimal compression is upper bounded by the maximum traveling salesman tour in the overlap graph, which in turn is upper bounded by the maximum cycle cover. The latter can be computed in polynomial time using matching, similar to a minimum weight cycle cover. the associated bipartite graph and finding a maximum weight perfect matching in it. Since the overlap graph has no self loops, each cycle has length at least 2. So, the lightest edge in each cycle has weight at most half that of the cycle. Open each cycle by discarding the lightest edge, overlap strings in the order given by remaining paths, and concatenate the resulting strings. This gives a superstring achieving compression of at least half the weight of the cycle cover, hence also half the optimal compression.

Chapter 8

Knapsack

In Chapter 10 we had mentioned that some **NP**-hard optimization problems allow approximability to any required degree. In this chapter, we will formalize this notion and will show that the knapsack problem admits such an approximability.

Definition 8.1 Let Π be an **NP**-hard optimization problem with objective function f_Π. We will say that algorithm $\mathcal{A}$ is an *approximation scheme* for Π if on input (I, ϵ), where I is an instance of Π and $\epsilon > 0$ is an error parameter, it outputs a solution s such that:

- $f_\Pi(I, s) \leq (1 + \epsilon) \cdot \mathrm{OPT}$ if Π is a minimization problem.
- $f_\Pi(I, s) \geq (1 - \epsilon) \cdot \mathrm{OPT}$ if Π is a maximization problem.

$\mathcal{A}$ will be said to be a *polynomial time approximation scheme*, abbreviated PTAS, if for each *fixed* $\epsilon > 0$, its running time is bounded by a polynomial in the size of instance I.

The definition given above allows the running time of $\mathcal{A}$ to depend arbitrarily on ϵ. This is rectified in the following more stringent notion of approximability:

Definition 8.2 If in the previous definition we require that the running time of $\mathcal{A}$ be bounded by a polynomial in the size of instance I and $\frac{1}{\epsilon}$, then $\mathcal{A}$ will be said to be a *fully polynomial approximation scheme*, abbreviated FPAS.

In a very technical sense, an FPAS is the best one can hope for for an **NP**-hard optimization problem, assuming $\mathbf{P} \neq \mathbf{NP}$; see the discussion at the end of this chapter. The knapsack problem admits an FPAS.

Problem 8.3 (Knapsack) Given a set $S = \{a_1, \ldots, a_n\}$ of objects, with specified sizes and profits, $\mathrm{size}(a_i) \in \mathbf{Z}^+$ and $\mathrm{profit}(a_i) \in \mathbf{Z}^+$, and a "knapsack capacity" $B \in \mathbf{Z}^+$, find a subset of objects whose total size is bounded by B and total profit is maximized.

An obvious algorithm for this problem is to sort the objects by decreasing ratio of profit to size, and then greedily pick objects in this order. It is easy to see that as such this algorithm can be made to perform arbitrarily badly. However, a small modification yields a factor 2 algorithm: Let the sorted order of objects be $a_1, \ldots, a_n$. Find the least k such that the size of the first k objects exceeds B. Now, pick the more profitable of $\{a_1, \ldots, a_{k-1}\}$ and $\{a_k\}$ (we have assumed that the size of each object is at most B).

Exercise 8.4 Show that this algorithm achieves factor 2.

A pseudo-polynomial time algorithm for knapsack

Before presenting an FPAS for knapsack, we need one more concept. For any optimization problem Π, an instance consists of *objects*, such as sets or graphs, and *numbers*, such as cost, profit, size etc. So far, we have assumed that all numbers occurring in a problem instance I are written in binary. The *size* of the instance, denoted $|I|$, was defined as the number of bits needed to write I under this assumption. Let us say that I_u will denote instance I with all numbers occurring in it written in unary. The *unary size* of instance I, denoted $|I_u|$, is defined as the number of bits needed to write I_u.

An algorithm for problem Π is said to be efficient if its running time on instance I is bounded by a polynomial in $|I|$. Let us consider the following weaker definition:

Definition 8.5 An algorithm for problem Π whose running time on instance I is bounded by a polynomial in $|I_u|$ will be called a *pseudo-polynomial time algorithm.*

The knapsack problem, being **NP**-hard, does not admit a polynomial time algorithm; however, it does admit a pseudo-polynomial time algorithm. This fact is used critically in obtaining an FPAS for it. All known pseudo-polynomial time algorithms for **NP**-hard problems are based on dynamic programming.

Let P be the profit of the most profitable object, i.e. $P = \max_{a \in S} \text{profit}(a)$. Then nP is a trivial upperbound on the profit that can be achieved by any solution. For each $i \in \{1, \ldots, n\}$ and $p \in \{1, \ldots, nP\}$, let $S_{i,p}$ denote a subset of $\{a_1, \ldots, a_i\}$ whose total profit is exactly p, and total size is minimized. Let $A(i,p)$ denote the size of the set $S_{i,p}$ ($A(i,p) = \infty$ if no such set exists). Clearly $A(1,p)$ is known for every $p \in \{1, \ldots, nP\}$. The following recurrence helps compute all values $A(i,p)$ in $O(n^2P)$ time:

$$A(i+1,p) = \min\left\{A(i,p),\ \text{size}(a_{i+1}) + A(i, p - \text{profit}(a_{i+1}))\right\}$$

if $\text{profit}(a_{i+1}) < p$, and $A(i+1,p) = A(i,p)$ otherwise.

The maximum profit achievable by objects of total size bounded by B is $\max\{p|\ A(n,p) \leq B\}$. We thus get a pseudo-polynomial algorithm for knapsack.

An FPAS for knapsack

Notice that if the profits of objects were small numbers, i.e., they were bounded by a polynomial in n, then this would be a regular polynomial time algorithm, since its running time would be bounded by a polynomial in $|I|$. The key idea behind obtaining an FPAS is to exploit precisely this fact: We will ignore a certain number of least significant bits of profits of objects (depending on the error parameter ϵ), so that the modified profits can be viewed as numbers bounded by a polynomial in n and $\frac{1}{\epsilon}$. This will enable us to find a solution whose profit is at least $(1-\epsilon) \cdot \text{OPT}$ in time bounded by a polynomial in n and $\frac{1}{\epsilon}$.

Algorithm 8.6 (FPAS for knapsack)

1. Given $\epsilon > 0$, let $K = \frac{\epsilon P}{n}$.
2. For each object a_i, define $\text{profit}'(a_i) = \left\lfloor \frac{\text{profit}(a_i)}{K} \right\rfloor$.
3. With these as profits of objects, using the dynamic programming algorithm, find the most profitable set, say S'.
4. Output either S' or the most profitable object of size at most B, whichever gives higher profit.

Lemma 8.7 *Let A denote the set output by the algorithm. Then,*

$$\text{profit}(A) \geq (1-\epsilon) \cdot \text{OPT}.$$

Proof : Let O denote the optimal set. For any object a, because of rounding down, $K \cdot \text{profit}'(a)$ can be smaller than $\text{profit}(a)$, but by not more than K. Therefore,

$$\text{profit}(O) - K \cdot \text{profit}'(O) \leq nK.$$

The dynamic programming step must return a set at least as good as O under the new profits. Therefore,

$$\text{profit}(S') \geq\geq \text{profit}'(O) \geq \text{profit}(O) - nK = \text{OPT} - \epsilon P.$$

Since the algorithm also considers the most profitable element, $\text{profit}(A) \geq P$. Therefore,

$$\text{profit}(A) \geq \text{profit}(S') \geq \text{OPT} - \epsilon \cdot \text{profit}(A).$$

Hence,

$$\text{profit}(A) \geq \frac{1}{1+\epsilon}\text{OPT}.$$

The lemma follows. □

Theorem 8.8 *Algorithm 8.6 is a fully polynomial approximation scheme for knapsack.*

Proof : By Lemma 8.7, the solution found is within $(1-\epsilon)$ factor of OPT. Since the running time of the algorithm is $O\left(n^2 \left\lfloor \frac{P}{K} \right\rfloor\right) = O\left(n^2 \left\lfloor \frac{n}{\epsilon} \right\rfloor\right)$, which is polynomial in n and $1/\epsilon$, the theorem follows. □

A preliminary hardness result

In this section, we will prove in a formal sense that very few of the known **NP**-hard problems admit an FPAS. First, here is a strengthening of the notion of **NP**-hardness in a similar sense in which a pseudo-polynomial algorithm is a weakening of the notion of an efficient algorithm:

Definition 8.9 A problem Π is *strongly* **NP**-*hard* if every problem in **NP** can be polynomially reduced to Π in such a way that numbers in the reduced instance are always written in unary.

The restriction automatically forces the transducer to use polynomially bounded numbers only. Most known **NP**-hard problems are in fact strongly **NP**-hard; this includes all the problems in previous chapters for which approximation algorithms were obtained. A strongly **NP**-hard problem cannot have a a psuedo-polynomial time algorithm, assuming $\mathbf{P} \neq \mathbf{NP}$. Thus, knapsack is not strongly **NP**-hard, assuming $\mathbf{P} \neq \mathbf{NP}$.

We will show below that under some very weak restrictions, any **NP**-hard problem admitting an FPAS must admit a pseudo-polynomial time algorithm. Theorem 8.10 is proven for a minimization problem; a similar proof holds for a maximization problem.

Theorem 8.10 *Let p be a polynomial and Π be an* **NP**-*hard minimization problem such that the objective function f_Π is integer valued and on any instance I, $\mathrm{OPT}(I) < p(|I_u|)$. If Π admits an FPAS, then it also admits a pseudo-polynomial time algorithm.*

Proof : Suppose there is an FPAS for Π whose running time on instance I and error parameter ϵ is $q(|I|, \frac{1}{\epsilon})$, where q is a polynomial.

On instance I, set the error parameter to $\epsilon = \frac{1}{p(|I_u|)}$, and run the FPAS. Now, the solution produced will have objective function value

$$\leq (1+\epsilon)\mathrm{OPT}(I) < \mathrm{OPT}(I) + \epsilon p(|I_u|) = \mathrm{OPT}(I) + 1.$$

So, in fact with this error parameter, the FPAS will be forced to produce an optimal solution. The running time will be $q(|I|, p(|I_u|))$, i.e., polynomial in $|I_u|$. Therefore, we have obtained a pseudo-polynomial time algorithm for Π. □

The following corollary applies to most known **NP**-hard problems.

Corollary 8.11 *Let Π be an* **NP**-*hard optimization problem satisfying the restrictions of Theorem 8.10. If Π is strongly* **NP**-*hard, then Π does not admit an FPAS, assuming $\mathbf{P} \neq \mathbf{NP}$.*

Proof : If Π admits an FPAS, then it admits a pseudo-polynomial time algorithm by Theorem 8.10. But then it is not strongly **NP**-hard, assuming $\mathbf{P} \neq \mathbf{NP}$, leading to a contradiction. □

The stronger assumption that $\mathrm{OPT} < p(|I|)$ in Theorem 8.10 would have enabled us to prove that there is a polynomial time algorithm for Π. However, this stronger assumption is less widely applicable. For instance, it is not satisfied by the minimum makespan problem, which we will study in Chapter 9.

The design of all known FPAS's and PTAS's is based on the idea of trading accuracy for running time – the given problem instance is mapped to a coarser instance, depending on the error parameter ϵ, which is solved optimally by a dynamic programming approach. The latter ends up being an exhaustive search of polynomially many different possibilities (for instance, for knapsack, this involves computing $A(i,p)$ for all i and p), and in most such algorithms, the running time is prohibitive even for reasonable n and ϵ. Further, if the algorithm had to resort to exhaustive search, does the problem really offer "footholds" to home in on a solution efficiently? So, is an FPAS the best one can hope for for an **NP**-hard problem? Clearly, the issue is complex and there is no straightforward answer.

Chapter 9

Minimum makespan scheduling

A central problem in scheduling theory is the following:

Problem 9.1 (Minimum makespan scheduling) Given processing times for n jobs, $p_1, p_2, \ldots, p_n$, and an integer m, find an assignment of the jobs to m identical machines so that the completion time, also called the *makespan*, is minimized.

Interestingly enough a factor 2 algorithm for this problem, regarded as the first approximation algorithm designed with a proven guarantee, was obtained even before the theory of NP-completeness was discovered. We begin by presenting this algorithm because of its historic importance, before giving a polynomial approximation scheme for the problem.

Factor 2 algorithm

The algorithm is very simple: arbitrarily order the jobs; schedule the next job on the machine that has been assigned the least amount of work so far.

This algorithm is based on the following two lower bounds on OPT, the optimal makespan:

1. the average time for which a machine has to run, $\left(\sum_i p_i\right)/m$, and
2. the running time of any one job.

For future reference, let LB denote the combined lower bound, i.e.,

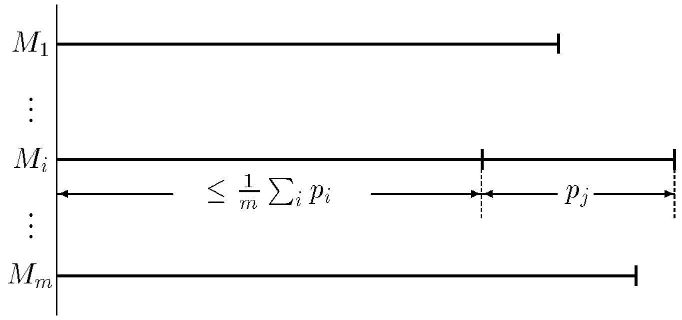
$$\mathrm{LB} = \max\left\{\frac{1}{m}\sum_i p_i,\ \max\{p_i\}\right\}.$$

Algorithm 9.2 (Minimum makespan scheduling)

1. Order the jobs arbitrarily.
2. Schedule jobs on machines in this order, scheduling the next job on the machine that has been assigned the least amount of work so far.

Theorem 9.3 *Algorithm 9.2 achieves an approximation guarantee of 2 for the minimum makespan problem.*

Proof : Let M_i be the machine that completes last in the schedule produced by the algorithm, and let j be the index of the last job scheduled on this machine.



Let $start_j$ be the time at which job j starts execution on M_i. Since the algorithm assigns a job to the least loaded machine, it follows that all machines are busy until $start_j$. This implies that

$$start_j \leq \frac{1}{m}\sum_i p_i \leq \text{OPT}.$$

Further, $p_j \leq \text{OPT}$. So, the makespan of the schedule is $start_j + p_j \leq 2 \cdot \text{OPT}$. □

Example 9.4 A tight example for this algorithm is provided by a sequence of m^2 jobs with unit processing time, followed by a single job of length m. The schedule obtained by the algorithm has a makespan of $2m$, while $\text{OPT} = m + 1$. □

A PTAS for the minimum makespan problem

By Corollary 8.11, the minimum makespan problem does not admit an FPAS, assuming $\mathbf{P} \neq \mathbf{NP}$; instead we will obtain a PTAS for it. The minimum makespan problem is closely related to the bin packing problem by the following observation: there exists a schedule with makespan t if and only if n objects of sizes $p_1, p_2, \ldots, p_n$ can be packed into m bins of capacity t each. This suggests a reduction from minimum makespan to bin packing as follows: Denoting the sizes of the n objects, $p_1, \ldots, p_n$, by I, let $\text{bins}(I, t)$ represent the minimum number of bins of size t required to pack these n objects. Then, the minimum makespan is given by

$$\min\{t : \text{bins}(I, t) \leq m\}.$$

As shown above, LB and $2 \cdot \text{LB}$ are lower and upper bounds on the minimum makespan. So, we can determine the minimum makespan by a binary search in this interval. At first sight, this reduction may not seem very useful since the bin packing problem is also **NP**-hard. However, it turns out that this problem is polynomial time solvable if the object sizes are drawn from a set of fixed cardinality; we will use this fact critically in solving the minimum makespan problem.

Bin packing with fixed number of object sizes

We first present a dynamic programming algorithm for the restricted bin packing problem. Let k be the fixed number of object sizes, and assume that bins have capacity 1. Fix an ordering on the object sizes. Now, an instance of the bin packing problem can be described by a k-tuple,

$(i_1, i_2, \ldots, i_k)$, specifying the number of objects of each size. Let $\mathrm{BINS}(i_1, i_2, \ldots, i_k)$ denote the minimum number of bins needed to pack these objects.

For a given instance, $(n_1, n_2, \ldots, n_k), \sum_{i=1}^{n} n_i = n$, we first compute $\mathcal{Q}$, the set of all k-tuples $(q_1, q_2, \ldots, q_k)$ such that $\mathrm{BINS}(q_1, q_2, \ldots, q_k) = 1$ and $0 \le q_i \le n_i, 1 \le i \le k$. Clearly, $\mathcal{Q}$ contains at most $O(n^k)$ elements. Next, we compute all entires of the k-dimensional table $\mathrm{BINS}(i_1, i_2, \ldots, i_k)$ for every $(i_1, i_2, \ldots, i_k) \in \{0, \ldots, n_1\} \times \{0, \ldots, n_2\} \times \ldots \times \{0, \ldots, n_k\}$. The table is initialized by setting $\mathrm{BINS}(q) = 1$ for every $q \in \mathcal{Q}$. Then, we use the following recurrence to compute the remaining entries:

$$\mathrm{BINS}(i_1, i_2, \ldots, i_k) = 1 + \min_{q \in \mathcal{Q}} \mathrm{BINS}(i_1 - q_1, \ldots, i_k - q_k). \tag{9.1}$$

Clearly, computing each entry takes $O(n^k)$ time. So, the entire table can be computed in $O(n^{2k})$ time, thereby determining $\mathrm{BINS}(n_1, n_2, \ldots, n_k)$.

Reducing makespan to restricted bin packing

The basic idea is that if we can tolerate some error in computing the minimum makespan, then we can reduce this problem to the restricted version of bin packing in polynomial time. There will be two sources of error:

- rounding object sizes so that there are a bounded number of different sizes, and
- terminating the binary search to ensure polynomial running time.

Each error can be made as small as needed, at the expense of running time. Moreover, for any fixed error bound, the running time is polynomial in n, and so we obtain a polynomial approximation scheme.

Let ϵ be an error parameter, and t be in the interval $[\mathrm{LB}, 2 \cdot \mathrm{LB}]$. We say that an object is *small* if its size is less than $t\epsilon$; small objects are discarded for now. The rest of the objects are rounded down as follows: each p_i in the interval $[t\epsilon(1+\epsilon)^i,\ t\epsilon(1+\epsilon)^{i+1})$ is replaced by $p_i' = t\epsilon(1+\epsilon)^i$, for $i \ge 0$. So, the resulting p_i''s can assume at most $k = \lceil \log_{1+\epsilon} \frac{1}{\epsilon} \rceil$ distinct values. Determine an optimal packing for the rounded objects in bins of size t using the dynamic programming algorithm. Since rounding reduces the size of each object by a factor of at most $1+\epsilon$, if we consider the original sizes of the objects, then the packing determined is valid for a bin size of $t(1+\epsilon)$. Keeping this as the bin size, pack the small objects greedily in leftover spaces in the bins; open new bins only if needed. Clearly, any time a new bin is opened, all previous bins must be full to the extent of at least t. Denote by $\alpha(I, t, \epsilon)$ the number of bins used by this algorithm; recall that these bins are of size $t(1+\epsilon)$.

Let us call the algorithm presented above the *core algorithm* since it will form the core of the PTAS for computing makespan. As shown in Lemma 9.5 and its Corollary, the core algorithm also helps establish a lower bound on the optimal makespan.

Lemma 9.5 $\alpha(I, t, \epsilon) \le \mathrm{bins}(I, t)$.

Proof : If the algorithm does not open any new bins for the small objects, then the assertion clearly holds since the rounded down pieces have been packed optimally in bins of size t. In the other case, all but the last bin are packed to the extent of t at least. Hence, the optimal packing of I in bins of size t must also use at least $\alpha(I, t, \epsilon)$ bins. □

Since $\mathrm{OPT} = \min\{t : \mathrm{bins}(I, t) \leq m\}$, Lemma 9.5 gives:

Corollary 9.6 $\min\{t : \alpha(I, t, \epsilon) \leq m\} \leq \mathrm{OPT}$.

If $\min\{t : \alpha(I, t, \epsilon) \leq m\}$ could be determined with no additional error during the binary search, then clearly we could use the core algorithm to obtain a schedule with a makespan of $(1+\epsilon)\mathrm{OPT}$. Next, we will specify the details of the binary search and show how to control the error it introduces. The binary search is performed on the interval $[\mathrm{LB}, 2 \cdot \mathrm{LB}]$. So, the length of the available interval is LB at the start of the search, and it reduces by a factor of 2 in each iteration. We continue the search until the avialable interval drops to a length of $\epsilon \cdot \mathrm{LB}$. This will require $\lceil \log_2 \frac{1}{\epsilon} \rceil$ iterations. Let T be the right endpoint of the interval we terminate with.

Lemma 9.7 $T \leq (1+\epsilon) \cdot \mathrm{OPT}$.

Proof : Clearly, $\min\{t : \alpha(I, t, \epsilon) \leq m\}$ must be in the interval $[T - \epsilon, T]$. Hence,

$$T \leq \min\{t : \alpha(I, t, \epsilon) \leq m\} + \epsilon \cdot \mathrm{LB}.$$

Now, using Corollary 9.6 and the fact that $\mathrm{LB} \leq \mathrm{OPT}$, the lemma follows. □

Finally, the output of the core algorithm for $t = T$ gives a schedule whose makespan is at most $T \cdot (1+\epsilon)$. So, we get:

Theorem 9.8 *The algorithm produces a valid schedule having makespan at most*

$$(1+\epsilon)^2 \cdot \mathrm{OPT} \leq (1+3\epsilon) \cdot \mathrm{OPT}.$$

The running time of the entire algorithm is $O\left(n^{2k} \lceil \log_2 \frac{1}{\epsilon} \rceil\right)$, where $k = \lceil \log_{1+\epsilon} \frac{1}{\epsilon} \rceil$.

Part II

LP BASED ALGORITHMS

Chapter 10

Introduction to LP-duality

As stated in Chapter , LP-duality theory unifies a large part of the theory of approximation algorithms as we know it today. For a comprehensive introduction to linear programming we recommend [?]. In this chapter, we will review some concepts that we will use critically.

Linear programming is the problem of optimizing (i.e., minimizing or maximizing) a linear function subject to linear inequality constraints; the function being optimized is called the *objective function*. Perhaps the most interesting fact about this problem from our perspective is that it is well-characterized in the sense of Chapter 10. Let us illustrate this through a simple example:

$$
\begin{array}{ll}
\text{minimize} & 7x_1 + x_2 + 5x_3 \\
\text{subject to} & \begin{array}{rcrcrcl} x_1 & - & x_2 & + & 3x_3 & \geq & 10 \\ 5x_1 & + & 2x_2 & - & x_3 & \geq & 6 \end{array} \\
& x_1, x_2, x_3 \geq 0
\end{array}
$$

Notice that in this example all constraints are of the kind "$\geq$" and all variables are constrained to be non-negative. This is the standard form of a minimization linear program; a simple transformation enables one to write any minimization linear program in this manner. The reason for choosing this form will become clear shortly.

Any solution, i.e., a setting for the variables in this linear program, that satisfies all the constraints is said to be a *feasible solution*. Let $\boldsymbol{z}^*$ denote the optimum value of this linear program. Let us consider the question, "Is $\boldsymbol{z}^*$ at most α?" where α is a given rational number. For instance, let us ask whether $\boldsymbol{z}^* \leq 30$. A Yes certificate for this question is simply a feasible solution whose objective function value is at most 30. For example, $\boldsymbol{x} = (2, 1, 3)$ constitutes such a certificate since it satisfies the two constraints of the problem, and the objective function value for this solution is $7 \cdot 2 + 1 + 5 \cdot 3 = 30$. Thus, any Yes certificate to this question provides an upper bound on $\boldsymbol{z}^*$.

How do we provide a No certificate for such a question? In other words, how do we place a good lower bound on $\boldsymbol{z}^*$? In our example, one such bound is given by the first constraint: since the x_i's are restricted to be non-negative, term by term comparison of coefficients shows that $7x_1 + x_2 + 5x_3 \geq x_1 - x_2 + 3x_3$. Since the right hand side of the first constraint is 10, we get that the objective function is at least 10 for any feasible solution. A better lower bound can be obtained by taking the sum of the two constraints: for any feasible solution $\boldsymbol{x}$,

$$7x_1 + x_2 + 5x_3 \geq (x_1 - x_2 + 3x_3) + (5x_1 + 2x_2 - x_3) \geq 16.$$

The idea behind this process of placing a lower bound is that we are finding suitable non-negative multipliers for the constraints so that when we take their sum, the coefficient of each x_i

in the sum is dominated by the coefficient in the objective function. Now, the right hand side of this sum is a lower bound on z^* since any feasible solution has a non-negative setting for each x_i. Notice the importance of ensuring that the multipliers are non-negative: they do not reverse the direction of the constraint inequality.

Clearly, the rest of the game lies in choosing the multipliers in such a way that the right hand side of the sum is as large as possible. Interestingly enough, the problem of finding the best such lower bound can be formulated as a linear program:

$$\begin{array}{ll} \text{maximize} & 10y_1 + 6y_2 \\ \text{subject to} & \begin{array}{rcrcl} y_1 & + & 5y_2 & \leq & 7 \\ -y_1 & + & 2y_2 & \leq & 1 \\ 3y_1 & - & y_2 & \leq & 5 \end{array} \\ & y_1, y_2 \geq 0 \end{array}$$

Here y_1 and y_2 were chosen to be the non-negative multipliers for the first and the second constraint respectively. Let us call the first linear program the *primal program* and the second the *dual program*. There is a systematic way of obtaining the dual of any linear program; one is a minimization problem and the other is a maximization problem. Further, the dual of the dual is the primal program itself (see [?]). By construction, every feasible solution to the dual program gives a lower bound on the optimum value of the primal. Observe that the reverse also holds: every feasible solution to the primal program gives an upper bound on the optimal value of the dual. Therefore, if we can find feasible solutions for the dual and the primal with matching objective function values, then both solutions must be optimal. In our example, $\boldsymbol{x} = (7/4, 0, 11/4)$ and $\boldsymbol{y} = (2, 1)$ both achieve objective function values of 26, and so both are optimal solutions. The reader may wonder whether our example was ingeniously chosen for this to happen. Surprisingly enough, this is not an exception but the rule! This is the central theorem of linear programming: *the LP-Duality Theorem.*

In order to state this theorem formally, let us consider the following minimization problem, written in standard form, as the primal program; equivalently, we could have started with a maximization problem as the primal program.

$$\begin{array}{lll} \text{minimize} & \sum_{j=1}^{n} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \geq b_i, & i = 1, \ldots, m \\ & x_j \geq 0, & j = 1, \ldots, n \end{array} \tag{10.1}$$

where a_{ij}, b_i, and c_j are given rational numbers.
Then the dual program is:

$$\begin{array}{lll} \text{maximize} & \sum_{i=1}^{m} b_i y_i & \\ \text{subject to} & \sum_{i=1}^{m} a_{ij} y_i \leq c_j, & j = 1, \ldots, n \\ & y_i \geq 0, & i = 1, \ldots, m \end{array} \tag{10.2}$$

Theorem 10.1 (LP-Duality Theorem) *The primal program has finite optimum if and only if its dual has finite optimum. Moreover, if $\boldsymbol{x}^* = (x_1^*, \ldots, x_n^*)$ and $\boldsymbol{y}^* = (y_1^*, \ldots, y_m^*)$ are optimal solutions for the primal and the dual program respectively, then*

$$\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^*.$$

Notice that the LP-Duality Theorem is really a min-max relation, since one program is a minimization problem and the other is a maximization problem. This theorem shows that the linear programming problem is well-characterized: feasible solutions to the primal (dual) provide Yes (No) certificates to the question, "Is the optimum value less than or equal to α?" Thus, as a corollary of this theorem we get that linear programming is in $\mathbf{NP} \cap \text{co-}\mathbf{NP}$.

Going back to our example, by construction, any feasible solution to the dual program gives a lower bound on the optimal value of the primal. So, in fact it also gives a lower bound on the objective function value achieved by any feasible solution to the primal. This is the easy half of the LP-Duality Theorem, sometimes called the *Weak Duality Theorem*. We give a formal proof of this since some steps in the proof will lead to the next important fact. For a proof of the LP-Duality Theorem see [**?**]. The design of several exact algorithms have their basis in the LP-Duality Theorem. In contrast, in approximation algorithms, typically the Weak Duality Theorem suffices.

Theorem 10.2 (Weak Duality Theorem) *If $\boldsymbol{x} = (x_1, \ldots, x_n)$ is a feasible solution for the primal program and $\boldsymbol{y} = (y_1, \ldots, y_m)$ is a feasible solution for the dual, then*

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{i=1}^{m} b_i y_i. \tag{10.3}$$

Proof : Since $\boldsymbol{y}$ is dual feasible and x_j's are non-negative,

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j. \tag{10.4}$$

Similarly, since $\boldsymbol{x}$ is primal feasible and y_i's are non-negative,

$$\sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i \geq \sum_{i=1}^{m} b_i y_i. \tag{10.5}$$

The theorem follows by observing that

$$\sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i.$$

□

By the LP-Duality Theorem, $\boldsymbol{x}$ and $\boldsymbol{y}$ are both optimal solutions if and only if (10.3) holds with equality. Clearly, this happens if and only if both (10.4) and (10.5) hold with equality. Hence, we get the following result about the structure of optimal solutions:

Theorem 10.3 (Complementary Slackness Conditions) *Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be primal and dual feasible solutions respectively. Then, $\boldsymbol{x}$ and $\boldsymbol{y}$ are both optimal if and only if all of the following conditions are satisfied:*

Primal complementary slackness conditions
For each $1 \le j \le n$: either $x_j = 0$ or $\sum_{i=1}^{m} a_{ij} y_i = c_j$; and

Dual complementary slackness conditions
For each $1 \le i \le m$: either $y_i = 0$ or $\sum_{j=1}^{n} a_{ij} x_j = b_i$.

The complementary slackness conditions play a vital role in the design of efficient algorithms, both exact and approximation; see Chapter 13 for details. For a better appreciation of their importance, we recommend that the reader study Edmonds' weighted matching algorithm (see, for example, [**?**]).

Min-max relations and the LP-Duality Theorem

In order to appreciate the role of LP-duality theory in approximation algorithms, it is important to first understand its role in exact algorithms. To do so, we will review some of these ideas in the context of the max-flow min-cut theorem; in particular, we will show how this and other min-max relations follow from the LP-Duality Theorem. Some of the ideas on cuts and flows developed here will also be used in the study of multicommodity flow in Chapters 14 17 and 18.

The problem of computing a maximum flow in a network is: Given a directed[1] graph, $G = (V, E)$ with two distinguished nodes, *source* s and *sink* t, and positive arc capacities, $c : E \rightarrow \mathbf{R}^+$, find the maximum amount of flow that can be sent from s to t subject to

1. *capacity constraint:* for each arc e, the flow sent through e is bounded by its capacity, and
2. *flow conservation:* at each node v, other than s and t, the total flow into v should equal the total flow out of v.

An *s-t cut* is defined by a partition of the nodes into two sets X and $\overline{X}$ so that $s \in X$ and $t \in \overline{X}$, and consists of the set of arcs going from X to $\overline{X}$. The *capacity* of this cut, $c(X, \overline{X})$, is defined to be the sum of capacities of these arcs. Because of the capacity constraints on flow, the capacity of any s-t cut is an upper bound on any feasible flow. So, if the capacity of an s-t cut, say $(X, \overline{X})$ equals the value of a feasible flow, then $(X, \overline{X})$ must be a minimum s-t cut and the flow must be a maximum flow in G. The max-flow min-cut theorem proves that it is always possible to find a flow and an s-t cut so that equality holds.

Let us formulate the maximum flow problem as a linear program. First, introduce a fictitious arc of infinite capacity from t to s, thus converting the flow to a circulation; the objective now is to maximize the flow on this arc, denoted by f_{ts}. The advantage of making this modification is that we can now require flow conservation at s and t as well. If f_{ij} denotes the amount of flow sent through arc $(i, j) \in E$, we can formulate the maximum flow problem as follows:

$$\begin{array}{lll} \text{maximize} & f_{ts} & \\ \text{subject to} & f_{ij} \le c_{ij}, & (i,j) \in E \end{array}$$

[1]The maximum flow problem in undirected graphs reduces to that in directed graphs: replace each edge (u, v) by two directed edges, $(u \rightarrow v)$ and $(v \rightarrow u)$, each of the same capacity as (u, v).

$$\sum_{j:\ (j,i)\in E} f_{ji} - \sum_{j:\ (i,j)\in E} f_{ij} \le 0, \qquad i \in V$$
$$f_{ij} \ge 0, \qquad (i,j) \in E$$

The second set of inequalities say that for each node i, the total flow into i is at most the total flow out of i. Notice that if this inequality holds at each node, then in fact it must be satisfied with equality at each node, thereby implying flow conservation at each node (this is so because a deficit in flow balance at one node implies a surplus at some other node). With this trick, we get a linear program in standard form.

To obtain the dual program we introduce variables d_{ij} and p_i corresponding to the two types of inequalities in the primal. We will view these variables as distance labels on arcs and potentials on nodes respectively. The dual program is:

$$\begin{array}{lll}
\text{minimize} & \displaystyle\sum_{(i,j)\in E} c_{ij} d_{ij} & \\
\text{subject to} & d_{ij} - p_i + p_j \ge 0, & (i,j) \in E \\
& p_s - p_t \ge 1 & \\
& d_{ij} \ge 0, & (i,j) \in E \\
& p_i \ge 0, & i \in V
\end{array}$$

For developing an intuitive understanding of the dual program, it will be best to first transform it into an integer program that seeks 0/1 solutions to the the variables:

$$\begin{array}{lll}
\text{minimize} & \displaystyle\sum_{(i,j)\in E} c_{ij} d_{ij} & \\
\text{subject to} & d_{ij} - p_i + p_j \ge 0, & (i,j) \in E \\
& p_s - p_t \ge 1 & \\
& d_{ij} \in \{0,1\}, & (i,j) \in E \\
& p_i \in \{0,1\}, & i \in V
\end{array}$$

Let $(\boldsymbol{d}^*, \boldsymbol{p}^*)$ be an optimal solution to this integer program. The only way to satisfy the inequality $p_s^* - p_t^* \ge 1$ with a 0/1 substitution is to set $p_s^* = 1$ and $p_t^* = 0$. So, this solution naturally defines an s-t cut $(X, \overline{X})$, where X is the set of potential 1 nodes, and $\overline{X}$ the set of potential 0 nodes. Consider an arc (i,j) with $i \in X$ and $j \in \overline{X}$. Since $p_i^* = 1$ and $p_j^* = 0$, by the first constraint, $d_{ij}^* \ge 1$. But since we have a 0/1 solution, $d_{ij}^* = 1$. The distance label for each of the remaining arcs can be set to either 0 or 1 without violating the first constraint; however, in order to minimize the objective function value, it must be set to 0. So, the objective function value is precisely the capacity of the cut $(X, \overline{X})$, and hence $(X, \overline{X})$ must be a minimum s-t cut.

Thus, this integer program is a formulation of the minimum s-t cut problem! What about the dual program? The dual program can be viewed as a relaxation of the integer program where the integrality constraint on the variables is dropped. This leads to the constraints $1 \ge d_{ij} \ge 0$ for $(i,j) \in E$ and $1 \ge p_i \ge 0$ for $i \in V$. Next, we notice that the upper bound constraints on the variables are redundant; their omission cannot give a better solution. Dropping these constraints gives the dual program in the form given above. We will say that this program is the *LP relaxation* of the integer program.

Consider an s-t cut C. Set C has the property that any path from s to t in G contains at least one edge of C. Using this observation, we can interpret any feasible solution to the dual program as a *fractional s-t cut*: the distance labels it assigns to arcs satisfy the property that on any path from s to t the distance labels add up to at least 1. To see this, consider an s-t path $(s = v_0, v_1, \ldots, v_k = t)$. Now, the sum of the potential differences on the end points of arcs on this path,

$$\sum_{i=0}^{k-1} (p_i - p_{i+1}) = p_s - p_t.$$

So, by the first constraint, the sum of the distance labels on the arcs must add up to at least $p_s - p_t$, which is ≥ 1. Let us define the *capacity* of this fractional s-t cut to be the dual objective function value achieved by it.

In principle, the best fractional s-t cut could have lower capacity than the best integral cut. Surprisingly enough, this does not happen. Consider the polyhedron defining the set of feasible solutions to the dual program. Let us call a feasible solution a *vertex solution* if it is a vertex of this polyhedron. From linear programming theory we know that for any objective function, i.e., assignment of capacities to the arcs of G, there is a vertex solution that is optimal (for this discussion let us assume that for the given objective function, an optimal solution exists). Now, it can be proven that each vertex solution is integral, with each coordinate being 0 or 1. (This follows from the fact that the constraint matrix of this program is totally unimodular[2]; see [**?**] for a proof.) Thus, the dual program always has an integral optimal solution.

By the LP-Duality Theorem maximum flow in G must equal capacity of a minimum fractional s-t cut. But since the latter equals the capacity of a minimum s-t cut, we get the max-flow min-cut theorem.

So, the max-flow min-cut theorem is a special case of the LP-duality theorem; it holds because the dual polyhedron has integral vertices. In fact, most min-max relations in combinatorial optimization hold for analogous reasons.

Finally, let us illustrate the usefulness of complementary slackness conditions by utilizing them to derive additional properties of optimal solutions to the flow and cut programs. Let $\boldsymbol{f}^*$ be an optimum solution to the primal LP (i.e., a maximum s-t flow). Also, let $(\boldsymbol{d}^*, \boldsymbol{p}^*)$ be an integral optimum solution to the dual LP, and let $(X, \overline{X})$ be the cut defined by $(\boldsymbol{d}^*, \boldsymbol{p}^*)$. Consider an arc (i,j) such that $i \in X$ and $j \in \overline{X}$. We have proven above that $d^*_{ij} = 1$. Since $d^*_{ij} \neq 0$, by the dual complementary slackness condition, $f^*_{ij} = c_{ij}$. Next, consider an arc (k,l) such that $k \in \overline{X}$ and $l \in X$. Since $p^*_k - p^*_l = -1$, and $d^*_{kl} \in \{0,1\}$, the constraint $d^*_{kl} - p^*_k + p^*_l \geq 0$ must be satisfied as a strict inequality. So, by the primal complementary slackness condition, $f^*_{kl} = 0$. Thus we have proven that arcs going from X to $\overline{X}$ are saturated by $\boldsymbol{f}^*$, and the reverse arcs carry no flow. (Observe that it was not essential to invoke complementary slackness conditions to prove these facts; they also follow from the fact that flow across cut $(X, \overline{X})$ equals its capacity.)

Exercise 10.4 Show that the dual of the dual of a linear program is the program itself.

Exercise 10.5 Show that any minimization program can be transformed into an equivalent program in standard form, i.e., the form of LP (10.1).

Exercise 10.6 Change some of the constraints of the primal program (10.1) into equalities, i.e., so they are of the form

$$\sum_{j=1}^{n} a_{ij} x_j = b_i.$$

[2]A matrix A is said to be *totally unimodular* if the determinant of every square submatrix of A is $1, -1$ or 0.

Show that the dual of this program involves modifying program (10.2) so that the corresponding dual variables y_i are *unconstrained*, i.e., they are not constrained to be non-negative. Additionally, if some of the variables x_j in program (10.1) are unconstrained, then the corresponding constraints in the dual become equalities.

Exercise 10.7 In this exercise, you will derive von Neumann's Minimax Theorem in game theory from the LP-Duality Theorem. A finite two-person zero-sum game is specified by an $m \times n$ matrix $\boldsymbol{A}$ with real entries. In each round, the row player, R, selects a row, say i; simultaneously, the column player, C, selects a column, say j. The *payoff* to R at the end of this round is a_{ij}. Thus, $|a_{ij}|$ is the amount that C pays R (R pays C) if a_{ij} is positive (a_{ij} is negative); no money is exchanged if a_{ij} is zero. *Zero sum game* refers to the fact that the total amount of money possessed by R and C together is conserved.

The *strategy* of each player is specified by vector whose entries are non-negative and add up to one, giving the probabilities with which the player picks each row or column. Let R's strategy be given by m-dimensional vector $\boldsymbol{x}$, and C's strategy be given by n-dimensional vector $\boldsymbol{y}$. Then, the excected pay off to R in a round is $\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{y}$. The job of each player is to pick a strategy that *guarantees* maximum possible expected winnings (equivalently, minimum possible expected losses), regardless of the strategy chosen by the other player. If R chooses strategy $\boldsymbol{x}$, he can be sure of winning only $\min_{\boldsymbol{y}} \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{y}$, where the minimum is taken over all possible strategies of C. So, the optimal choice for R is given by $\max_{\boldsymbol{x}} \min_{\boldsymbol{y}} \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{y}$. Similarly, C will minimize her losses by choosing the strategy given by $\min_{\boldsymbol{y}} \max_{\boldsymbol{x}} \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{y}$. The Minimax Theorem states that for every matrix $\boldsymbol{A}$, $\max_{\boldsymbol{x}} \min_{\boldsymbol{y}} \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{y} = \min_{\boldsymbol{y}} \max_{\boldsymbol{x}} \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{y}$.

Let us say that a strategy is *pure* if it picks a single row or column, i.e., the vector corresponding to it consists of one 1 and the rest 0's. A key observation is that for any strategy $\boldsymbol{x}$ of R, $\min_{\boldsymbol{y}} \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{y}$ is attained for a pure strategy of C: Suppose the minimum is attained for strategy $\boldsymbol{y}$. Consider the pure strategy corresponding to any non-zero component of $\boldsymbol{y}$. The fact that the components of $\boldsymbol{y}$ are non-negative and add up to one leads to an easy proof that this pure strategy attains the same minimum. So, R's optimum strategy is given by $\max_{\boldsymbol{x}} \min_j \sum_{i=1}^m a_{ij} x_i$. The second critical observation is that the problem of computing R's optimal strategy can be expressed as a linear program:

$$
\begin{array}{lll}
\text{maximize} & z & \\
\text{subject to} & z - \sum_{i=1}^{m} a_{ij} x_i \leq 0, & j = 1, \ldots, n \\
& \sum_{i=1}^{m} x_i = 1 & \\
& x_i \geq 0, & i = 1, \ldots, m
\end{array}
$$

Find the dual of this LP, and show that it computes the optimal strategy for C. (Use the fact that for any strategy $\boldsymbol{y}$ of C, $\max_{\boldsymbol{x}} \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{y}$ is attained for a pure strategy of R.) Hence, prove the Minimax Theorem using the LP-Duality Theorem.

Chapter 11

Rounding applied to set cover

As stated in the Introduction, a key step in designing an approximation algorithm for an **NP**-hard problem is establishing a good lower bound on the cost of the optimal solution (for this discussion we will assume that we have a minimization problem at hand). This is where linear programming helps out: Many combinatorial optimization problems can be expressed as integer programming problems. For these problems, the cost of an optimal solution to the LP-relaxation provides the desired lower bound. As in the case of the minimum s-t cut problem (see Chapter 10), a feasible solution to the LP-relaxation represents a *fractional solution* to the original problem. However, in this case, we cannot expect the polyhedron defining the set of feasible solutions to have integer vertices, since the original problem is **NP**-hard.

An obvious strategy for obtaining a "good" solution to the original problem is to solve the linear program and then convert the solution obtained into an integral solution, trying to ensure that in the process the cost does not increase much. The approximation guarantee is established by comparing the cost of the integral and fractional solutions. This strategy is called *rounding*.

A second, less obvious and perhaps more sophisticated, strategy is to use the dual of the LP-relaxation in the design of the algorithm and in the proof of its approximation guarantee. Under this strategy lies the *primal dual schema*, a rather general algorithm design schema that yields the best known approximation algorithms for a diverse collection of problems. These ideas will be presented in Chapters 12 and 13.

In this chapter, we will present two different algorithms for the set cover problem based on rounding. Define the *frequency* of an element in a set cover instance to be the number of sets it is in. The approximation guarantee achieved by the first algorithm is the frequency of the most frequent element. Frequently, the rounding procedure uses randomization. This is illustrated in the second algorithm which achieves an approximation guarantee of $O(\log n)$. Notice that neither algorithm dominates the other on all instances.

A simple rounding algorithm

Recall the minimum set cover problem: Given a set U with n elements, a collection of subsets of U, $\mathcal{S}$, and a cost function $c : \mathcal{S} \to \mathbf{Q}^+$, find a minimum cost sub-collection of $\mathcal{S}$ that covers all elements of U. Let f denote the frequency of the most frequent element.

To formulate this problem as an integer program, let us assign a variable x_S for each set $S \in \mathcal{S}$, which is allowed 0/1 values. This variable will be set to 1 iff set S is picked in the set cover. Clearly, the constraint is that for each element $e \in U$ we want that at least one of the sets containing it be picked.

$$\begin{array}{lll} \text{minimize} & \sum_{S \in \mathcal{S}} c(S) x_S & \\ \text{subject to} & \sum_{S:\ e \in S} x_S \geq 1, & e \in U \\ & x_S \in \{0,1\}, & S \in \mathcal{S} \end{array} \tag{11.1}$$

The LP-relaxation of this integer program is obtained by letting the domain of variables x_S be $1 \geq x_S \geq 0$. Since the upper bound on x_S is redundant, we get:

$$\begin{array}{lll} \text{minimize} & \sum_{S \in \mathcal{S}} c(S) x_S & \\ \text{subject to} & \sum_{S:\ e \in S} x_S \geq 1, & e \in U \\ & x_S \geq 0, & S \in \mathcal{S} \end{array} \tag{11.2}$$

One way of converting a solution to this linear program into an integral solution is to round up all non-zero variables to 1. It is easy to construct examples showing that this could increase the cost by a factor of $\Omega(n)$ (see Example 11.3). However, this simple algorithm does achieve the desired approximation guarantee of f (see Exercise 11.4). Let us consider a slight modification of this algorithm that is easier to prove, and picks fewer sets in general:

Algorithm 11.1 (Set cover via rounding)

1. Find an optimal solution to the LP-relaxation.
2. Pick all sets S for which $x_S \geq \frac{1}{f}$ in this solution.

Theorem 11.2 *Algorithm 11.1 achieves an approximation factor of f for the set cover problem.*

Proof : Let $\mathcal{C}$ be the collection of picked sets. Consider an arbitrary element e. Since e is in at most f sets, one of these sets must be picked to the extent of at least $\frac{1}{f}$ in the fractional cover. So, e is covered by $\mathcal{C}$, and hence $\mathcal{C}$ is a valid set cover. For each set $S \in \mathcal{C}$, x_S has been increased by a factor of at most f. Therefore, the cost of $\mathcal{C}$ is at most f times the cost of the fractional cover, thereby proving the desired approximation guarantee. □

The weighted vertex cover problem can be seen as a set cover problem with $f = 2$: elements correspond to edges and sets correspond to vertices, with inclusion corresponding to incidence. The cost of a set is the weight of the corresponding vertex. Since each edge is incident to two vertices, each element is in two sets, and so $f = 2$. Using the algorithm given above, we get a factor 2 algorithm for weighted vertex cover, matching the approximation guarantee for the unweighted problem presented in Chapter 10; this is the best known approximation guarantee for the weighted as well as unweighted vertex cover problems.

Example 11.3 Let us give a tight example for the set cover algorithm. For simplicity, we will describe a hypergraph and use the transformation given above to obtain a set cover instance from it. Let $V_1, \ldots, V_k$ be disjoint sets of cardinality n each. The hypergraph has vertex set $V = V_1 \cup \ldots \cup V_k$, and n^k hyperedges; each hyperedge picks one vertex from each V_i. In the set cover instance, elements correspond to hyperedges and sets correspond to vertices. Once again, inclusion corresponds to incidence. Each set has cost 1. Picking each set to the extent of $\frac{1}{k}$ gives an optimal fractional cover of cost n. Given this fractional solution, the rounding algorithm will pick all nk sets. On the other hand, picking all sets corresponding to vertices in V_1 gives a set cover of cost n. $\square$

Exercise 11.4 Show, using the primal complementary slackness conditions, that picking all sets that are non-zero in the fractional solution also leads to a factor f algorithm.

A randomized rounding approach to set cover

Let $x_S = p_S, S \in \mathcal{S}$ be an optimal solution to the linear program. Viewing p_S's as probabilities, we will pick a sub-collection of $\mathcal{S}$ as follows: for each set $S \in \mathcal{S}$, pick S with probability p_S. Let $\mathcal{C}$ be the collection of sets picked. The expected cost of $\mathcal{C}$ is

$$\sum_{S \in \mathcal{S}} \mathbf{Pr}[S \text{ is picked}] \cdot c_S = \sum_{S \in \mathcal{S}} p_S \cdot c_S = \text{OPT},$$

where OPT is the cost of the optimal solution to the linear program. Let us compute the probability that an element $e \in U$ is covered by $\mathcal{C}$. Suppose that e occurs in k sets of $\mathcal{S}$, and the probabilities associated with these sets are $p_1, \ldots, p_k$. Since e is fractionally covered in the optimal solution, $p_1 + p_2 + \cdots + p_k \geq 1$. Using elementary calculus, it is easy to show that under this condition, the probability that e is covered by $\mathcal{C}$ is minimized when each of the p_i's is $1/k$. So,

$$\mathbf{Pr}[e \text{ is covered by } \mathcal{C}] \geq 1 - \left(1 - \frac{1}{k}\right)^k \geq 1 - \frac{1}{\mathbf{e}},$$

where $\mathbf{e}$ represents the base of the natural logarithms. Thus, each element is covered with constant probability by $\mathcal{C}$.

To get a complete set cover, independently pick $c \log n$ such sub-collections, and obtain their union, say $\mathcal{C}'$, where c is a constant such that

$$\left(\frac{1}{e}\right)^{c \log n} \leq \frac{1}{2n}.$$

Now,

$$\mathbf{Pr}[e \text{ is not covered by } \mathcal{C}'] \leq \left(\frac{1}{e}\right)^{c \log n} \leq \frac{1}{2n}.$$

Summing over all elements $e \in U$,

$$\mathbf{Pr}[\text{some } e \in U \text{ is not covered by } \mathcal{C}'] \leq n \cdot \frac{1}{2n} \leq \frac{1}{2}.$$

Therefore $\mathcal{C}'$ is a set cover with probability at least $1/2$. If $\mathcal{C}'$ is not a set cover, the above procedure is repeated, until a valid sub-collection is found. Clearly, the expected number of repetitions needed is at most 2. The expected cost of $\mathcal{C}'$ is at most $2 \cdot \mathrm{OPT} \cdot (c \log n) = O(\log n) \cdot \mathrm{OPT}$.

We have presented perhaps the simplest randomized rounding algorithm for set cover. Using more elaborate probabilistic machinery, one can get algorithms achieving essentially the same factor as the greedy algorithm presented in Chapter 2.

Exercise 11.5 Show that with some constant probability, $\mathcal{C}$ covers at least half the elements at a cost of at most OPT.

Chapter 12

LP-duality based analysis for set cover

As stated in Chapter 11, there is a more sophisticated way, than rounding, of using the LP-relaxation. Let us call the LP-relaxation the primal program. We know from Chapter 10 that any feasible solution to the dual gives a lower bound on the primal, and hence also on the original integer program. Now, we devise an algorithm that finds an integral solution to the primal and simultaneously a feasible (fractional) solution to the dual. The approximation guarantee is established by comparing the cost of these two solutions. The main advantage over rounding is that instead of having to work with an arbitrary optimal solution to the LP-relaxation, we can pick the two solutions carefully so they have nice combinatorial properties. Another advantage is that the algorithm can be made more efficient, since it does not have to first solve the LP-relaxation optimally.

The reader may suspect that from the viewpoint of approximation guarantee, the current method is inferior to rounding, since an optimal solution to the primal gives a tighter lower bound than a feasible solution to the dual. Let us define the *integrality gap* of a minimizing integer program to be the maximum ratio of an optimal integral and optimal fractional solution (for a maximizing integer program, we will seek the minimum such ratio). Clearly, the integrality gap of the integer programming formulation being used is the best approximation guarantee one can hope to achieve by using either of the two approaches. Interestingly enough, for most problems studied, the approximation guarantee researchers have managed to establish using the LP-duality approach is essentially equal to the integrality gap of the integer programming formulation.

We will use this strategy to re-analyze the natural greedy algorithm for the minimum set cover problem given in Chapter 2. Recall that in Chapter 2 we had deferred giving the lower bounding method on which this algorithm was based. The method is precisely the one described above. The power of this approach will become apparent when we show the ease with which it extends to solving generalizations of the set cover problem.

Introducing a variable y_e corresponding to each element $e \in U$, we obtain the dual for (11.2):

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in U} y_e && && (12.1) \\ \text{subject to} \quad & \sum_{e:\ e \in S} y_e \leq c(S), && S \in \mathcal{S} \\ & y_e \geq 0, && e \in U \end{aligned}$$

Intuitively, why is 12.1 the dual of 11.2? In our experience, this is not the right question to be asked. As stated in Chapter 10, there is a purely mechanical procedure for obtaining the dual of a linear program. Once the dual is obtained, one can devise intuitive, and possibily physically meaningful, ways of thinking about it. Using this mechanical procedure, one can obtain the dual of a complex linear program in a fairly straightforward manner. Indeed, the LP-duality based approach derives its wide applicability from this fact.

An intuitive way of thinking about program 12.1 is that it is packing "stuff" into elements, trying to maximize the total amount packed, subject to the constraint that no set is *overpacked*; a set is said to be overpacked if the total amount packed into its elements exceeds its cost. Thus, (11.2) and (12.1) can be thought of as a *covering* and *packing* linear programs respectively. Whenever the coefficients in the constraint matrix, objective function and right hand side are all non-negative, we get such a pair of linear programs. Such pairs of programs will arise frequently in subsequent chapters.

Let us re-analyze the natural greedy set cover algorithm using LP-duality theory. Recall that the algorithm picks the most cost-effective set in each iteration, and for each element e, $\text{price}(e)$ is defined to be the average cost at which it is covered for the first time (see Algorithm 2.7). The prices of elements give us a setting for the dual variables as follows:

$$y_e \leftarrow \frac{\text{price}(e)}{H_n}.$$

Lemma 12.1 *The vector $\boldsymbol{y}$ defined above is a feasible solution for the dual program.*

Proof : Essentially, we need to show that no set is overpacked by the solution $\boldsymbol{y}$. Consider a set $S \in \mathcal{S}$ consisting of k elements. Number the elements in the order in which they are covered by the algorithm, breaking ties arbitrarily, say $e_1, \ldots, e_k$.

Consider the iteration in which the algorithm covers element e_i. At this point, S contains at least $k-i+1$ uncovered elements. So, in this iteration, S itself can cover e_i at an average cost of at most $\frac{c(S)}{k-i+1}$. Since the algorithm chose the most cost-effective set in this iteration, $\text{price}(e_i) \le \frac{c(S)}{k-i+1}$. So,

$$y_{e_i} \le \frac{1}{H_n} \cdot \frac{c(S)}{k-i+1}.$$

Summing over all elements in S,

$$\sum_{i=1}^{k} y_{e_i} \le \frac{c(S)}{H_n} \cdot \left(\frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{1}\right) = \frac{H_k}{H_n} \cdot c(S) \le c(S).$$

Therefore, S is not overpacked. □

Theorem 12.2 *The approximation guarantee of the greedy algorithm is H_n.*

Proof : The cost of the set cover picked is

$$\sum_{e \in U} \text{price}(e) = H_n \left(\sum_{e \in U} y_e\right) \le H_n \cdot \text{OPT},$$

where OPT denotes the cost of the optimal fractional set cover. The last inequality follows from the fact that $\boldsymbol{y}$ is dual feasible. $\square$

Exercise 12.3 Show that the analysis given above actually establishes an approximation guarantee of H_k, where k is size of the largest set in the given instance.

As shown in Example 2.5, this analysis is tight. As a corollary of Theorem 12.2 we also get an upper bound of H_n on the integrality gap of the integer programming formulation (11.1). Can we hope to design a better algorithm using this formulation, or is this upper bound on the integrality gap tight? The next example shows that this bound is essentially tight. So, to obtain a better algorithm using the current approach, we will first have to think of an integer programming formulation with a smaller integrality gap. Assuming $\mathbf{P} \neq \mathbf{NP}$, such a formulation does not exist, since it has been proven that under this assumption, one cannot obtain a better approximation guarantee for the set cover problem.

Example 12.4 Consider the following set cover instance. Let $n = 2^k - 1$, where k is a positive integer, and let $U = \{e_1, e_2, \ldots, e_n\}$. For $1 \leq i \leq n$, consider i written as a k-bit number. We can view this as a k-dimensional vector over $GF[2]$; let $\vec{i}$ denote this vector. For $1 \leq i \leq n$ define set $S_i = \{e_j | \ \vec{i} \cdot \vec{j} = 1\}$, where $\vec{i} \cdot \vec{j}$ denotes the inner product of these two vectors. Finally, let $\mathcal{S} = \{S_1, \ldots, S_n\}$, and define the cost of each set to be 1.

It is easy to check that each set contains $2^{k-1} = \frac{n+1}{2}$ elements, and each element is contained in $\frac{n+1}{2}$ sets. So, $x_i = \frac{2}{n+1}, 1 \leq i \leq n$ is a fractional set cover. Its cost is $\frac{2n}{n+1}$.

Next, we will show that any integral set cover must pick at least k of the sets, by showing that the union of any $k-1$ sets must leave some element uncovered. Let $i_1, \ldots, i_{k-1}$ be the indices of $k-1$ sets, and let $\boldsymbol{A}$ be a $(k-1) \times k$ matrix over $GF[2]$ whose rows consist of vectors $\vec{i}_1, \ldots, \vec{i}_{k-1}$ respectively. Let $\boldsymbol{B}$ be the $(k-1) \times (k-1)$ submatrix consisting of the first $k-1$ columns of $\boldsymbol{A}$. Let $\vec{z} = (z_1, \ldots, z_k)^T$ be a k-dimensional column vector, where $z_1, \ldots, z_k$ are indeterminates over $GF[2]$. Let us show that the equation $\boldsymbol{A}\vec{z} = \vec{0}$ has a solution $\vec{z} \neq 0$; this gives an element of U that is not covered by any of the $k-1$ sets. If $\boldsymbol{B}$ is singular, one can find such a solution after first setting $z_k = 0$. If $\boldsymbol{B}$ is non-singular, there is a solution with $z_k = 1$.

Therefore, any integral set cover has cost at least $k = \log_2(n+1)$. Hence, the lower bound on the integrality gap established by this example is

$$\left(\frac{n+1}{2n}\right) \cdot \log_2(n+1) > \frac{\log_2 n}{2}.$$

$\square$

Generalizations of the set cover problem

The greedy algorithm and its analysis using LP-duality extend naturally to several generalizations of the set cover problem:

- **Set multicover:** each element needs to be covered a specified integer number of times
- **Multiset multicover:** we are given a collection of multisets, rather than sets, of U (a multi-set contains a specified number of copies of each element)
- **Covering integer programs:** these are integer programs of the form:

$$\begin{array}{ll} \text{minimize} & \boldsymbol{c} \cdot \boldsymbol{x} \\ \text{subject to} & A\boldsymbol{x} \geq \boldsymbol{b}, \end{array}$$

where all entries in $\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}$ are non-negative, and $\boldsymbol{x}$ is required to be non-negative and integral.

For the first two problems, let us also consider variants in which we impose the additional constraint that each set be picked at most once. One can obtain an $O(\log n)$ factor approximation algorithm for covering integer programs, and H_n factor algorithms for the rest of the problems. In this section, we will present an H_n factor approximation algorithm for set multicover with the constraint that each set can be picked at most once, and will leave the rest of the problems as exercise.

Let $r_e \in Z_+$, be the coverage requirement for each element $e \in U$. The integer programming formulation of set multicover is not very different from that of set cover:

$$\begin{array}{lll} \text{minimize} & \displaystyle\sum_{S \in \mathcal{S}} c(S) x_S & \\ \text{subject to} & \displaystyle\sum_{S:\ e \in S} x_S \geq r_e, & e \in U \\ & x_S \in \{0, 1\}, & S \in \mathcal{S} \end{array}$$

Notice, however, that in the LP-relaxation, the constraints $x_S \leq 1$ are no longer redundant; if we drop them, then a set may be picked multiple number of times to satisfy the coverage requirement of an element. Thus, the LP-relaxation looks different from that for set cover. In particular, because of the negative numbers in the constraint matrix and the right hand side, it is not even a covering linear program; this is the reason we are providing details for this case.

$$\begin{array}{lll} \text{minimize} & \displaystyle\sum_{S \in \mathcal{S}} c(S) x_S & \\ \text{subject to} & \displaystyle\sum_{S:\ e \in S} x_S \geq r_e, & e \in U \\ & -x_S \geq -1 & S \in \mathcal{S} \\ & x_S \geq 0, & S \in \mathcal{S} \end{array}$$

The additional constraints in the primal lead to new variables, z_S, in the dual, which becomes:

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{e \in U} r_e y_e - \sum_{S \in \mathcal{S}} z_S & \\ \text{subject to} & \left(\displaystyle\sum_{e:\ e \in S} y_e \right) - z_S \leq c(S), & S \in \mathcal{S} \\ & y_e \geq 0, & e \in U \\ & z_S \geq 0, & S \in \mathcal{S} \end{array}$$

The algorithm is again greedy. Let us say that element e is *alive* if it is covered by fewer than r_e picked sets. In each iteration, the algorithm picks, from amongst the currently unpicked sets, the most cost-effective set, where the *cost-effectiveness* of a set is defined to be the average cost at which it covers alive elements. The algorithm halts when there are no more alive elements, i.e., each element has been covered to the extent of its requirement.

When a set S is picked, its cost is distributed equally among the alive elements it covers as follows: if S covers e for the j^{th} time, we set $\text{price}(e, j)$ to the current cost-effectiveness of S. Clearly, the cost-effectiveness of sets picked is non-decreasing. Hence, for each element e, $\text{price}(e, 1) \leq \text{price}(e, 2) \leq \ldots \leq \text{price}(e, r_e)$.

At the end of the algorithm, the dual variables are set as follows: For each $e \in U$, let $y_e = \frac{1}{H_n}\text{price}(e, r_e)$. For each $S \in \mathcal{S}$ that is picked by the algorithm, let

$$z_S = \frac{1}{H_n}\left[\sum_{e \text{ covered by } S} (\text{price}(e, r_e) - \text{price}(e, j_e))\right],$$

where j_e is the copy of e that is covered by S. Notice that since $\text{price}(e, j_e) \leq \text{price}(e, r_e)$, z_S is non-negative. If S is not picked by the algorithm, z_S is defined to be 0.

Lemma 12.5 *The pair $(\boldsymbol{y}, \boldsymbol{z})$ is a feasible solution for the dual program.*

Proof : Consider a set $S \in \mathcal{S}$ consisting of k elements. Number its elements in the order in which their requirements are fulfilled, i.e., the order in which they stopped being alive. Let the ordered elements be $e_1, \ldots, e_k$.

First, assume that S is not picked by the algorithm. When the algorithm is about to cover the last copy of e_i, S contains at least $k - i + 1$ alive elements, so

$$\text{price}(e_i, r_{e_i}) \leq \frac{c(S)}{k - i + 1}.$$

Since z_S is zero, we get that

$$\left(\sum_{i=1}^{k} y_{e_i}\right) - z_S = \frac{1}{H_n}\sum_{i=1}^{k}\text{price}(e_i, r_{e_i}) \leq \frac{c(S)}{H_n} \cdot \left(\frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{1}\right) \leq c(S).$$

Next, assume that S is picked by the algorithm, and before this happens, $k' \geq 0$ elements of S are already completely covered. Then

$$\left(\sum_{i=1}^{k} y_{e_i}\right) - z_S = \frac{1}{H_n} \cdot \left[\sum_{i=1}^{k'}\text{price}(e_i, r_{e_i}) + \sum_{i=k'+1}^{k} \text{price}(e_i, j_i)\right],$$

where S covers j_i^{th} copy of e_i, for each $i \in \{k'+1, \ldots, k\}$. But $\sum_{i=k'+1}^{k}\text{price}(e_i, j_i) = \text{price}(S)$, since the cost of S is equally distributed among the copies it covers. Finally consider elements e_i, $i \in \{1, \ldots, k'\}$. When the last copy of e_i is being covered, S is not yet picked and covers at least $k - i + 1$ alive elements. So, $\text{price}(e_i, r_{e_i}) \leq c(S)/(k - i + 1)$. Therefore,

$$\left(\sum_{i=1}^{k} y_{e_i}\right) - z_S \leq \frac{c(S)}{H_n} \cdot \left(\frac{1}{n} + \cdots + \frac{1}{n - k' + 1} + 1\right) \leq c(S).$$

Hence, $(\boldsymbol{y}, \boldsymbol{z})$ is feasible for the dual program. □

Theorem 12.6 *The greedy algorithm for set multicover has an approximation factor of H_n.*

Proof : The value of the dual feasible solution $(\boldsymbol{y}, \boldsymbol{z})$ is

$$\sum_{e\in U} r_e y_e - \sum_{S\in\mathcal{S}} z_S = \frac{1}{H_n} \sum_{e\in U} \sum_{j=1}^{r_e} \text{price}(e, j).$$

Since the cost of the sets picked by the algorithm is distributed among the covered elements, it follows that the total cost of the multicover produced by the algorithm is

$$\sum_{e\in U} \sum_{j=1}^{r_e} \text{price}(e, j).$$

So, by weak duality, the algorithm produces a multicover of cost

$$H_n \cdot \left[\sum_{e\in U} r_e y_e - \sum_{S\in\mathcal{S}} z_S \right] \leq H_n \cdot \text{OPT}$$

□

Exercise 12.7 Give an $O(\log n)$ factor approximation algorithm for covering integer programs and H_n factor approximation algorithms for the rest of the generalizations of set cover stated above.

Exercise 12.8 Consider the following variant on the set multi-cover problem: Let U be the universal set, $|U| = n$, and $\mathcal{S}$ a collection of subsets of U. For each $S \in \mathcal{S}$, its cost is given as a function of time, $t \in \{1, \ldots, T\}$. Each of these cost functions is non-increasing with time. In addition, for each element in U, a coverage requirement is specified, again as a function of time; these functions are non-decreasing with time. The problem is to pick sets at a minimum total cost so that the coverage requirements are satisfied for each element at each time. A set can be picked any number of times; the cost of picking a set depends on the time at which it is picked. Once picked, the set remains in the cover for all future times at no additional cost. Give an H_n factor algorithm for this problem. (An $H_{(n\cdot T)}$ factor algorithm is straightforward.)

Chapter 13

The primal-dual schema

LP-duality theory not only provides a method of lower bounding the cost of the optimal solution for several **NP**-hard problems, but also provides a general schema for obtaining the approximation algorithm itself: the primal-dual schema. In this chapter, we will first present the central ideas behind this schema, and then illustrate them in a simple setting by again obtaining an f factor algorithm for the minimum set cover problem, where f is the frequency of the most frequent element. An algorithm achieving this factor, using rounding, was presented in Chapter 11.

The primal-dual schema has its origins in the design of exact algorithms. In that setting, this schema yielded the most efficient known algorithms to some of the cornerstone problems in **P**, including matching, network flow and shortest paths. These problems have the property that their LP-relaxations have integral optimal solutions. By Theorem 10.3 we know that optimal solutions to linear programs are characterized by fact that they satisfy all the complementary slackness conditions. In fact, the primal-dual schema is driven by these conditions: starting with initial feasible solutions to the primal and dual programs, it iteratively satisfies more and more complementary slackness conditions; when they are all satisfied, both solutions must be optimal. During the iterations, the primal is always modified integrally, so that eventually we get an integral optimal solution.

In the case of **NP**-hard problems, we cannot be looking for an optimal solution to the LP-relaxation since, in general, none of these solutions may be integral. Does this rule out a complementary slackness condition driven approach? Interestingly enough, the answer is "No". It turns out that the algorithm can be driven by a suitable relaxation of these conditions!

Overview of the schema

Let us consider the following primal program, written in standard form:

$$\begin{array}{lll} \text{minimize} & \displaystyle\sum_{j=1}^{n} c_j x_j & \\ \text{subject to} & \displaystyle\sum_{j=1}^{n} a_{ij} x_j \geq b_i, & i = 1, \ldots, m \\ & x_j \geq 0, & j = 1, \ldots, n \end{array}$$

where a_{ij}, b_i, and c_j are specified in the input. The dual program is:

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{i=1}^{m} b_i y_i & \\ \text{subject to} & \displaystyle\sum_{i=1}^{m} a_{ij} y_i \leq c_j, & j = 1, \ldots, n \\ & y_i \geq 0, & i = 1, \ldots, m \end{array}$$

All known approximation algorithms using the primal-dual schema run by ensuring the primal complementary slackness conditions and relaxing the dual conditions. They find primal and dual feasible solutions satisfying:

Primal complementary slackness conditions
For each $1 \leq j \leq n$: either $x_j = 0$ or $\sum_{i=1}^{m} a_{ij} y_i = c_j$; and

Relaxed dual complementary slackness conditions
For each $1 \leq i \leq m$: either $y_i = 0$ or $\sum_{j=1}^{n} a_{ij} x_j \leq \alpha \cdot b_i$,

where $\alpha > 1$ is a constant; if α were set to 1, we would get the usual condition.

Proposition 13.1 *If $\boldsymbol{x}$ and $\boldsymbol{y}$ are primal and dual feasible solutions satisfying the conditions stated above then*

$$\sum_{j=1}^{n} c_j x_j \leq \alpha \cdot \sum_{i=1}^{m} b_i y_i.$$

Proof :

$$\sum_{j=1}^{n} c_j x_j = \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i \leq \alpha \sum_{i=1}^{m} b_i y_i. \tag{13.1}$$

The first equality is obtained by applying the primal conditions, and the inequality follows by applying the relaxed dual conditions. The second equality follows by simply changing the order of summation. □

The algorithm starts with a primal infeasible solution and a dual feasible solution; these are usually the trivial solutions $\boldsymbol{x} = \boldsymbol{0}$ and $\boldsymbol{y} = \boldsymbol{0}$. It iteratively improves the feasibility of the primal solution, and the optimality of the dual solution, ensuring that in the end a primal feasible solution is obtained and all conditions stated above are satisfied. The primal solution is always extended integrally, thus ensuring that the final solution is integral. The improvements to the primal and the dual go hand-in-hand: the current primal solution is used to determine the improvement to the dual, and vice versa. Finally, the cost of the dual solution is used as a lower bound on OPT, and by Proposition 13.1, the approximation guarantee of the algorithm is α.

Despite being so general, this schema leaves sufficient scope for exploiting the special combinatorial structure of the specific problem: in designing the solution improving algorithms and in comparing the final solutions, thereby yielding very good approximation guarantees. Since numerous **NP**-hard problems can be expressed as integer programming problems, the primal-dual schema should find even greater applicability in the setting of approximation algorithms than exact algorithms.

Primal-dual schema applied to set cover

Let us obtain a factor f algorithm for the set cover problem using the primal-dual schema. A statement of this problem appears in Chapter 11, its LP-relaxation is given in (11.2) and dual program is given in (12.1). As shown in Chapter 11, the weighted vertex cover problem can be stated as a set cover problem with $f = 2$.

Let us start by stating the primal complementary slackness conditions and relaxing the dual conditions appropriately.

Primal conditions: These can be written as:

$$\forall S \in \mathcal{S} : x_S \neq 0 \Rightarrow \sum_{e:\ e \in S} y_e = c(S).$$

Set S will said to be *tight* if $\sum_{e:\ e\in S} y_e = c(S)$. Since we will increment the primal variables integrally, we can state the conditions as: *Pick only tight sets in the cover.*
Clearly, in order to maintain dual feasiblity, we are not allowed to overpack any set.

Dual conditions: The dual conditions will be relaxed with $\alpha = f$.

$$\forall e : y_e \neq 0 \Rightarrow \sum_{S:\ e \in S} x_S \leq f$$

Since we will find a 0/1 solution for $\boldsymbol{x}$, these conditions are equivalently to:
Cover each element with non-zero dual at most f times.
Since each element is in at most f sets, this condition is trivially satisfied for all elements.

The two sets of conditions naturally suggest the following algorithm:

Algorithm 13.2 (Set cover – factor f)

1. **Initialization:** $\boldsymbol{x} \leftarrow \boldsymbol{0}$; $\boldsymbol{y} \leftarrow \boldsymbol{0}$
2. Until all elements are covered, do:

   Pick an uncovered element, say e, and raise y_e until some set goes tight.

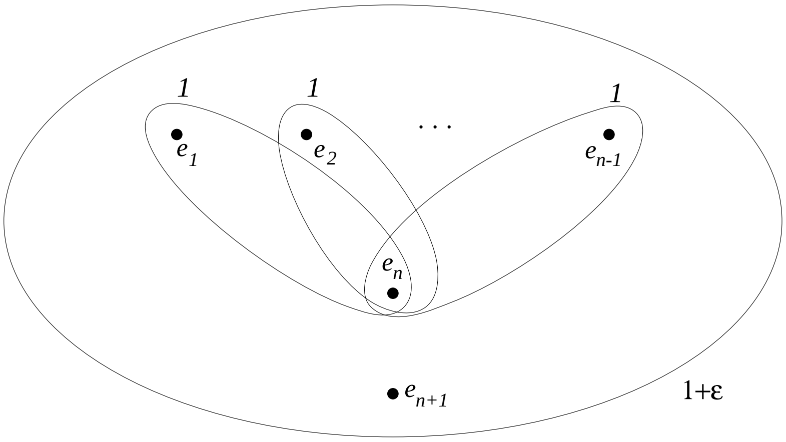
   Pick all tight sets in the cover and update $\boldsymbol{x}$.

   Declare all the elements occurring in these sets as "covered".
3. Output the set cover $\boldsymbol{x}$.

Theorem 13.3 *Algorithm 13.2 achieves an approximation factor of f.*

Proof : Clearly there will be no uncovered elements and no overpacked sets at the end of the algorithm. So, the primal and dual solutions will both be feasible. Since they satisfy the relaxed complementary slackness conditions with $\alpha = f$, by Proposition refprimal-dual.prop-relax the approximation factor is f. □

Example 13.4 A tight example for this algorithm is provided by the following set system:



Here, $\mathcal{S}$ consists of $n-1$ sets of cost 1, $\{e_1, e_n\}, \ldots, \{e_{n-1}, e_n\}$, and one set of cost $1+\epsilon$, $\{e_1, \ldots, e_{n+1}\}$, for a small $\epsilon > 0$. Since e_n appears in all n sets, this set system has $f = n$.

Suppose the algorithm raises y_{e_n} in the first iteration. When y_{e_n} is raised to 1, all sets $\{e_i, e_n\}$, $i = 1, \ldots, n-1$ go tight. They are all picked in the cover, thus covering the elements $e_1, \ldots, e_n$. In the second iteration, $y_{e_{n+1}}$ is raised to ϵ and the set $\{e_1, \ldots, e_{n+1}\}$ goes tight. The resulting set cover has a cost of $n + \epsilon$, whereas the optimum cover has cost $1 + \epsilon$. $\square$

Chapter 14

Multicut and integer multicommodity flow in trees

In Chapter 13 we used the primal-dual schema to derive a factor 2 algorithm for the weighted vertex cover problem. This algorithm was particularly easy to obtain because the relaxed dual complementary slackness conditions were automatically satisfied in any integral solution. In this chapter, we will use the primal-dual schema to obtain an algorithm for a generalization of this problem; this time enforcing the relaxed dual complementary slackness conditions will be a non-trivial part of the algorithm. Furthermore, we will introduce the procedure of reverse delete for pruning the primal solution – this is a critical procedure in several other primal-dual algorithms as well.

Problem 14.1 (Minimum multicut) Let $G = (V, E)$ be an undirected graph with non-negative capacity c_e for each edge $e \in E$. Let $\{(s_1, t_1), \ldots, (s_k, t_k)\}$ be a specified set of pairs of vertices, where each pair is distinct, but vertices in different pairs are not required to be distinct. A *multicut* is a set of edges whose removal separates each of the pairs. The problem is to find a minimum capacity multicut in G.

Notice that the multiway cut problem, defined in Chapter 4, is a special case of the minimum multicut problem, since separating terminals $s_1, \ldots, s_l$ is equivalent to separating the vertex pairs (s_i, s_j), for $1 \leq i < j \leq l$. This observation implies that the minimum multicut problem is **NP**-hard even for $k = 3$, since the multiway cut problem is **NP**-hard for the case of 3 terminals. For $k = 2$, the minimum multicut problem is in **P**.

In Chapter 17 we will obtain an $O(\log k)$ factor approximation algorithm for the minimum multicut problem. In this chapter, we will obtain a factor 2 algorithm for the special case when G is restricted to be a tree. Notice that in this case there is a unique path between s_i and t_i, and the multicut must pick an edge on this path to disconnect s_i from t_i. The problem looks deceptively simple; the following lemma should convince the reader that in fact it is quite non-trivial.

Lemma 14.2 *The minimum cardinality vertex cover problem is polynomial time equivalent to the minimum multicut problem when restricted to trees of height 1 and unit capacity edges.*

Proof : Let us first reduce the restriction of the multicut problem to the minimum vertex cover problem. Let G be a tree of height 1 with root r and leaves $v_1, \ldots, v_n$, and let $(s_1, t_1), \ldots, (s_k, t_k)$ be the pairs of vertices that need to be be disconnected. If the root r is contained in a pair (s_i, t_i), the edge (s_i, t_i) must be included in every multicut. So, w.l.o.g. assume that r does not occur in any of the pairs.

Construct graph H with vertex set $\{v_1, \ldots, v_n\}$ and k edges $(s_1, t_1), \ldots, (s_k, t_k)$. Vertex v_i in H corresponds to the edge (v_i, r) in G. Now, it is easy to see that a subset of vertices in H is a vertex cover for H iff the corresponding set of edges in G is a multicut for G. This establishes the reduction.

The reduction in the other direction should now be obvious: Given a graph H, construct a tree G of height 1 whose leaves correspond to the vertices of H. Each edge of H specifies a vertex pair that needs to be disconnected in G. $\square$

By Lemma 14.2, the minimum multicut problem is **NP**-hard even if restricted to trees of height 1 and unit capacity edges. By essentially the same proof, if the edge capacities are allowed to be arbitrary, but the tree is still of height 1, the problem is equivalent to the weighted vertex cover problem.

Since we want to apply LP-duality theory for designing the algorithm, let us first give an integer programming formulation for the problem, and obtain its LP-relaxation. Introduce a 0/1 variable d_e for each edge $e \in E$, which will be set to 1 iff e is picked in the multicut. Let p_i denote the unique path between s_i and t_i in the tree.

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c_e d_e \\ \text{subject to} \quad & \sum_{e \in p_i} d_e \geq 1, \quad && i \in \{1, \ldots, k\} \\ & d_e \in \{0, 1\}, && e \in E \end{aligned}$$

The LP-relaxation is obtained by replacing the constraint $d_e \in \{0, 1\}$ by $d_e \geq 0$; as before, there is no need to add the constraint $d_e \leq 1$ explicitly.

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c_e d_e \\ \text{subject to} \quad & \sum_{e \in p_i} d_e \geq 1, \quad && i \in \{1, \ldots, k\} \\ & d_e \geq 0, && e \in E \end{aligned}$$

We can now think of d_e as specifying the fractional extent to which edge e is picked. A solution to this linear program is a *fractional multicut*: on each path p_i, the sum of fractions of edges picked is at least 1. In general, minimum fractional multicut may be strictly cheaper than minimum integral multicut. This is illustrated in Example 14.3.

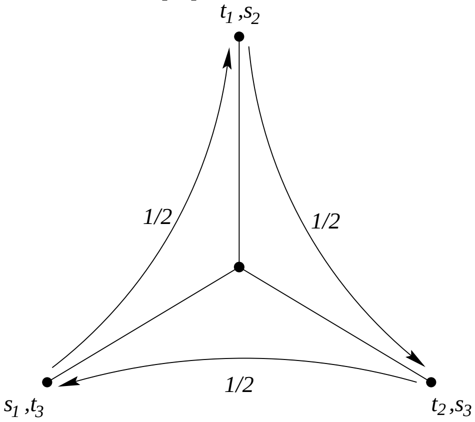
We will interpret the dual program as specifying a *multicommodity flow* in G, with a separate commodity corresponding to each vertex pair (s_i, t_i). Dual variable f_i will denote the amount of this commodity routed along the unique path from s_i to t_i.

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{k} f_i && \qquad (14.1) \\ \text{subject to} \quad & \sum_{i:\ e \in p_i} f_i \leq c_e, \quad && e \in E \\ & f_i \geq 0, && i \in \{1, \ldots, k\} \end{aligned}$$

The commodities are routed concurrently. The object is to maximize the sum of the commodities routed, subject to the constraint that the sum of flows routed through an edge is bounded by the capacity of the edge. Notice that the sum of flows through an edge (u, v) includes flow going in either direction, u to v, and v to u.

By the Weak Duality Theorem, a feasible multicommodity flow gives a lower bound on the minimum fractional multicut, and hence also on the minimum integral multicut. By the LP-Duality Theorem, minimum fractional multicut equals maximum multicommodity flow.

Example 14.3 Consider the following graph with unit capacity edges and 3 vertex pairs:



The arrows show how to send $\frac{3}{2}$ units of flow by sending $\frac{1}{2}$ unit of each commodity. Picking each edge to the extent of $\frac{1}{2}$ gives a multicut of capacity $\frac{3}{2}$ as well. So, these must be optimal solutions to the primal and dual programs. On the other hand, any integral multicut must pick at least two of the three edges in order to disconnect all three pairs. Hence, minimum integral multicut has capacity 2. □

Finally, let us state one more problem.

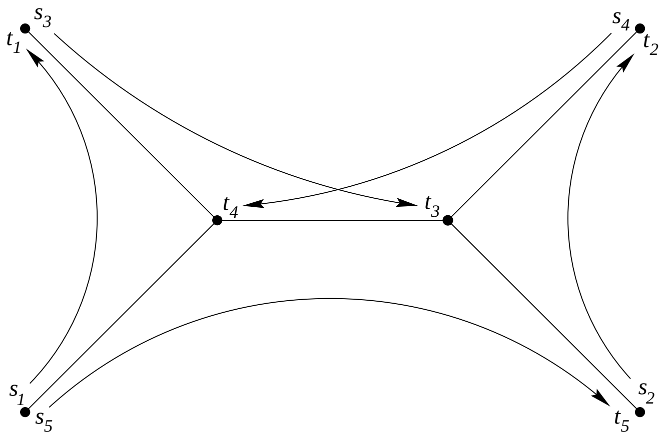
Problem 14.4 (Maximum integer multicommodity flow) Graph G and the pairs are specified as in the minimum multicut problem; however, the edge capacities are all integral. A separate commodity is defined for each (s_i, t_i) pair. The object is maximize the sum of the commodities routed, subject to edge capacity constraints, and that on each path, each commodity is routed integrally.

Let us consider this problem when G is restricted to be a tree. If in (14.1), the variables are constrained to be non-negative integral, we would get an integer programming formulation for this problem. Clearly, the objective function value of this integer program is bounded by that of the linear program (14.1). Furthermore, the best fractional flow may be strictly larger. For instance, in Example 14.3, maximum integral multicommodity flow is 1, since sending 1 unit of any of the three commodities will saturate two of the edges. This problem is **NP**-hard, even for trees of height 3.

Exercise 14.5 Show that the maximum integer multicommodity flow problem on trees of height 1 is equivalent to the maximum weight matching problem, and hence is in **P**.

Exercise 14.6 Give a polynomial time algorithm for computing a maximum integer multicommodity flow on unit capacity trees of arbitrary height. Assume that you are allowed to use a subroutine for the maximum matching problem.

Exercise 14.7 Find the best integral and fractional multicut and multicommodity flow in the following graph. All capacities are 1, and the specified pairs are $(s_1, t_1), \ldots (s_5, t_5)$. Notice that the optimal fractional multicut is not half intergal. In contrast, it is known that LP-relaxation of the multiway cut problem always has a half-integral solution, even in general graphs.



The algorithm

We will use the primal-dual schema to obtain an algorithm that simultaneously finds a multicut and an integer multicommodity flow that are within a factor of 2 of each other, provided the given graph is a tree. Hence, we get approximation algorithms for both problems, of factor 2 and $\frac{1}{2}$ respectively.

Let us define the multicut LP to be the primal program. An edge e is *saturated* if the total flow through it equals its capacity. Our first task is to relax the dual complementary slackness conditions appropriately.

Primal conditions: For each $e \in E$, $d_e \neq 0 \Rightarrow \sum_{i:\ e \in p_i} f_i = c_e$.
Equivalently, *any edge picked in the multicut must be saturated.*

Relaxed dual conditions: Let us relax the dual conditions to:
For each $i \in \{1, \ldots, k\}$, $f_i \neq 0 \Rightarrow \sum_{e \in p_i} d_e \leq 2$.
Equivalently, *at most two edges can be picked from a path carrying non-zero flow.*
Clearly, we must pick at least one edge from each (s_i, t_i) path simply to ensure feasibility of the multicut.

Let us root the tree G at an arbitrary vertex. Define the *depth* of vertex v to be the length of the path from v to the root; the depth of the root is 0. For two vertices $u, v \in V$, let $\mathrm{lca}(u, v)$ denote the *lowest common ancestor* of u and v, i.e. the minimum depth vertex on the path from u to v. Let e_1 and e_2 be two edges on a path from a vertex to the root. If e_1 occurs before e_2 on this path, then, e_1 is said to be *deeper* than e_2.

The algorithm starts with an empty multicut and flow, and interatively improves the feasibility of the primal solution and the optimality of the dual solution. In an iteration, it picks the deepest unprocessed vertex, say v, and greedily routes integral flow between pairs that have v as their lowest common ancestor. When no more flow can be routed between these pairs, all edges that got saturated, i.e., got saturated, in this iteration are added to the list D in arbitrary order. When all the vertices have been processed, D will be a multicut; however, it may have redundant edges. To remove them, a *reverse delete* step is performed: Edges are considered in the reverse of the order in which they were added to D, and if the deletion of edge e from D still gives a valid multicut, e is discarded from D.

Algorithm 14.8 (Multicut and integer multicommodity flow in trees)

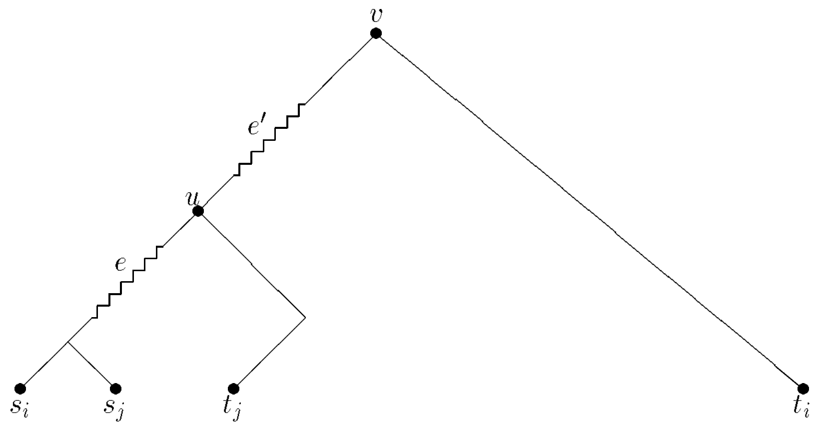
1. **Initialization:** $f \leftarrow \mathbf{0}$; $D \leftarrow \emptyset$
2. **Flow routing:** For each vertex v, in non-increasing order of depth, do:

   For each pair (s_i, t_i) such that $\mathrm{lca}(s_i, t_i) = v$, greedily route integral flow from s_i to t_i.

   Add to D all edges that got saturated in the current iteration, in arbitrary order.
3. Let $e_1, e_2, \ldots, e_l$ be the ordered list of edges in D.
4. **Reverse delete:** For $j = l$ downto 1 do:
   If $D - \{e_j\}$ is a multicut in G, then $D \leftarrow D - \{e_j\}$
5. Output the flow and multicut D.

Lemma 14.9 *Let (s_i, t_i) be a pair with non-zero flow, and let $\mathrm{lca}(s_i, t_i) = v$. At most one edge is picked in the multicut from each of the two paths, s_i to v, and t_i to v.*

Proof : The argument is the same for each path. Suppose two edges e and e' are picked from the s_i–v path, with e being the deeper edge. Clearly, e' must be in D all through reverse delete. Consider the moment during reverse delete when edge e is being tested. Since e is not discarded, there must be a pair, say (s_j, t_j), such that e is the only edge of D on the s_j–t_j path. Let u be the lowest common ancestor of s_j and t_j. Since e' does not lie on the s_j–t_j path, u must be deeper than e', and hence deeper than v. After u has been processed, D must contain an edge from the s_j–t_j path, say e''.



Since non-zero flow has been routed from s_i to t_i, e must be added during or after the iteration in which v is processed. Since v is an ancestor of u, e is added after e''. So e'' must be in D when e is being tested. This contradicts the fact that at this moment e is the only edge of D on the s_j–t_j path. □

Theorem 14.10 *Algorithm 14.8 achieves approximation guarantees of factor 2 for the minimum multicut problem and factor $\frac{1}{2}$ for the maximum integer multicommodity flow problem on trees.*

Proof : The flow found at the end of Step 2 is maximal, and since at this point D contains all the saturated edges, D is a multicut. Since the reverse delete step only discards redundant edges,

D is a multicut after this step as well. Thus feasible solutions have been found for both, the flow and the multicut.

Since each edge in the multicut is saturated, the primal conditions are satisfied. By lemma 14.9, at most two edges have been picked in the multicut from each path carrying non-zero flow. Therefore, the relaxed dual conditions are also satisfied. Hence, by Proposition 13.1, the capacity of the multicut found is within twice the flow. Since a feasible flow is a lower bound on the optimal multicut, and a feasible multicut is an upper bound on the optimal integer multicommodity flow, the claim follows. □

Exercise 14.11 Prove that if e and e' are both in D in Step 3, and e is deeper than e', e is added before or in the same iteration as e'.

Finally, we obtain the following approximate min-max relation from Theorem 14.10:

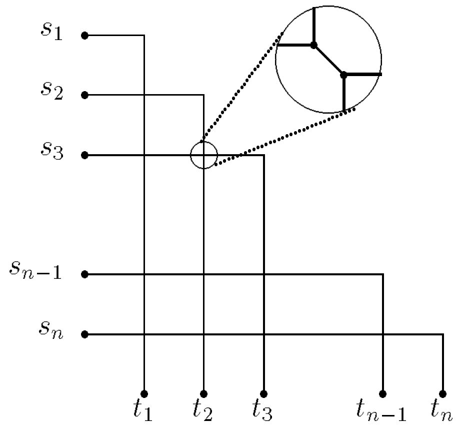
Corollary 14.12 *On trees with integer edge capacities,*

$$\max_{\text{int. flow } F} |F| \leq \min_{\text{multicut } C} |C| \leq 2 \max_{\text{int. flow } F} |F|,$$

where $|F|$ represents the value of flow function F, and $|C|$ represents the capacity of multicut C.

In Chapter 17 we will present an $O(\log k)$ factor algorithm for the minimum multicut problem in general graphs; once again, the lower bound used is an optimal fractional multicut. On the other hand, no non-trivial approximation algorithms are known for the integer multicommodity flow problem in graphs more general than trees. As shown in Example 14.13, even for planar graphs, the integrality gap of an LP analogous to (14.1) is lower bounded by $\frac{n}{2}$, where n is the number of source-sink pairs specified.

Example 14.13 Consider the following planar graph with n source-sink pairs. Every edge is of unit capacity. Any pair of paths between the i^{th} and j^{th} source-sink pairs intersect in at least one unit capacity edge. The magnified part shows how this is arranged at each intersection. So, sending one unit of any commodity blocks all other commodities. On the other hand, half a unit of each commodity can be routed simultaneously.



□

Chapter 15

Steiner forest

In this chapter, we will obtain a factor 2 approximation algorithm for the Steiner forest problem using the primal-dual schema. An interesting feature of this algorithm is the manner in which dual complementary slackness conditions will be relaxed. The Steiner forest problem generalizes the metric Steiner tree problem, for which a factor 2 algorithm was presented in Chapter 3. Recall, however, that we had postponed giving the lower bounding method behind that algorithm; we will clarify this as well.

Problem 15.1 (Steiner forest) Given a graph $G = (V, E)$, a cost function on edges $c : E \rightarrow \mathbf{Q}^+$ (not necessarily satisfying the triangle inequality), and a collection of disjoint subsets of V, $S_1, \ldots S_k$, find a minimum cost subgraph in which each pair of vertices belonging to the same set S_i is connected.

Exercise 15.2 Show that there is no loss of generality in requiring that the edge costs satisfy the triangle inequality for the above problem. (The reasoning is the same as that for the Steiner tree problem.)

Let us restate the problem; this will also help generalize it later. Define a *connectivity requirement function* r that maps unordered pairs of vertices to $\{0, 1\}$ as follows:

$$r(u, v) = \begin{cases} 1 & \text{if } u \text{ and } v \text{ belong to the same set } S_i \\ 0 & \text{otherwise} \end{cases}$$

Now, the problem is to find a minimum cost subgraph F that contains a u–v path for each pair (u, v) with $r(u, v) = 1$. The solution will be a forest, in general.

In order to give an integer programming formulation for this problem, let us define a function on all cuts in G, $f : 2^V \rightarrow \{0, 1\}$, which specifies the minimum number of edges that must cross each cut in any feasible solution.

$$f(S) = \begin{cases} 1 & \text{if } \exists\ u \in S \text{ and } v \in \overline{S} \text{ such that } r(u, v) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Let us also introduce a 0/1 variable x_e for each edge $e \in E$; x_e will be set to 1 iff e is picked in the subgraph. The integer program is:

$$\text{minimize} \quad \sum_{e \in E} c_e x_e \tag{15.1}$$

$$\begin{array}{lll} \text{subject to} & \displaystyle\sum_{e:\ e\in\delta(S)} x_e \geq f(S), & S \subseteq V \\ & x_e \in \{0,1\}, & e \in E \end{array}$$

where $\delta(S)$ denotes the set of edges crossing the cut $(S, \overline{S})$.

Exercise 15.3 Show, using the max-flow min-cut theorem, that a subgraph has all the required paths iff it does not violate any of the cut requirements. Use this fact to show that (15.1) is an integer programming formulation for the Steiner forest problem.

Following is the LP-relaxation of (15.1); once again, we have dropped the redundant conditions $x_e \leq 1$.

$$\begin{array}{lll} \text{minimize} & \displaystyle\sum_{e\in E} c_e x_e & \\ \text{subject to} & \displaystyle\sum_{e:\ e\in\delta(S)} x_e \geq f(S), & S \subseteq V \\ & x_e \geq 0, & e \in E \end{array} \tag{15.2}$$

The dual program is:

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{S\subseteq V} f(S) \cdot y_S & \\ \text{subject to} & \displaystyle\sum_{S:\ e\in\delta(S)} y_S \leq c_e & e \in E \\ & y_S \geq 0 & S \subseteq V \end{array} \tag{15.3}$$

Notice that the primal and dual programs form a covering and packing pair of LP's; see Chapter 12 for this notion. Some figurative terminology will help describe the algorithm more easily. Let us say that edge e *feels* dual y_S if $y_S > 0$ and $e \in \delta(S)$. Say that set S has been *raised* in a dual solution if $y_S > 0$. Clearly, raising S or $\overline{S}$ has the same effect. So, sometimes we will also say that we have raised the cut $(S, \overline{S})$. Further, there is no advantage in raising set S with $f(S) = 0$, since this does not contribute to the dual objective function. So, we may assume that such cuts are never raised. Say that edge e is *tight* if the total amount of dual it feels equals its cost. The dual program is trying to maximize the sum of the dual variables y_S subject to the condition that no edge feels more dual than its cost, i.e., is not *over-tight.*

Next, let us state the primal and relaxed dual complementary slackness conditions. The algorithm will pick edges integrally only. Define the *degree of set* S to be the number of picked edges crossing the cut $(S, \overline{S})$.

Primal conditions: For each $e \in E$, $x_e \neq 0 \Rightarrow \sum_{i:\ e\in\delta(S)} y_S = c_e$.
Equivalently, *every picked edge must be tight.*

Relaxed dual conditions: The following relaxation of the dual conditions would have led to a factor 2 algorithm:
For each $S \subseteq V, y_S \neq 0 \Rightarrow \sum_{e:\ e\in\delta(S)} x_e \leq 2 \cdot f(S)$, i.e., every raised cut has degree at most 2. Clearly, each cut $(S, \overline{S})$ with $f(S) = 1$ must have degree at least one, just to ensure feasibility. We do not know how to enforce this relaxation of the dual condition. Interestingly enough, we can still obtain a factor 2 algorithm – by relaxing this condition further! Raised sets will be allowed to

have high degree; however, we will ensure that *on average, raised duals have degree at most 2*. The exact definition of "on average" will be given later.

The algorithm starts with no edges picked and no cuts raised. In the spirit of the primal-dual schema, the current primal solution indicates which cuts need to be raised, and in turn, the current dual solution indicates which edge needs to be picked. Thus, the algorithm iteratively improves the feasibility of the primal, and the optimality of the dual, until a feasible primal is obtained.

Let us describe what happens in an iteration. In any iteration, the picked edges form a forest. Say that set S is *unsatisfied* if $f(S) = 1$, but there is no picked edge crossing the cut $(S, \overline{S})$. Set S is said to be *active* if it is a minimal (w.r.t. inclusion) unsatisfied set in the current iteration.

Lemma 15.4 *Set S is active iff it is a connected component in the currently picked forest and $f(S) = 1$.*

Proof : Let S be an active set. Now, S cannot cross a connected component because otherwise there will already be a picked edge in the cut $(S, \overline{S})$. So, S is a union of connected components. Since $f(S) = 1$, there is a vertex $u \in S$ and $v \in \overline{S}$ such that $r(u, v) = 1$. Let S' be the connected component containing u. Clearly, S' is also unsatisfied, and by minimality of S, $S = S'$. □

By the the characterization of active sets given in Lemma 15.4, it is easy to find all active sets in the current iteration. The dual variables of these sets are raised simultaneously until some edge goes tight. Any one of the newly tight edges is picked, and the current iteration terminates.

When a primal feasible solution is found, say F, the edge augmentation step terminates. However, F may contain redundant edges, which need to be pruned for achieving the desired approximation factor; this is illustrated in Example 15.6. Formally, edge $e \in F$ is said to be *redundant* if $F - \{e\}$ is also a feasible solution. All redundant edges can be dropped simultaneously from F. Equivalently, only non-redundant edges are retained.

This algorithm is presented below. We leave its efficient implementation as an exercise.

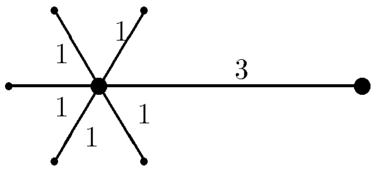
Algorithm 15.5 (Steiner forest)

1. **(Initiallization)** $F \leftarrow \emptyset$; for each $S \subseteq V$, $y_S \leftarrow 0$.
2. **(Edge augmentation)** while there exists an unsatisfied set do:

   simultaneously raise y_S for each active set S, until some edge e goes tight;

   $F \leftarrow F \cup \{e\}$.
3. **(Pruning)** return $F' = \{e \in F \mid F - \{e\}$ is primal infeasible$\}$

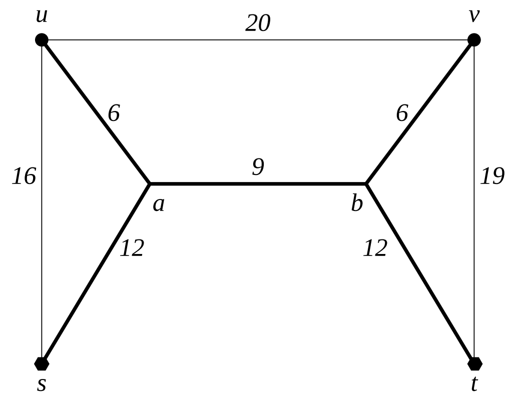
Example 15.6 Consider a star in which all edges have cost 1, except one edge whose cost is 3.



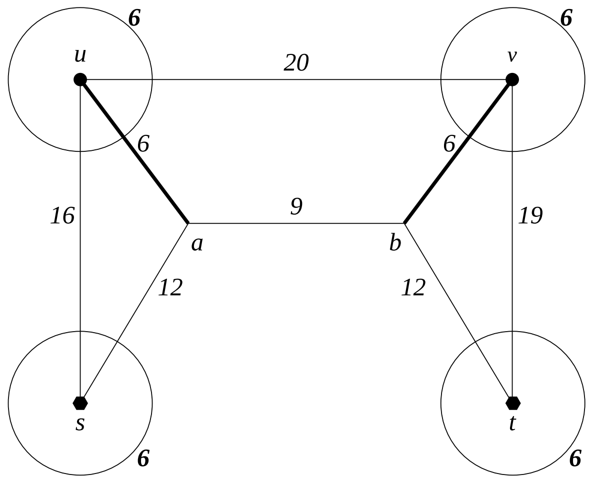
The only requirement is to connect the end vertices of the edge of cost 3. The algorithm will add to F all edges of cost 1 before adding the edge of cost 3. Clearly, at this point, F is not within twice the optimal. However, this will be corrected in the pruning step when all edges of cost 1 will be removed. □

Let us run the algorithm on a non-trivial example to illustrate its finer points.

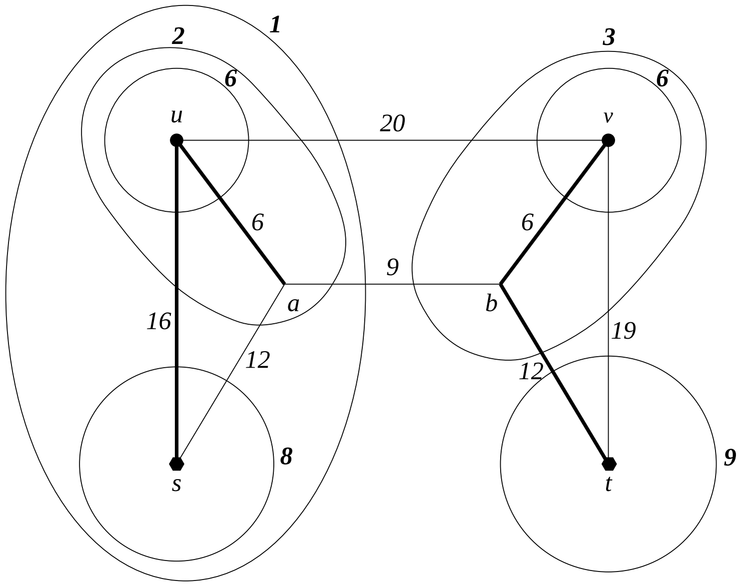
Example 15.7 Consider the following graph. Costs of edges are marked, and the only non-zero connectivity requirements are $r(u,v) = 1$ and $r(s,t) = 1$. The thick edges indicate an optimal solution, of cost 45.



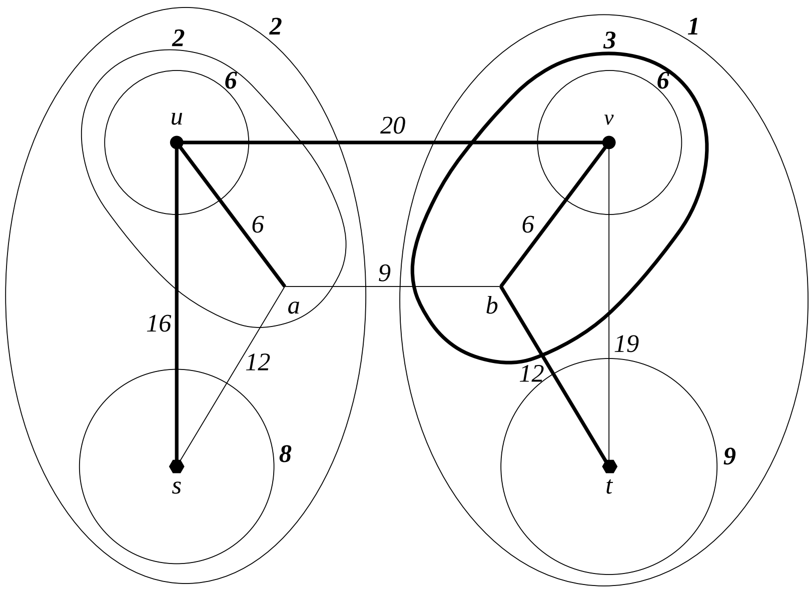
In the first iteration, the following four singleton sets are active: $\{s\}$, $\{t\}$, $\{u\}$, and $\{v\}$. When their dual variables are raised to 6 each, edges (u,a) and (v,b) go tight. One of them, say (u,a) is picked, and the iteration ends. In the second iteration, $\{u,a\}$ replaces $\{u\}$ as an active set. However, in this iteration there is no need to raise duals, since there is already a tight edge, (v,b). This edge is picked, and the iteration terminates. The primal and dual solutions at this point are shown below, with picked edges marked thick:



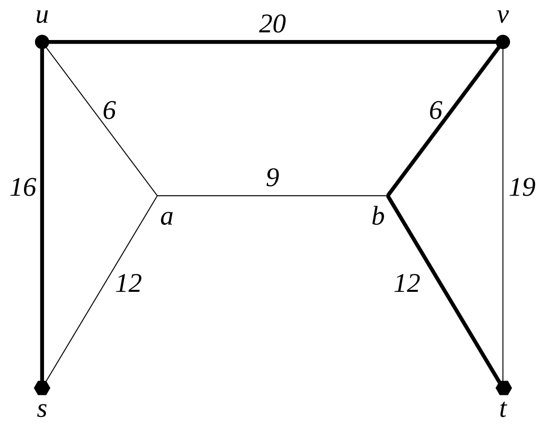
In the third iteration, $\{v,b\}$ replaces $\{v\}$ as an active set. When the active sets are raised by 2 each, edge (u,s) goes tight and is picked. In the fourth iteration, the active sets are $\{u,s,a\}$, $\{v\}$ and $\{t\}$. When they are raised by 1 each, edge (b,t) goes tight and is picked. The situation now is:



In the fifth iteration, the active sets are $\{a, s, u\}$ and $\{b, v, t\}$. When they are raised by 1 each, (u, v) goes tight, and we now have a primal feasible solution:



In the pruning step, edge (u, a) is deleted, and we obtain the following solution of cost 54:



□

In Lemma 15.8 we will show that *simultaneously* deleting all redundant edges still leaves us with a primal feasible solution, i.e., it is never the case that two edges e and f are both redundant individually, but on deletion of e, f becomes non-redundant.

Lemma 15.8 *At the end of the algorithm, F' and $\boldsymbol{y}$ are primal and dual feasible solutions respectively.*

Proof : At the end of Step 2, F satisfies all connectivity requirements. In each iteration, dual variables of connected components only are raised. Therefore, no edge running within the same component can go tight, and so F is acyclic, i.e., it is a forest. Therefore, if $r(u, v) = 1$, there is a *unique* u–v path in F. So, each edge on this path in non-redundant and is not deleted in Step 3. Hence F' is primal feasible.

Since whenever an edge goes tight the current iteration ends and active sets are redefined, no edge is overtightened. Hence $\boldsymbol{y}$ is dual feasible. □

Let $\deg_{F'}(S)$ denote the number of edges of F' crossing the cut $(S, \overline{S})$. The characterization of degrees of satisfied components established in the next lemma will be used crucially in proving the approximation guarantee of the algorithm.

Lemma 15.9 *Consider any iteration of the algorithm, and let C be a component w.r.t. the currently picked edges. If $f(C) = 0$ then $\deg_{F'}(C) \neq 1$.*

Proof : Suppose $\deg_{F'}(C) = 1$, and let e be the unique edge of F' crossing the cut $(C, \overline{C})$. Since e is non-redundant (every edge in F' is non-redundant), there is a pair of vertices, say u, v, such that $r(u, v) = 1$ and e lies on the unique u–v path in F'. Since this path crosses the cut $(C, \overline{C})$ exactly once, one of these vertices must lie in C and the other in $\overline{C}$. But since $r(u, v) = 1$, we get that $f(C) = 1$, thus leading to a contradiction. □

Lemma 15.10
$$\sum_{e \in F'} c_e \leq 2 \sum_{S \subseteq V} y_S$$

Proof : Since every picked edge is tight,

$$\sum_{e \in F'} c_e = \sum_{e \in F'} \sum_{S:\ e \in \delta(S)} y_S$$

Changing the order of summation we get:

$$\sum_{e \in F'} c_e = \sum_{S \subseteq V} \sum_{e \in \delta(S) \cap F'} y_S = \sum_{S \subseteq V} \deg_{F'}(S) \cdot y_S,$$

So, we need to show that

$$\sum_{S \subseteq V} \deg_{F'}(S) \cdot y_S \leq 2 \sum_{S \subseteq V} y_S.$$

We will prove a stronger claim: that in each iteration, the increase in the l.h.s. of this inequality is bounded by the increase in the r.h.s. Consider an iteration, and let Δ be the extent to which active sets were raised in this iteration. Then, we need to show:

$$\Delta \times \left(\sum_{S \text{ active}} \deg_{F'}(S) \right) \leq 2\Delta \times (\#\text{ of active sets})$$

Notice that the degree w.r.t. F' of any active set S is due to edges that will be picked *during or after* the current iteration. Let us rewrite this inequality as follows:

$$\frac{\sum_{S \text{ active}} \deg_{F'}(S)}{\#\text{ of active sets}} \leq 2. \tag{15.4}$$

Thus we need to show that in this iteration, the average degree of active sets with respect to F' is at most 2. The mechanics of the argument lies in the fact that in a tree, or in general in a forest, the average degree of vertices is at most 2.

Let H be a graph on vertex set V and edge set F'. Consider the set of connected components w.r.t. F at the beginning of the current iteration. In H, shrink the set of vertices of each of these components to a single node, to obtain graph H' (we will call the vertices of H' as nodes for clarity). Notice that in going from H to H', all edges picked in F before the current iteration have been shrunk. Clearly, the degree of a node in H' is equal to the degree of the corresponding set in H. Let us say that a node of H' corresponding to an active component is an *active node*; any other node will be called *inactive*. Each active node of H' has non-zero degree (since there must be an edge incident to it to satisfy its requirement), and H' is a forest. Now, remove all isolated nodes from H'. The remaining graph is a forest with average degree at most 2. By Lemma 15.9 the degree of each inactive node in this graph is at least 2, i.e., the forest has no inactive leaves. Hence, the average degree of active nodes is at most 2. □

Observe that the proof given above is essentially a charging argument: for each active node of degree greater than 2, there must be correspondingly many active nodes of degree one, i.e., leaves, in the forest. The exact manner in which the dual conditions have been relaxed must also be clear now: in each iteration, the duals being raised have average degree at most 2. Lemmas 15.8 and 15.10 give:

Theorem 15.11 *Algorithm 15.5 achieves an approximation guarantee of factor 2 for the Steiner forest problem.*

The tight example given for the metric Steiner tree problem, Example 3.3, is also a tight example for this algorithm.

Let us run Algorithm 15.5 on an instance of the metric Steiner tree problem. If the edge costs satisfy strict triangle inequality, i.e., for any three vertices u, v, w, $c(u, v) < c(u, w) + c(v, w)$, then it is easy to see that the algorithm will find a minimum spanning tree on the required vertices, i.e., it is essentially the algorithm for the metric Steiner tree problem presented in Chapter 3. Even if triangle inequality is not strictly satisfied, the cost of the solution found is the same as the cost of an MST, and if among multiple tight edges, the algorithm always prefers picking edges running between required vertices, it will find an MST. This clarifies the lower bound on which that algorithm was based.

The minimum spanning tree problem is a further special case: every pair of vertices need to be connected. Observe that when run on such an instance, Algorithm 15.5 essentially executes Kruskal's algorithm, i.e., in each iteration, it picks the cheapest edge running between two connected components. Although the primal solution found is optimal, the dual found can be as small as half the primal. For instance consider a cycle on n vertices, with all edges of cost 1. The cost of the tree is $n - 1$, but the dual found is $\frac{n}{2}$. Indeed, this is an optimal dual solution, since there is a fractional primal solution of the same value: pick each edge to the extent of half. This example places a lower bound of (essentially) 2 on the integrality gap of the primal program. In turn, the algorithm places an upper bound of 2.

Extensions and generalizations

Algorithm 15.5 actually works for a general class of problems that includes the Steiner forest problem as a special case. A function $f : 2^V \rightarrow \{0, 1\}$ is said to be *proper* if satisfies the following properties:

1. $f(V) = 0$
2. $f(S) = f(\overline{S})$
3. If A and B are two disjoint subsets of V and $f(A \cup B) = 1$ then $f(A) = 1$ or $f(B) = 1$.

Notice that function f defined for the Steiner forest problem is a proper function. Consider the integer program (15.1) with f restricted to be a proper function. This class of integer programs captures several natural problems, besides the Steiner forest problem. Consider for instance:

Problem 15.12 (Point-to-point connection) Given a graph $G = (V, E)$, a cost function on edges $c : E \rightarrow \mathbf{Q}^+$ (not necessarily satisfying the triangle inequality), and two disjoint sets of vertices, S and T, of equal cardinality, find a minimum cost subgraph that has a path connecting each vertex in S to a unique vertex in T.

Exercise 15.13 Show that the point-to-point connection problem can be formulated as an integer program using (15.1), with f being a proper function.

Exercise 15.14 Show that Algorithm 15.5 is in fact a factor 2 approximation algorithm for integer program (15.1) with f restricted to be any proper function.

Exercise 15.15 Consider the following generalization of the Steiner forest problem to higher connectivity requirements: the specified connectivity requirement function r maps pairs of vertices to $\{0, \ldots, k\}$, where k is part of the input. Assume that multiple copies of any edge can be used; each copy of edge e will cost $c(e)$. Using Algorithm 15.5 as a subroutine, give a factor $2 \cdot (\lfloor \log_2 k \rfloor + 1)$ algorithm for the problem of finding a minimum cost graph satisfying all connectivity requirements.

Chapter 16

Steiner network

The following generalization of the Steiner forest problem to higher connectivity requirements has applications in network design, and is also known as the *survivable network design problem.* In this chapter, we will give a factor 2 approximation algorithm for this problem using LP-rounding. A special case of this problem was considered in Exercise 15.15.

Problem 16.1 (Steiner network) We are given a graph $G = (V, E)$, a cost function on edges $c : E \rightarrow \mathbf{Q}^+$ (not necessarily satisfying the triangle inequality), a connectivity requirement function r mapping unordered pairs of vertices to $\mathbf{Z}^+$, and a function $u : E \rightarrow \mathbf{Z}^+ \cup \{\infty\}$ stating an upper bound on the number of copies of edge e we are allowed to use; if $u_e = \infty$, then there is no bound on the number of copies of edge e. The problem is to find a minimum cost multi-graph on vertex set V that has $r(u, v)$ edge disjoint paths for each pair of vetices $u, v \in V$. Each copy of edge e used for constructing this graph will cost $c(e)$.

In order to give an integer programming formulation for this problem, we will first define a *cut requirement function*, as we did for the metric Steiner forest problem. Function $f : 2^V \rightarrow \mathbf{Z}^+$, for $S \subseteq V$ is defined to be the largest connectivity requirement separated by the cut $(S, \overline{S})$, i.e., $f(S) = \max\{r(u, v) | u \in S \text{ and } v \in \overline{S}\}$.

$$\begin{aligned}
\text{minimize} \quad & \sum_{e \in E} c_e x_e && && (16.1)\\
\text{subject to} \quad & \sum_{e:\ e \in \delta(S)} x_e \geq f(S), && S \subseteq V \\
& x_e \in \mathbf{Z}^+, && e \in E \text{ and } u_e = \infty \\
& x_e \in \{0, 1, \ldots, u_e\}, && e \in E \text{ and } u_e \neq \infty
\end{aligned}$$

The LP-relaxation is:

$$\begin{aligned}
\text{minimize} \quad & \sum_{e \in E} c_e x_e && && (16.2)\\
\text{subject to} \quad & \sum_{e:\ e \in \delta(S)} x_e \geq f(S), && S \subseteq V \\
& x_e \geq 0, && e \in E \text{ and } u_e = \infty \\
& u_e \geq x_e \geq 0, && e \in E \text{ and } u_e \neq \infty
\end{aligned}$$

As shown in Chapter **??**, certain NP-hard problems, such a vertex cover and node multiway cut admit LP-relaxations having the remarkable property that they always have a half-integral optimal solution. Rounding up all halves to 1 in such a solution leads to a factor 2 approximation algorithm. Does relaxation (16.2) have this property? The following lemma shows that the answer is "No".

Lemma 16.2 *Consider the Petersen graph with a connectivity requirement of one between each pair of vertices and with each edge of unit cost. Relaxation (16.2) does not have a half-integral optimal solution for this instance.*

Proof : Consider the fractional solution $x_e = 1/3$ for each edge e. Since the Petersen graph is 3-edge connected (in fact, it is 3-vertex connected as well), this is a feasible solution. The cost of this solution is 5. In any feasible solution, the sum of edge variables incident at any vertex must be at least one, to allow connectivity to other vertices. Therefore any feasible solution must have cost at least 5 (since the Petersen graph has 10 vertices). Hence, the solution given above is in fact optimal.

Any solution with $x_e = 1$ for some edge e must have cost exceeding 5, since additional edges are required to connect the endpoints of e to the rest of the graph. Therefore, any half integral solution of cost 5 would have to pick, to the extent of half each, the edges of a Hamiltonian cycle. Since the Petersen graph has no Hamiltonian cycles, there is no half integral optimal solution. □

Let us say that an *extreme solution*, also called a vertex solution or a basic feasible solution, for an LP is a feasible solution that cannot be written as the convex combination of two feasible solutions. It turns out that the solution $x_e = 1/3$, for each edge e, is not an extreme solution. An extreme optimal solution is shown in Figure 16.1; thick edges are picked to the extent of 1/2, thin edges to the extent of 1/4, and the missing edge is not picked. The isomorphism group of the Petersen graph is edge-transitive, and so there are 15 related extreme solutions; the solution $x_e = 1/3$ for each edge e is the average of these.

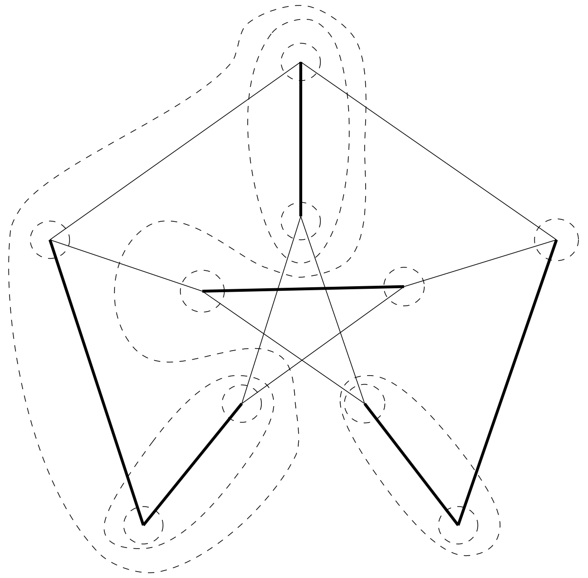


Figure 16.1:

Notice that although the extreme solution is not half-integral, it picks some edges to the extent of half. We will show below that in fact this is a property of any extreme solution to LP (16.2). We will obtain a factor 2 algorithm by rounding up these edges and iterating. Let H be the set of edges picked by the algorithm at some point. Then, the residual requirement of cut $(S, \overline{S})$ is $f'(S) = f(S) - |\delta_H(S)|$, where $\delta_H(S)$ represents the set of edges of H crossing the cut $(S, \overline{S})$. In general, the *residual cut requirement function*, f', may not correspond to the cut requirement function for any set of connectivity requirements. We will need the following definitions to characterize it:

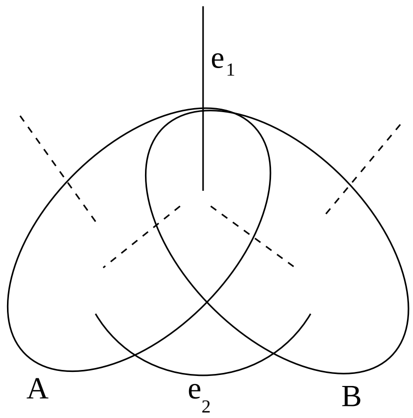
Definition 16.3 Function $f : 2^V \rightarrow \mathbf{Z}^+$ is said to be *submodular* if $f(V) = 0$, and for every two sets $A, B \subseteq V$, the following two conditions hold:

- $f(A) + f(B) \geq f(A - B) + f(B - A)$.
- $f(A) + f(B) \geq f(A \cap B) + f(A \cup B)$.

Two subsets of V, A and B, are said to *cross* if each of the sets $A - B$, $B - A$ and $A \cap B$ is non-empty. If A and B don't cross then either they are disjoint or one of these sets is contained in the other.

Lemma 16.4 *For any graph G on vertex set V, the function $|\delta_G(.)|$ is submodular.*

Proof : If sets A and B do not cross, then the two conditions given in the definition of submodular functions hold trivially. Otherwise, edges having one endpoint in $A \cap B$ and the other in $\overline{A \cup B}$ (edge e_1 in the figure below) contribute to $\delta(A)$ and $\delta(B)$ but not to $\delta(A - B)$ or $\delta(B - A)$. Similarly, edge e_2 below does not contribute to $\delta(A \cap B)$ or to $\delta(A \cup B)$. The remaining edges contribute equally to both sides of both conditions. $\square$



Definition 16.5 Function $f : 2^V \rightarrow \mathbf{Z}$ is said to be *weakly supermodular* if $f(V) = 0$, and for every two sets $A, B \subseteq V$, at least one of the following conditions holds:

- $f(A) + f(B) \leq f(A - B) + f(B - A)$
- $f(A) + f(B) \leq f(A \cap B) + f(A \cup B)$

It is easy to check that the original cut requirement function is weakly supermodular; by Lemma 16.6, so is the residual cut requirement function.

Lemma 16.6 *Let H be a subgraph of G. If $f : 2^{V(G)} \rightarrow \mathbf{Z}^+$ is a weakly supermodular function, then so is the residual cut requirement function f'.*

Proof : Suppose $f(A) + f(B) \leq f(A - B) + f(B - A)$; the proof of the other case is similar. By Lemma 16.4, $|\delta_H(A)| + |\delta_H(B)| \geq |\delta_H(A - B)| + |\delta_H(B - A)|$. Subtracting, we get $f'(A) + f'(B) \leq f'(A - B) + f'(B - A)$. $\square$

We can now state the central polyhedral fact needed for the factor 2 algorithm in its full generality.

Theorem 16.7 *For any weakly supermodular function f, any extreme solution, $\boldsymbol{x}$, to LP (16.2) must pick some edge to the extent of at least a half, i.e., $x_e \geq 1/2$ for at least one edge e.*

The algorithm

Based on Theorem 16.7 the factor 2 algorithm is:

Algorithm 16.8 (Steiner network)

1. **Initialization:** $H \leftarrow \emptyset$: $f' \leftarrow f$.
2. While $f' \not\equiv \mathbf{0}$, do:

   Find an extreme optimal solution, $\boldsymbol{x}$, to LP (16.2) with cut requirements given by f'.

   For each edge e such that $x_e \geq 1/2$, include $\lceil x_e \rceil$ copies of e in H,
   and decrement u_e by this amount.

   Update f': for $S \subseteq V$, $f'(S) \leftarrow f(S) - |\delta_H(S)|$.
3. Output H.

The algorithm presented above achieves an approximation guarantee of factor 2 for an arbitrary weakly supermodular function f. Establishing a polynomial running time involves showing that an extreme optimal solution to LP (16.2) can be found efficiently. We do not know how to do this for an arbitrary weakly supermodular function f. However, if f is the original cut requirement function for some connectivity requirements, then a polynomial time implementation is possible, by showing the existence of a polynomial time separation oracle for each iteration.

For the first iteration, a separation oracle follows from a max-flow subroutine. Given a solution $\boldsymbol{x}$, construct a graph on vertex set V with capacity x_e for each edge e. Then, for each pair of vertices $u, v \in V$, check if this graph admits a flow of at least $r(u,v)$ from u to v. If not, we will get a violated cut, i.e., a cut $(S, \overline{S})$ such that $\delta_{\boldsymbol{x}}(S) < f(S)$, where

$$\delta_{\boldsymbol{x}}(S) = \sum_{e:\ e \in \delta(S)} x_e .$$

Let f' be the cut requirement function of a subsequent iteration. Given a solution to LP (16.2) for this function, say $\boldsymbol{x}'$, define $\boldsymbol{x}$ as follows: for each edge e, $x_e = x'_e + e_H$, where e_H is the number of copies of edge e in H. The following lemma shows that a separation oracle for the original function f leads to a separation oracle for f'. Furthermore, this lemma also shows that there is no need to update f' explictly after each iteration.

Lemma 16.9 *A cut $(S, \overline{S})$ is violated by solution $\boldsymbol{x}'$ under cut requirement function f' iff it is violated by solution $\boldsymbol{x}$ under cut requirement function f.*

Proof : Notice that $\delta_{\boldsymbol{x}}(S) = \delta_{\boldsymbol{x}'}(S) + |\delta_H(S)|$. Since $f(S) = f'(S) + |\delta_H(S)|$, $\delta_{\boldsymbol{x}}(S) \geq f(S)$ iff $\delta_{\boldsymbol{x}'}(S) \geq f'(S)$. □

Lemma 16 implies that solution $\boldsymbol{x}'$ is feasible for the cut requirement function f' iff solution $\boldsymbol{x}$ is feasible for f. Assuming Theorem 16.7, whose proof we will provide below, let us show that Algorithm 16.8 achieves an approximation guarantee of 2.

Theorem 16.10 *Algorithm 16.8 achieves an approximation guarantee of 2 for the Steiner network problem.*

Proof : By induction on the number of iterations executed by the algorithm when run with a weakly supermodular cut requirement function f, we will prove that the cost of the integral solution obtained is within a factor of two of the cost of the optimal fractional solution. Since the latter is a lower bound on the cost of the optimal integral solution, the claim follows.

For the base case, if f requires one iteration, the claim follows, since the algorithm rounds up only edges e with $x_e \geq 1/2$.

For the induction step, assume that $\boldsymbol{x}$ is the extreme optimal solution obtained in the first iteration. Obtain $\hat{\boldsymbol{x}}$ from $\boldsymbol{x}$ by zeroing out components that are strictly smaller than 1/2. By Theorem 16.7, $\hat{\boldsymbol{x}} \neq 0$. Let H be the set of edges picked in the first iteration. Since H is obtained by rounding up non-zero components of $\hat{\boldsymbol{x}}$, and each of these components is $\geq 1/2$, $\text{cost}(H) \leq 2\text{cost}(\hat{\boldsymbol{x}})$.

Let f' be the residual requirement function after the first iteration, and H' be the set of edges picked in subsequent iterations for satisfying f'. The key observation is that $\boldsymbol{x} - \hat{\boldsymbol{x}}$ is a feasible solution for f', and so by the induction hypothesis, $\text{cost}(H') \leq 2\text{cost}(\boldsymbol{x} - \hat{\boldsymbol{x}})$. Let us denote by $H + H'$ the edges of H together with those of H'. Clearly, $H + H'$ satisfies f. Now,

$$\text{cost}(H + H') \leq \text{cost}(H) + \text{cost}(H') \leq 2\text{cost}(\hat{\boldsymbol{x}}) + 2\text{cost}(\boldsymbol{x} - \hat{\boldsymbol{x}}) \leq 2\text{cost}(\boldsymbol{x}).$$

□

Corollary 16.11 *The integrality gap of LP (16.2) is bounded by 2.*

Example 16.12 The tight example given for the metric Steiner tree problem, Example 3.3, is also a tight example for this algorithm. Observe that after including a subset of edges of the cycle, an extreme optimal solution to the resulting problem picks the remaining edges of the cycle to the extent of half each. So, the algorithm finds a solution of cost $(2-\epsilon)(n-1)$, whereas the cost of the optimal solution is n. □

Characterizing extreme solutions

From polyhedral combinatorics we know that a feasible solution for a set of linear inequalities in $\mathbf{R}^m$ is an extreme solution iff it satisfies m linearly independent inequalities with equality. Extreme solutions of LP (16.2) satisfy an additional property which leads to a proof of Theorem 16.7.

We will assume that the cut requirement function f in LP (16.2) is an arbitrary weakly supermodular function. Given a solution $\boldsymbol{x}$ to this LP, we will say that an inequality is *tight* if it holds with equality. If this inequality corresponds to the cut requirement of a set S, then we will say that *set S is tight*. Let us make some simplifying assumptions. If $x_e = 0$ for some edge e, this edge can be removed from the graph, and if $x_e \geq 1$, $\lfloor x_e \rfloor$ copies of edge e can be picked and the cut requirement function be updated accordingly. So, we may assume without loss of generality that an extreme solution $\boldsymbol{x}$ satisfies $0 < x_e < 1$, for each edge e in graph G. Therefore, each tight inequality corresponds to a tight set. Let the number of edges in G be m.

We will say that a collection, $\mathcal{L}$, of subsets of V forms a *laminar family* if no two sets in this collection cross. The inequality corresponding to a set S defines a vector in $\mathbf{R}^m$: the vector has a 1 corresponding to each edge $e \in \delta_G(S)$, and 0 otherwise. We will call this the *incidence vector* of set S, and will denote it by $\mathcal{A}_S$.

Theorem 16.13 *Corresponding to any extreme solution to LP (16.2) there is a collection of m tight sets such that*

- *their incidence vectors are linearly independent, and*
- *collection of sets forms a laminar family.*

Example 16.14 The extreme solution for the Peterson graph given in Figure 16.1 assigns non-zero values to 14 of the 15 edges. By Theorem 16.13, there should be 14 tight sets whose incidence vectors are linearly independent. These are marked in Figure 16.1. □

Fix an extreme solution, $\boldsymbol{x}$, to LP (16.2). Let $\mathcal{L}$ be a laminar family of tight sets whose incidence vectors are linearly independent. Denote by $\text{span}(\mathcal{L})$ the vector space generated by the set of vectors $\{\mathcal{A}_S | S \in \mathcal{L}\}$. Since $\boldsymbol{x}$ is an extreme solution, the span of the collection of all tight sets is m. We will show that if $\text{span}(\mathcal{L}) < m$, then there is a tight set S whose addition to $\mathcal{L}$ does not violate laminarity and also increases the span. Continuing in this manner, we will obtain m tight sets as required in Theorem 16.13.

We begin by studying properties of crossing tight sets.

Lemma 16.15 *Let A and B be two crossing tight sets. Then, one of the following must hold:*

- *$A - B$ and $B - A$ are both tight and $\mathcal{A}_A + \mathcal{A}_B = \mathcal{A}_{A-B} + \mathcal{A}_{B-A}$*
- *$A \cup B$ and $A \cap B$ are both tight and $\mathcal{A}_A + \mathcal{A}_B = \mathcal{A}_{A\cup B} + \mathcal{A}_{A\cap B}$.*

Proof : Since f is weakly supermodular, either $f(A) + f(B) \leq f(A - B) + f(B - A)$ or $f(A) + f(B) \leq f(A \cup B) + f(A \cap B)$. Let us assume the former holds; the proof for the latter is similar. Since A and B are tight, we have

$$\delta_{\boldsymbol{x}}(A) + \delta_{\boldsymbol{x}}(B) = f(A) + f(B).$$

Since $A - B$ and $B - A$ are not violated,

$$\delta_{\boldsymbol{x}}(A - B) + \delta_{\boldsymbol{x}}(B - A) \geq f(A - B) + f(B - A).$$

Therefore,

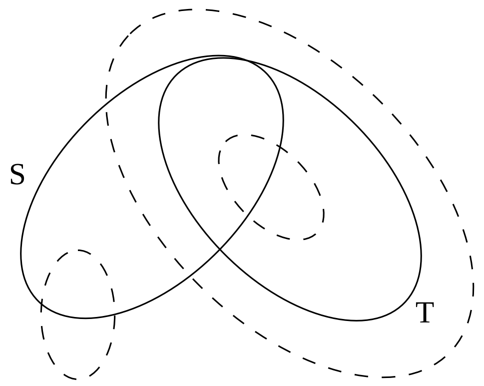
$$\delta_{\boldsymbol{x}}(A) + \delta_{\boldsymbol{x}}(B) \leq \delta_{\boldsymbol{x}}(A - B) + \delta_{\boldsymbol{x}}(B - A).$$

As argued in Lemma 16.4 (which established the submodularity of function $|\delta_G(.)|$), edges having one endpoint in $A \cup B$ and the other in $A \cap B$ can contribute only to the l.h.s. of this inequality. The rest of the edges must contribute equally to both sides. So, this inequality must be satisfied with equality. Furthermore, since $x_e > 0$ for each edge e, G cannot have any edge having one endpoint in $A \cup B$ and the other in $A \cap B$. Therefore, $\mathcal{A}_A + \mathcal{A}_B = \mathcal{A}_{A-B} + \mathcal{A}_{B-A}$. □

For any set $S \subseteq V$, define its *crossing number* to be the number of sets of $\mathcal{L}$ that S crosses.

Lemma 16.16 *Let S be a set that crosses set $T \in \mathcal{L}$. Then, each of the sets $S - T, T - S, S \cup T$ and $S \cap T$ has a smaller crossing number than S.*

Proof : The figure below illustrates the three ways in which a set $T' \in \mathcal{L}$ can cross one of these four sets without crossing T itself (T' is shown dotted). In all cases, T' crosses S as well. In addition, T crosses S but not any of the four sets. □



Lemma 16.17 *Let S be a tight set such that $\mathcal{A}_S \notin \mathrm{span}(\mathcal{L})$ and S crosses some set in $\mathcal{L}$. Then, there is a tight set S' having a smaller crossing number than S, and such that $\mathcal{A}_{S'} \notin \mathrm{span}(\mathcal{L})$.*

Proof : Let S cross $T \in \mathcal{L}$. Suppose the first possibility established in Lemma 16.15 holds; the proof of the second possibility is similar. Then, $S - T$ and $T - S$ are both tight sets and $\mathcal{A}_S + \mathcal{A}_T = \mathcal{A}_{S-T} + \mathcal{A}_{T-S}$. This linear dependence implies that $\mathcal{A}_{S-T}$ and $\mathcal{A}_{T-S}$ cannot both be in $\mathrm{span}(\mathcal{L})$, since otherwise $\mathcal{A}_S \in \mathrm{span}(\mathcal{L})$. By Lemma 16.16, $S-T$ and $T-S$ both have a smaller crossing number than S. The lemma follows. □

Corollary 16.18 *If $\mathrm{span}(\mathcal{L}) \neq \mathbf{R}^m$, then there is a tight set S such that $\mathcal{A}_S \notin \mathrm{span}(\mathcal{L})$ and $\mathcal{L} \cup \{S\}$ is a laminar family.*

By Corollary 16.18, if $\mathcal{L}$ is a maximal laminar family of tight sets with linearly independent incidence vectors, then $|\mathcal{L}| = m$. This establishes Theorem 16.13.

A counting argument

The characterization of extreme solutions given in Theorem 16.13 will yield Theorem 16.7 via a counting argument. Let $\boldsymbol{x}$ be an extreme solution and $\mathcal{L}$ be the collection of tight sets established in Theorem 16.13. The number of sets in $\mathcal{L}$ equals the number of edges in G, i.e., m. The proof is by contradiction. Suppose that for each edge $e, x_e < 1/2$. Then, we will show that G has more than m edges.

Since $\mathcal{L}$ is a laminar family, it can be viewed as a forest of trees if its elements are ordered by inclusion. Let us make this precise. For $S \in \mathcal{L}$, if S is not contained in any other set of $\mathcal{L}$, then we will say that S is a *root set.* If S is not a root set, we will say that T is the *parent* of S if T is a minimal set in $\mathcal{L}$ containing S; by laminarity of $\mathcal{L}$, T is unique. Further, S will be called a *child* of T. Let the relation *descendent* be the reflexive transitive closure of the relation “child”. Sets that have no children will be called *leaves.* In this manner, $\mathcal{L}$ can be partitioned into a forest of trees, each rooted at a root set. For any set S, by the *subtree rooted at* S we mean the set of all descendents of S.

Edge e is *incident* at set S if $e \in \delta_G(S)$. The *degree* of S is defined to be $|\delta_G(S)|$. Set S *owns* endpoint v of edge $e = (u, v)$ if S is the smallest set of $\mathcal{L}$ containing v. The subtree rooted at set S *owns* endpoint v of edge $e = (u, v)$ if some descendent of S owns v.

Since G has m edges, it has $2m$ endpoints. Under the assumption that $\forall e, x_e < 1/2$, we will prove that the for any set S, the endpoints owned by the subtree rooted at S can be redistributed in such a way that S gets at least 3 endpoints, and each of its proper descendents gets 2 endpoints. Carrying out this procedure for each of the root sets of the forest, we get that the total number of endpoints in the graph must exceed $2m$, leading to a contradiction.

We have assumed that $\forall e : 0 < x_e < 1/2$. For edge e, define $y_e = 1/2 - x_e$, the *halves complement* of e. Clearly, $0 < y_e < 1/2$. For $S \in \mathcal{L}$ define its *co-requirement* to be

$$\text{coreq}(S) = \sum_{e \in \delta(S)} y_e = \frac{1}{2}|\delta_G(S)| - f(S).$$

Clearly, $0 < \text{coreq}(S) < |\delta_G(S)|/2$. Furthermore, since $|\delta_G(S)|$ and $f(S)$ are both integral, $\text{coreq}(S)$ is half integral. Let us say that *coreq*(S) is *semi-integral* if it is not integral, i.e., if $\text{coreq}(S) \in \{1/2, 3/2, 5/2, \ldots\}$. Since $f(S)$ is integral, *coreq*(S) is semi-integral iff $|\delta_G(S)|$ is odd.

Sets having co-requirement of $1/2$ play a special role in this argument. The following lemma will be useful in establishing that certain sets have this co-requirement.

Lemma 16.19 *Suppose S has α children and owns β endpoints, where $\alpha + \beta = 3$. Furthermore, each child of S, if any, has a co-requirement of* $1/2$. *Then,* $\text{coreq}(S) = 1/2$.

Proof : Since each child of S has co-requirement of $1/2$, it has odd degree. Using this and the fact that $\alpha + \beta = 3$, one can show that S must have odd degree (see Exercise 16.23). Therefore the co-requirement of S is semi-integral. Next, we show that $\text{coreq}(S)$ is strictly smaller than $< 3/2$, thereby proving the lemma. Clearly,

$$\text{coreq}(S) = \sum_{e \in \delta(S)} y_e \leq \sum_{S'} \text{coreq}(S') + \sum_{e} y_e,$$

where the first sum is over all children S' of S, and the second sum is over all edges e having an endpoint in S. Since y_e is strictly smaller than $1/2$, if $\beta > 0$, then $\text{coreq}(S) < 3/2$. If $\beta = 0$, all edges incident at the children of S cannot also be incident at S, since otherwise the incidence vectors of these four sets will be linearly dependent. Therefore,

$$\text{coreq}(S) < \sum_{S'} \text{coreq}(S') = 3/2.$$

□

The next two lemmas place lower bounds on the number of endpoints owned by certain sets.

Lemma 16.20 *If set S has only one child, then it must own at least two endpoints.*

Proof : Let S' be the child of S. If S has no end point incident at it, the set of edges incident at S and S' must be the same. But then $\mathcal{A}_S = \mathcal{A}_{S'}$, leading to a contradiction. S cannot own exactly one endpoint, because then $\delta_{\boldsymbol{x}}(S)$ and $\delta_{\boldsymbol{x}}(S')$ will differ by a fraction, contradicting the fact that both these sets are tight and have integral requirements. The lemma follows. $\square$

Lemma 16.21 *If set S has two children, one of which has co-requirement of $1/2$, then it must own at least one endpoint.*

Proof : Let S' and S'' be the two children of S, with $\text{coreq}(S') = 1/2$. Suppose S does not own any endpoints. Since the three vectors $\mathcal{A}_S, \mathcal{A}_{S'}$ and $\mathcal{A}_{S''}$ are linearly independent, the set of edges incident at S' cannot all be incident at S, or all be incident at S''. Let a denote the sum of y_e's of all edges incident at S' and S, and let b denote the sum of y_e's of all edges incident at S' and S''. Thus, $a > 0$, $b > 0$, and $a + b = \text{coreq}(S) = 1/2$.

Since S' has semi-integral co-requirement, it must have odd degree. Therefore, the degrees of S and S'' have different parities, and so these two sets have different co-requirements. Furthermore, $\text{coreq}(S) = \text{coreq}(S'') + a - b$. Therefore, $\text{coreq}(S) - \text{coreq}(S'') = a - b$. But $-1/2 < a - b < 1/2$. Therefore, S and S'' must have the same co-requirement, leading to a contradiction. $\square$

Lemma 16.22 *Consider a tree T rooted at set S. Under the assumption that $\forall e, x_e < 1/2$, the endpoints owned by T can be redistributed in such a way that the S gets at least 3 endpoints, and each of its proper descendents gets 2 endpoints. Furthermore, if coreq$(S) \neq 1/2$, then S must get at least 4 endpoints.*

Proof : The proof is by induction on the height of tree T. For base case, consider a leaf set S. S must have degree at least 3, because otherwise an edge e incident at it will have $x_e \geq 1/2$. If it has degree exactly 3, $\text{coreq}(S)$ is semi-integral. Further, since $\text{coreq}(S) < |\delta_G(S)|/2 = 3/2$, the co-requirement of S is $1/2$. Since S is a leaf, it owns an endpoint of each edge incident at it. Therefore, S has the required number of endpoints.

Let us say that a set has a *surplus* of 1 if 3 endpoints have been assigned to it and a surplus of 2 if 4 endpoints have been assigned to it. For the induction step, consider a non-leaf set S. We will prove that by moving the surplus of the children of S and considering the endpoints owned by S itself, we can assign the required number of endpoints to S. There are four cases:

1. If S has 4 or more children, we can assign the surplus of each child to S, thus assigning at least 4 endpoints to S.

2. Suppose S has 3 children. If at least one of them has a surplus of 2, or if S owns an endpoint, we can assign 4 endpoints to S. Otherwise, each child must have a co-requirement of half, and by Lemma 16.19, $\text{coreq}(S) = 1/2$ as well. So, assigning S the surplus of its children suffices.

3. Suppose S has two children. If each has a surplus of 2, we can assign 4 endpoints to S. If one of them has surplus 1, then by Lemma 16.21, S must own at least one endpoint. If each child has a surplus of 1 and S owns exactly one endpoint, then we can assign 3 endpoints to S, and this suffices by Lemma 16.19. Otherwise, we can assign 4 endpoints to S.

4. If S has one child, say S', then by Lemma 16.20, S owns at least 2 endpoints. If S owns exactly 2 endpoints and S' has surplus of exactly 1, then we can assign 3 endpoints to S; by Lemma 16.19, $\text{coreq}(S) = 1/2$, so this suffices. In all other cases, we can assign 4 endpoints to S.

□

Exercise 16.23 Prove that set S in Lemma 16.19 must have odd degree. (Consider the following possibilities: S owns endpoint v of edge (u, v) that is incident at S, S owns endpoint v of edge (u, v) that is incident at a child of S, and an edge is incident at two children of S.)

Exercise 16.24 Prove that there must be a set in $\mathcal{L}$ that has degree at most 3, and so some edge must have $x_e \geq 1/3$. The counting argument required for this is much simpler. Notice that this fact leads to a factor 3 algorithm. (The counting argument requires the use of Lemma 16.20.)

Chapter 17

Multicut in general graphs

The importance of min-max relations to combinatorial optimization was mentioned in Chapter 10. Perhaps the most useful of these has been the celebrated max-flow min-cut theorem – indeed, much of flow theory, and the theory of cuts in graphs, has been built around this theorem. It is not surprising, therefore, that a concerted effort was made to obtain generalizations of this theorem to the case of multiple commodities.

There are two different natural generalizations of the maximum flow problem to multiple commodities. In the first generalization, the objective is to maximize the sum of the commodities routed, subject to flow conservation and capacity constraints. In the second generalization, we are also specified a demand D_i for each commodity i, and the objective is to maximize f, called throughput, such that for each i, $f \cdot D_i$ amount of commodity i can be routed simultaneously. We will call these *sum multicommodity flow* and *demands multicommodity flow* problems respectively. Clearly, for the case of a single commodity, both problems are same as the maximum flow problem.

Each of these generalizations is associated with a fundamental **NP**-hard cut problem; the first with the minimum multicut problem, Problem 14.1, and the second with the sparsest cut problem, Problem 18.2. In each case, an approximation algorithm for the cut problem gives, as a corollary, an approximate max-flow min-cut theorem. In this chapter, we will study the first generalization; the second is presented in Chapter 18. We will obtain an $O(\log k)$ factor approximation algorithm for the minimum multicut problem, where k is the number of commodities; a factor 2 algorithm for the special case of trees was presented in Chapter 14.

Problem 17.1 (Sum multicommodity flow) Let $G = (V, E)$ be an undirected graph with non-negative capacity c_e for each edge $e \in E$. Let $\{(s_1, t_1), \ldots, (s_k, t_k)\}$ be a specified set of pairs of vertices, where each pair is distinct, but vertices in different pairs are not required to be distinct. A separate commodity is defined for each (s_i, t_i) pair; for convenience, we will think of s_i as the source and t_i as the sink of this commodity. The objective is to maximize the sum of the commodities routed. Each commodity must satisfy flow conservation at each vertex other than its own source and sink. Also, the sum of flows routed through an edge, in both directions combined, should not exceed the capacity of this edge.

Let us first give a linear programming formulation for this problem. For each commodity i, let P_i denote the set of all paths from s_i to t_i in G, and let $P = \bigcup_{i=1}^{k} P_i$. The LP will have a variable f_p for each $p \in P$, which will denote the flow along path p; the end points of this path uniquely specify the commodity that flows on this path. The objective is to maximize the sum of flows routed on these paths, subject to edge capacity constraints; flow conservation constraints are automatically satisfied in this formulation. The program has exponentially many variables; however, that is not a concern since we will use it primarily to obtain a clean formulation of the dual program.

$$
\begin{aligned}
\text{maximize} \quad & \sum_{p \in P} f_p && && (17.1)\\
\text{subject to} \quad & \sum_{p: e \in p} f_p \le c_e, && e \in E \\
& f_p \ge 0, && p \in P
\end{aligned}
$$

Let us obtain the dual of this program. For this, let d_e be the dual variable associated with edge e; we will interpret these variables as distance labels of edges.

$$
\begin{aligned}
\text{minimize} \quad & \sum_{e \in E} c_e d_e && && (17.2)\\
& \sum_{e \in p} d_e \ge 1, && p \in P \\
& d_e \ge 0, && e \in E
\end{aligned}
$$

The dual program is trying to find a distance label assignment to edges so that on each path $p \in P$, the distance labels of edges add up to at least 1. Equivalently, a distance label assignment is feasible iff for each commodity i, the shortest path from s_i to t_i has length at least 1.

Notice that the programs (14.1) and (14.1) are special cases of the two programs presented above, for the restriction that G is a tree.

Exercise 17.2 By defining for each edge e and commodity i a flow variable $f_{e,i}$, give an LP that is equivalent to LP (17.1) and has polynomially many variables. Obtain the dual of this program, and show that it is equivalent to LP (17.2); however, unlike LP (17.2), it has only polynomially many constraints.

The following remarks made in Chapter 14 hold for the two programs presented above as well: An optimal integral solution to LP (17.2) is a minimum multicut, and an optimal fractional solution can be viewed as a minimum fractional multicut. By the LP-Duality Theorem, minimum fractional multicut equals maximum multicommodity flow, and as shown in Example 14.3, it may be strictly smaller than minimum integral multicut.

This naturally raises the question whether the ratio of minimum multicut and maximum multicommodity flow is bounded. Equivalently, is the integrality gap of LP (17.2) bounded? In the next section, we present an algorithm for finding a multicut within an $O(\log k)$ factor of the maximum flow, thereby showing that the gap is bounded by $O(\log k)$.

The algorithm

First notice that the dual program (17.2) can be solved in polynomial time using the ellipsoid algorithm, since there is a simple way of obtaining a separation oracle for it: Simply compute the length of a minimum s_i–t_i path, for each commodity i, w.r.t. the current distance labels. If all these lengths are ≥ 1, we have a feasible solution. Otherwise, the shortest such path provides a violated inequality. Alternatively, the LP obtained in Exercise 17.2 can be solved in polynomial time. Let d_e be the distance label computed for each edge e, and let $F = \sum_{e \in E} c_e d_e$.

Our goal is to pick a set of edges of small capacity, compared to F, that is a multicut. Let D be the set of edges with positive distance labels, i.e., $D = \{e \mid d_e > 0\}$. Clearly, D is a multicut; however, in general, its capacity may be very large compared to F. How do we pick a small capacity subset of D that is still a multicut? Since the optimal fractional multicut is the most cost-effective way of disconnecting all source–sink pairs, edges with large distance labels are more important than those with small distance labels for this purpose. The algorithm described below indirectly gives preference to edges with large distance labels.

Exercise 17.3 Intuitively, our goal in picking a multicut is picking edges that are bottlenecks for multicommodity flow. In this sense, D is a very good starting point: Prove that D is precisely the set of edges that are saturated in *every* maximum multicommodity flow. (Hint: Use complementary slackness conditions.)

The algorithm will work on graph $G = (V, E)$ with edge lengths given by d_e. Let $\mathrm{dist}(u, v)$ denote the length of the shortest path from u to v in this graph. For a set of vertices $S \subset V$, $\delta(S)$ denotes the set of edges in the cut $(S, \overline{S})$, $c(\delta(S))$ denotes the capacity of this cut, and $\mathrm{wt}(S)$ denotes the *weight* of set S, which is *roughly* $\sum c_e d_e$, where the sum is over all edges that have at least one end point in set S (a more precise definition is given below).

The algorithm will find disjoint sets of vertices, $S_1, \ldots, S_l$, in G, called *regions*, such that:

- No region contains any source-sink pair, and for each i, either s_i or t_i is in one of the regions.
- For each region S_i, $c(\delta(S_i)) \le \epsilon\, \mathrm{wt}(S_i)$, where ϵ is a parameter that will be defined below.

By the first condition, the union of the cuts of these regions, i.e., $M = \delta(S_1) \cup \delta(S_2) \cup \ldots \cup \delta(S_l)$, is a multicut, and by the second condition, its capacity $c(M) \le 2\epsilon F$. (The factor of 2 appears because an edge can contribute to the weight of at most two of the regions; later, we will argue away this factor).

Growing a region: the continuous process

In this section, we will determine parameter ϵ so that the two conditions stated above can be made to hold; of course, we want to find the smallest ϵ that suffices. For this purpose, it will be convenient to consider a continuous region growing process; the algorithm will use a discretized version of this process.

Each region is found by growing a set, starting from one vertex, the source or sink of a pair; for convenience, we will always start with a source. This will be called the *root* of the region. Let us see how to find region S with root s_1. For each radius r, define $S(r)$ to be the set of vertices at a distance at most r from s_1, i.e., $S(r) = \{v \mid \mathrm{dist}(s_1, v) \le r\}$. $S(0) = \{s_1\}$, and as r increases continuously from 0, at discrete points, $S(r)$ grows by adding vertices in increasing order of their distance to s_1.

Lemma 17.4 *If the region growing process is terminated before the radius becomes $\frac{1}{2}$, then the set S found contains no source-sink pair.*

Proof : The distance between any pair of vertices in $S(r)$ is $\le 2r$. Since for each commodity i, $\mathrm{dist}(s_i, t_i) \ge 1$, the lemma follows. □

For technical reasons that will become clear soon, we will assign a weight to the root, $\mathrm{wt}(s_1) = \frac{F}{k}$. For edges e having at least one end point in $S(r)$, q_e denotes the *fraction* of edge e that is in $S(r)$. If both endpoints of e are in $S(r)$, then $q_e = 1$. Otherwise, suppose $e = (u, v)$ with $u \in S(r)$ and $v \notin S(r)$. Then,

$$q_e = \frac{r - \mathrm{dist}(s_1, u)}{\mathrm{dist}(s_1, v) - \mathrm{dist}(s_1, u)}.$$

For the continuous process, the weight of a region is defined as

$$\mathrm{wt}(S(r)) = \mathrm{wt}(s_1) + \sum c_e d_e q_e,$$

where the sum is over all edges having at least one end point in $S(r)$. We will modify this definition slightly for the discrete process.

We want to guarantee that we will encounter the condition $c(S(r)) \leq \epsilon\,\mathrm{wt}(S(r))$ for $r < \frac{1}{2}$. Clearly, this can be guaranteed even for $r = 0$ by picking ϵ large enough; below we show how to pick a small ϵ.

Lemma 17.5 *Picking $\epsilon = 2\ln(k+1)$ suffices to ensure that the condition $c(S(r)) \leq \epsilon\,\mathrm{wt}(S(r))$ will be encountered before the radius becomes $\frac{1}{2}$.*

Proof : The proof is by contradiction. Suppose that throughout the region growing process, starting with $r = 0$ and ending at $r = \frac{1}{2}$, $c(S(r)) > \epsilon\,\mathrm{wt}(S(r))$. At any point, the incremental change in the weight of the region is

$$d\mathrm{wt}(S(r)) = \sum_e c_e d_e \, dq_e.$$

Clearly, only edges having one end point in $S(r)$ will contribute to the sum. Consider such an edge $e = (u, v)$ such that $u \in S(r)$ and $v \notin S(r)$. Then,

$$c_e d_e \, dq_e \;=\; c_e \frac{d_e}{\mathrm{dist}(s_1, v) - \mathrm{dist}(s_1, u)} \, dr.$$

Since $\mathrm{dist}(s_1, v) \leq \mathrm{dist}(s_1, u) + d_e$, we get $d_e \geq \mathrm{dist}(s_1, v) - \mathrm{dist}(s_1, u)$, and hence $c_e d_e \, dq_e \geq c_e \, dr$. This gives

$$d\mathrm{wt}(S(r)) \geq c(S(r)) \, dr > \epsilon\,\mathrm{wt}(S(r)) \, dr.$$

So, as long as the terminating condition is not encountered, the weight of the region increases exponentially with the radius. The initial weight of the region is $\frac{F}{k}$ and the final weight is at most $F + \frac{F}{k}$. Integrating we get

$$\int_{\frac{F}{k}}^{F+\frac{F}{k}} \frac{1}{\mathrm{wt}(S(r))} \, d\mathrm{wt}(S(r)) > \int_0^{\frac{1}{2}} \epsilon \, dr.$$

Therefore, $\ln(k+1) > \frac{1}{2}\epsilon$. However, this contradicts the assumption that $\epsilon = 2\ln(k+1)$, thus proving the lemma. □

The discrete process

The discrete process starts with $S = \{s_1\}$, and adds vertices to S in increasing order of their distance from s_1. So, essentially, it involves executing a shortest path computation from the root. Clearly, the sets of vertices found by both processes are the same.

The weight of region S is redefined for the discrete process as follows:

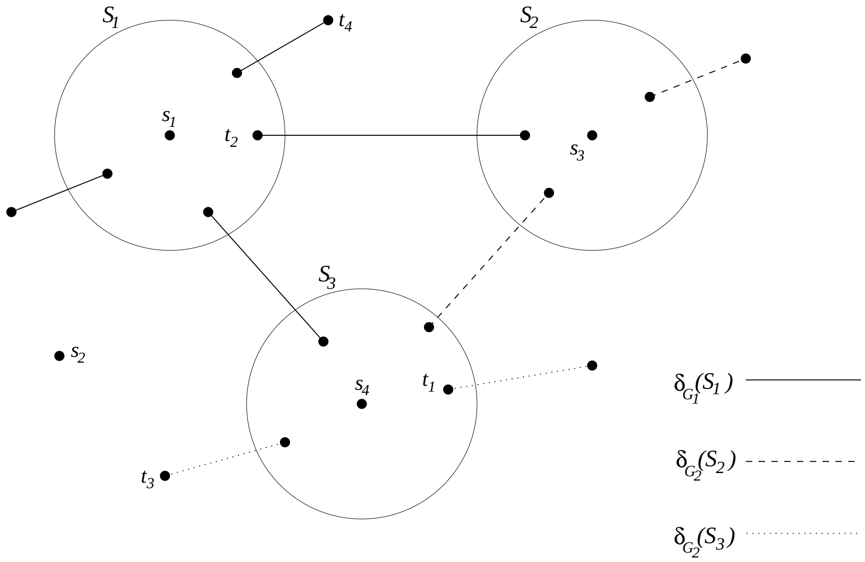
$$\mathrm{wt}(S) = \mathrm{wt}(s_1) + \sum_e c_e d_e,$$

where the sum is over all edges that have at least one end point in S, and $\mathrm{wt}(s_1) = \frac{F}{k}$. The discrete process stops at the first point when $c(S) \leq \epsilon\, \mathrm{wt}(S)$, where ϵ is again $2\ln(k+1)$. Notice that for the same set S, $\mathrm{wt}(S)$ in the discrete process is at least as large as that in the continuous process. Therefore, the discrete process cannot terminate with a larger set than that found by the continuous process. Hence, the set S found contains no source-sink pair.

Finding successive regions

The first region is found in graph G, starting with any one of the sources as the root. Successive regions are found iteratively. Let $G_1 = G$, and S_1 be the region found in G_1. Consider a general point in the algorithm when regions $S_1, \ldots, S_{i-1}$ have already been found. Now, G_i is defined to be the graph obtained by removing vertices $S_1 \cup \ldots \cup S_{i-1}$, together with all edges incident at them from G.

If G_i does not contain a source-sink pair, we are done. Otherwise, we pick the source of such a pair, say s_j, as the root, define its weight to be $\frac{F}{k}$, and grow a region in G_i. All defintions, such as distance and weight are w.r.t. graph G_i; we will denote these with a subscript of G_i. As before, the value of ϵ is $2\ln(k+1)$, and the terminating condition is $c(\delta_{G_i}(S_i)) \leq \epsilon\, \mathrm{wt}_{G_i}(S_i)$. Notice that in each iteration, the root is the only vertex that is defined to have non-zero weight.



In this manner, we will find regions $S_1, \ldots, S_l$, $l \leq k$, and will output the set $M = \delta_{G_1}(S_1) \cup \ldots \cup \delta_{G_l}(S_l)$. Since edges of each cut are removed from the graph for successive iterations, the sets in this union are disjoint, and $c(M) = \sum_i c(\delta_{G_i}(S_i))$.

The algorithm is summarized below. Notice that while a region is growing, edges with large distance labels will remain in its cut for a longer time, and so are more likely to be included in the multicut found. (Of course, the precise time that an edge remains in the cut is given by the difference between the distances from the root to the two end points of the edge.) So, as promised, the algorithm indirectly gives preference to edges with large distance labels.

Algorithm 17.6 (Minimum multicut)

1. Find an optimal solution to the LP (17.2), thus obtaining distance labels for edges of G.
2. $\epsilon \leftarrow 2\ln(k+1)$, $H \leftarrow G$, $M \leftarrow \emptyset$;
3. While $\exists$ a source-sink pair in H do:

   Pick such a source, say s_j;
   Grow a region S with root s_j until $c(\delta_H(S)) \leq \epsilon\, \mathrm{wt}_H(S)$;
   $M \leftarrow M \cup \delta_H(S)$;
   $H \leftarrow H$ with vertices of S removed;
4. Output M;

Lemma 17.7 *The set M found is a multicut.*

Proof : We need to prove that no region contains a source-sink pair. In each iteration i, the sum of weights of edges of the graph and the weight defined on the current root is bounded by $F + \frac{F}{k}$. So, by the proof of Lemma 17.5, the continuous region growing process is guaranteed to encounter the terminating condition before the radius of the region becomes $\frac{1}{2}$. Therefore, the distance between a pair of vertices in the region, S_i, found by the discrete process is also bounded by 1. Notice that we had defined these distances w.r.t. graph G_i. Since G_i is a subgraph of G, the distance between a pair of vertices in G cannot be larger than that in G_i. Hence, S_i contains no source-sink pair. □

Lemma 17.8 $c(M) \leq 2\epsilon F = 4\ln(k+1)F$.

Proof : In each iteration i, by the terminating condition, we have $c(\delta_{G_i}(S_i)) \leq \epsilon\, \mathrm{wt}_{G_i}(S_i)$. Since all edges contributing to $\mathrm{wt}_{G_i}(S_i)$ will be removed from the graph after this iteration, each edge of G contributes to the weight of at most one region. The total weight of all edges of G is F. Since each iteration helps disconnect at least one source-sink pair, the number of iterations is bounded by k. Therefore, the total weight attributed to source vertices is at most F. Adding up we get,

$$c(M) = \sum_i c(\delta_{G_i}(S_i)) \leq \epsilon \left(\sum_i \mathrm{wt}_{G_i}(S_i) \right) \leq \epsilon \left(k\frac{F}{k} + \sum_e c_e d_e \right) = 2\epsilon F.$$

□

Theorem 17.9 *Algorithm 17.6 achieves an approximation guarantee of $O(\log k)$ for the minimum multicut problem.*

Proof : The proof follows from Lemmas 17.7 and 17.8, and the fact that the value of the fractional multicut, F, is a lower bound on the minimum multicut. □

Corollary 17.10 *In an undirected graph with k source-sink pairs,*

$$\max_{\text{m/c flow } F} |F| \;\leq\; \min_{\text{multicut } C} |C| \;\leq\; O(\log k) \left(\max_{\text{m/c flow } F} |F| \right),$$

where $|F|$ represents the value of multicommodity flow F, and $|C|$ represents the capacity of multicut C.

Example 17.11 We will construct an infinite family of graphs for which the integrality gap for LP (17.2) is $\Omega(\log k)$, thereby showing that our analysis of Algorithm 17.6 and the approximate max-flow min-multicut theorem presented in Corollary 17.10 are tight within constant factors.

The construction uses expander graphs. Graph $G = (V, E)$ is said to be an *expander graph* if for each set $S \subset V$ with $|S| \leq \frac{|V|}{2}$, $|\delta(S)| \geq |S|$. For an appropriate constant d, there exists an infinite family of expander graphs with each vertex having degree d. Let H be such a graph containing k vertices.

Source-sink pairs are designated in H as follows. Consider a breadth first search tree rooted at some vertex v. The number of vertices within distance $\alpha - 1$ of vertex v is at most $1 + d + d^2 + \ldots + d^{\alpha-1} < d^{\alpha}$. Picking $\alpha = \lfloor \log_d \frac{k}{2} \rfloor$ ensures that at least $\frac{k}{2}$ vertices are at a distance $\geq \alpha$ of v. Let us say that a pair of vertices are a source-sink pair if the distance between them is at least α. So, we have chosen $\Theta(k^2)$ pairs of vertices as source-sink pairs.

Each edge in H is of unit capacity. So, the total capacity of edges of H is $O(k)$. Since the distance between each source-sink pair is $\Omega(\log k)$, any flow path carrying a unit of flow uses up $\Omega(\log k)$ units of capacity. Therefore, the value of maximum multicommodity flow in H is bounded by $O(\frac{k}{\log k})$. Next, we will prove that a minimum multicut in H, say M, has capacity $\Omega(k)$ thereby proving the claimed integrality gap. Consider the connected components obtained by removing M from H.

Claim 17.12 *Each connected component has at most $\frac{k}{2}$ vertices.*

Proof : Suppose a connected component has $> \frac{k}{2}$ vertices. Pick an arbitrary vertex v in this component. By the argument given above, the number of vertices that are within distance $\alpha - 1$ of v in the entire graph H is $< d^{\alpha} \leq \frac{k}{2}$. So, there is a vertex u in the component such that the distance between u and v is at least α, i.e., u and v form a source-sink pair. Thus removal of M has failed to disconnect a source-sink pair, leading to a contradiction. □

By Claim 17.12, and the fact that H is an expander, each component S has $|\delta(S)| \geq |S|$. Since each vertex of H is in one of the components, $\sum_S |\delta(S)| \geq k$, where the sum is over all connected components. Since an edge contributes to the cuts of at most two components, we get that the number of edges crossing components is $\Omega(k)$. This gives the desired lower bound on the minimum multicut.

Next, let us ensure that the number of surce-sink pairs defined in the graph is not related to the number of vertices in it. Notice that replacing an edge of H by a path of unit capacity edges does not change the value of maximum flow or minimum multicut. Using this operation, we can construct from H a graph G having n vertices, for arbitrary $n \geq k$. The integrality gap of LP (17.2) for G is $\Omega(\log k)$.

□

Some applications of multicut

In this section, we will obtain $O(\log n)$ factor approximation algorithms for some problems by reducing to the minimum multicut problem.

Problem 17.13 (Minimum clause deletion in 2CNF $\equiv$ formula) A 2CNF $\equiv$ formula consists of a set of clauses of the form $(p \equiv v)$, where u and v are literals. Let F be such a formula, and wt be a function assigning non-negative rational weights to its clauses. The problem is to delete a minimum weight set of clauses of F so that the remaining formula is satisfiable.

Given a 2CNF $\equiv$ formula F on n Boolean variables, let us define graph $G(F)$ with edge capacities as follows: The graph has $2n$ vertices, one corresponding to each literal. Corresponding to each clause $(p \equiv q)$ we include the two edges (p, q) and $(\overline{p}, \overline{q})$, each having capacity equal to the weight of the clause $(p \equiv q)$.

Notice that the two clauses $(p \equiv q)$ and $(\overline{p} \equiv \overline{q})$ are equivalent. W.l.o.g. we may assume that F does not contain two such equivalent clauses, since we can merge their weights and drop one of these clauses. With this assumption, each clause corresponds to two distinct edges in $G(F)$.

Lemma 17.14 *Formula F is satisfiable iff no connected component of $G(F)$ contains a variable and its negation.*

Proof : If (p, q) is an edge in $G(F)$ then the literals p and q must take the same truth value in every satisfying truth assignment. So, all literals of a connected component of $G(F)$ are forced to take the same truth value. Therefore, if F is satisfiable, no connected component in $G(F)$ contains a variable and its negation.

Conversely, notice that if literals p and q occur in the same connected component, then so do their negations. So, if no connected component contains a variable and its negation, the components can be paired, so that in each pair, one component contains a set of literals and the other contains the complementary literals. For each pair, set the literals of one component to true and the other to false, to obtain a satisfying truth assignment. □

For each variable and its negation, designate the corresponding vertices in $G(F)$ to be a source-sink pair, thus defining n source-sink pairs. Let M be a minimum multicut in $G(F)$ and C be a minimum weight set of clauses whose deletion makes F satisfiable. In general, M may have only one of the two edges corresponding to a clause.

Lemma 17.15 $\mathrm{wt}(C) \leq c(M) \leq 2\mathrm{wt}(C)$.

Proof : Delete clauses corresponding to edges of M from F to get formula F'; the weight of clauses deleted is at most $c(M)$. Since $G(F')$ does not contain any edges of M, it does not have any component containing a variable and its negation. By Lemma 17.14, F' is satisfiable, thus proving the first inequality.

Next, delete from $G(F)$ the two edges corresponding to each clause in C. This will disconnect all source-sink pairs. Since the capacity of edges deleted is $2\mathrm{wt}(C)$, this proves the second inequality. □

Since we can approximate minimum multicut to within an $O(\log n)$ factor, we get:

Theorem 17.16 *There is an $O(\log n)$ factor approximation algorithm for Problem 17.13.*

Next, consider the following problem, which has applications in VLSI design:

Problem 17.17 (Minimum edge deletion graph bipartization) Given an edge weighted undirected graph $G = (V, E)$, remove a minimum weight set of edges to leave a bipartite graph.

Exercise 17.18 Obtain an $O(\log n)$ factor approximation algorithm for Problem 17.17 by reducing it to Problem 17.13.

Exercise 17.19 In Chapter 4, we had presented a $(2 - \frac{2}{k})$ factor algorithm for the minimum multiway cut problem by comparing the solution found to the integral optimal solution. In this exercise, we develop an algorithm with the same guarantee using LP-duality.

Given terminals $s_1, \ldots, s_k$, consider the multicommodity flow problem in which each pair of terminals can form a source-sink pair. Thus there are $\binom{k}{2}$ commodities. Give an LP for maximizing this multicommodity flow, and obtain the dual LP. The dual seeks a distance label assignment for edges satisfying the triangle inequality, and ensuring that the distance between any two terminals is at least 1. An optimal solution to the dual can be viewed as a fractional multiway cut.

The algorithm for finding a multiway cut is: Solve the dual LP to obtain an optimal fractional multiway cut; this gives a distance label assignment, say d. Pick ρ at random in the interval $[0, \frac{1}{2}]$. An edge (u, v) is picked iff for some terminal s, $d(u, s) \leq \rho \leq d(v, s)$.

Prove that the expected capacity of cut picked is at most twice the optimal fractional multiway cut, by showing that for each edge (u, v), the probability that it is picked is bounded by $2d(u, v)$. Derandomize this algorithm, and give a modification to make it a factor $(2 - \frac{2}{k})$ algorithm.

Chapter 18

Sparsest cut

In this chapter, we will obtain an approximation algorithm for the sparsest cut problem using an interesting LP-rounding procedure that employs results on low distortion embeddings of metrics in ℓ_1 spaces. As mentioned in Chapter 17, we will get, as a corollary, an approximate max-flow min-cut theorem for the demands version of multicommodity flow. Several important problems can be viewed as special cases of the sparsest cut problem.

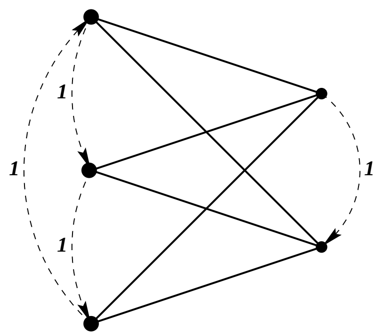
Problem 18.1 (Demands multicommodity flow) Let $G = (V, E)$ be an undirected graph with a non-negative capacity c_e for each edge $e \in E$. Let $\{(s_1, t_1), \ldots, (s_k, t_k)\}$ be a specified set of pairs of vertices, where each pair is distinct, but vertices in different pairs are not required to be distinct. A separate commodity is defined for each (s_i, t_i) pair; for convenience, we will think of s_i as the source and t_i as the sink of this commodity. For each commodity i, a non-negative demand, dem_i, is also specified. The objective is to maximize f, called *throughput*, such that for each commodity i, $f \cdot \mathrm{dem}_i$ units of this commodity can be routed simultaneously, subject to flow conservation and capacity constraints, i.e., each commodity must satisfy flow conservation at each vertex other than its own source and sink, and the sum of flows routed through an edge, in both directions combined, should not exceed the capacity of this edge. We will denote the optimal throughput by f^*.

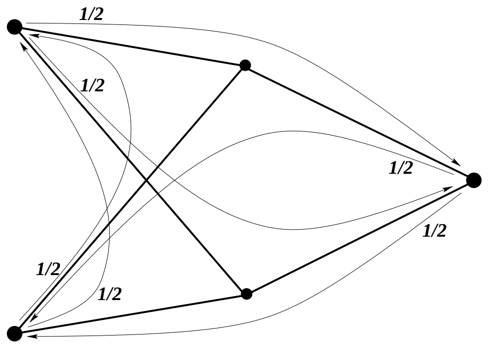
Consider a cut $(S, \overline{S})$ in G. Let $c(S)$ denote the capacity of edges in this cut, and $\mathrm{dem}(S)$ denote the total demand separated by this cut, i.e., $\mathrm{dem}(S) = \sum\{\mathrm{dem}_i : |\{s_i, t_i\} \cap S\}| = 1\}$. Clearly, the ratio of these quantities places an upper bound on the throughput, i.e., $f^* \leq \frac{c(S)}{\mathrm{dem}(S)}$. This motivates:

Problem 18.2 (Sparsest cut) Let $G = (V, E)$ be an undirected graph, with capacities, source-sink pairs and demands defined as in Problem 18.1. The *sparsity* of cut $(S, \overline{S})$ is given by $\frac{c(S)}{\mathrm{dem}(S)}$. The problem is to find a cut of minimum sparsity. We will denote the sparsity of this cut by α^*.

Among all cuts, α^* puts the most stringent upper bound on f^*. Is this upper bound tight? Example 18.3 shows that it is not. However, minimum sparsity cannot be arbitrarily larger than maximum throughput; we will show that their ratio is bounded by $O(\log k)$.

Example 18.3 Consider the bipartite graph $K_{3,2}$ with all edges of unit capacity, and a unit demand between each pair of non-adjacent vertices; a total of four commodities.





It is easy to check that a sparsest cut of $K_{3,2}$ has sparsity 1. The figure shows the unique way of routing one unit of each of the three commodities having the source and the sink on the side containing 3 vertices. However, this saturates all edges, making it impossible to route the fourth commodity. Hence throughput is strictly smaller than 1. □

Linear programming formulation

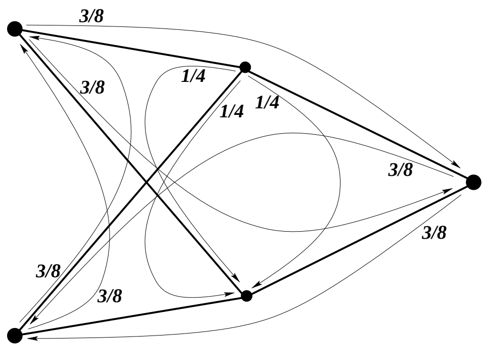
We start by giving a linear programming formulation of the demands multicommodity flow problem. Let $\mathcal{P}_i = \{q_j^i\}$ denote the set of all paths between s_i and t_i. Introduce variable f_j^i to denote the flow of commodity i sent along path q_j^i.

$$\begin{array}{lll} \text{maximize} & f & \\ \text{subject to} & \sum_j f_j^i \geq f \cdot \text{dem}(i), & i = 1, \ldots, k \\ & \sum_{e \in q_j^i} f_j^i \leq c_e, & e \in E \\ & f \geq 0 & \\ & f_j^i \geq 0 & \end{array} \tag{18.1}$$

Let l_i and d_e be dual variables associated with the first and second set of inequalities respectively. We will interpret d_e's as distance label assignments to the edges of G. The first set of inequalities are simply ensuring that for each commodity i, l_i is bounded by the length of a shortest path from s_i to t_i w.r.t. the distance label assignment.

$$\begin{array}{lll} \text{minimize} & \sum_{e \in E} c_e d_e & \\ \text{subject to} & \sum_{e \in q_j^i} d_e \geq l_i, & q_j^i \in \mathcal{P}_i \\ & \sum_{i=1}^k l_i \text{dem}(i) \geq 1 & \\ & d_e \geq 0, & e \in E \\ & l_i \geq 0, & i = 1, \ldots, k \end{array} \tag{18.2}$$

Example 18.4 For the instance given in Example 18.3, the optimal throughput is $f^* = \frac{3}{4}$; this corresponds to routing the four commodities as follows:



The optimal dual solution is: $d_e = \frac{1}{8}$ for each edge e, and $l_i = \frac{1}{4}$ for each commodity i. It will be instructive for the reader to verify feasibility and optimality of these solutions. □

Claim 18.5 *There is an optimal distance label assignment* $\boldsymbol{d}$ *for the dual program (18.2) that is a metric on* V*. Furthermore, for each commodity* i*,* $l_i = d_{(s_i,t_i)}$*, and the second inequality holds with equality, i.e.,* $\sum_i d_{(s_i,t_i)}\mathrm{dem}(i) = 1$*.*

Proof : If for some three points u, v and w, $d_{uw} > d_{uv} + d_{vw}$, then we can decrease d_{uw} to $d_{uv} + d_{vw}$. Since this does not decrease the shortest path between any s_i-t_i pair, the solution still remains feasible. Moreover, the objective function value cannot increase by this process. Continuing in this manner, we will obtain a metric on V.

Now, the length of a shortest path from s_i to t_i is given by the distance label $d_{(s_i,t_i)}$. Setting $l_i = d_{(s_i,t_i)}$ does not change the feasibility or the objective function value of the solution. Finally, if the second inequality holds strictly, then we can scale down all distance labels without violating feasibility, thus contradicting the optimality of $\boldsymbol{d}$. □

Define the graph H with vertex set $V_H = \{s_i, t_i | 1 \le i \le k\}$ and edge set $E_H = \{(s_i, t_i) | 1 \le i \le k\}$ to be the *demand graph*. By Claim 18.5, the dual program yields a metric $(V, \boldsymbol{d})$ that minimizes

$$\frac{\sum_{e \in G} c_e d_e}{\sum_{e \in H} \mathrm{dem}(e) d_e}.$$

By the LP-Duality Theorem, this equals the optimal throughput. Hence,

$$f^* = \min_{\text{metric } \boldsymbol{d}} \frac{\sum_{e \in G} c_e d_e}{\sum_{e \in H} \mathrm{dem}(e) d_e}.$$

Approximate cut packing

We will obtain a sparse cut from metric $(V, \boldsymbol{d})$ obtained above, using the notion of approximate cut packings. Let us think of this metric as defining the lengths of edges of the complete graph on V; let E_n denote the set of all edges in the complete graph. Let y be a function assigning non-negative values to subsets of V, i.e., $y : 2^V \to \mathbf{R}^+$; we will denote the value of y on set S by y_S. As before, let us say that edge e *feels* y_S if e is in the cut $(S, \overline{S})$. The *amount of cut* that edge e feels is $\sum_{S: e \in \delta(S)} y(S)$. Function y is called a *cut packing* for metric (V, d) if no edge feels more cut than its length, i.e., for each edge $e \in E_n$, $\sum_{S: e \in \delta(S)} y(S) \le d_e$. If this inequality holds with equality for each edge $e \in E_n$, then y is said to be an *exact cut packing*. The reason for the name "cut packing" is that equivalently, we can think of y as assigning value $y_S + y_{\overline{S}}$ to each cut $(S, \overline{S})$.

As shown below, in general, there may not be an exact cut packing for metric $(V, \boldsymbol{d})$. Let us relax this notion by allowing edges to be underpacked up to a specified extent. For $\beta \geq 1$, y is said to be a β-approximate cut packing if the amount of cut felt by any edge is at least $\frac{1}{\beta}$ fraction of its length, i.e., for each edge $e \in E_n$, $\frac{d_e}{\beta} \leq \sum_{S:e\in\delta(S)} y(S) \leq d_e$. Clearly, the smaller β is, the better is the cut packing. The following theorem shows the importance of finding a good cut packing for $(V, \boldsymbol{d})$.

Theorem 18.6 *Let y be a β-approximate cut packing for metric $(V, \boldsymbol{d})$. Let $(S', \overline{S'})$ be the sparsest cut with $y_{S'} \neq 0$. Then, the sparsity of this cut is at most βf^*.*

Proof : Let y be a β-approximate cut packing for metric $(V, \boldsymbol{d})$. Then,

$$f^* = \frac{\sum_{e\in G} c_e d_e}{\sum_{e\in H} \mathrm{dem}(e) d_e} \geq \frac{\sum_{e\in G} c_e \sum_{S:e\in\delta(S)} y(S)}{\sum_{e\in H} \mathrm{dem}(e) \sum_{S:e\in\delta(S)} \beta y(S)} = \frac{\sum_S y_S c(S)}{\beta \sum_S y_S \mathrm{dem}(S)} \geq \frac{1}{\beta}\left(\frac{c(S')}{\mathrm{dem}(S')}\right).$$

The first inequality follows using both the upper bound and the lower bound on the amount of cut felt by an edge; the former in the numerator and the latter in the denominator. The equality after that follows by changing the order of summation. The last inequality follows from the well known Proposition 18.7 stated below. □

Proposition 18.7 *For any non-negative reals $a_1, \ldots, a_n$, and positive reals $b_1, \ldots, b_n$,*

$$\frac{\sum_i a_i}{\sum_i b_i} \geq \min_i \frac{a_i}{b_i}.$$

Moreover, this inequality holds with equality iff the n values a_i/b_i are all equal.

Corollary 18.8 *If there is an exact cut packing for metric $(V, \boldsymbol{d})$, then every cut $(S, \overline{S})$ with $y_S \neq 0$ has sparsity f^*, and so is a sparsest cut in G.*

Proof : By Theorem 18.6, the minimum sparsity cut with $y_S \neq 0$ has sparsity at most f^* (since $\beta = 1$). Since the sparsity of any cut upper bounds f^*, the sparsity of this cut equals f^*, and this is a sparsest cut in G. But then all inequalities in the proof of Theorem 18.6 must hold with equality. Now, by the second statement in Proposition 18.7, we get that every cut $(S, \overline{S})$ with $y_S \neq 0$ has sparsity f^*. □

The sparsest cut in the instance specified in Example 18.3 has sparsity strictly larger than f^*. So, by Corollary 18.8, the optimal metric for this instance does not have an exact cut packing. However, it turns out that every metric has an $O(\log n)$-approximate cut packing – we will show this using the notion of ℓ_1-embeddability of metrics.

ℓ_1-embeddability of metrics

A *norm* on the vector space $\mathbf{R}^m$ is a function $\|\cdot\| : \mathbf{R}^m \to \mathbf{R}^+$, such that for any $\boldsymbol{x}, \boldsymbol{y} \in \mathbf{R}^m$, and $\lambda \in \mathbf{R}$:

- $\|\boldsymbol{x}\| = 0$ if and only if $\boldsymbol{x} = 0$,
- $\|\lambda \boldsymbol{x}\| = |\lambda| \cdot \|\boldsymbol{x}\|$,

- $\|\boldsymbol{x}+\boldsymbol{y}\| \leq \|\boldsymbol{x}\|+\|\boldsymbol{y}\|$.

For $p \geq 1$, the ℓ_p-*norm* is defined by

$$\|\boldsymbol{x}\|_p = \left(\sum_{1 \leq k \leq m} |x_k|^p \right)^{\frac{1}{p}}.$$

The associated ℓ_p-*metric*, denoted by $\boldsymbol{d}_{\ell_p}$, is defined by

$$d_{\ell_p}(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|_p$$

for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbf{R}^m$. In this chapter, we will only consider the ℓ_1-norm.

Let σ be a mapping, $\sigma : V \to \mathbf{R}^m$ for some m. Let us say that $\|\sigma(u) - \sigma(v)\|_1$ is the ℓ_1 *length of edge* (u, v) *under* σ. We will say that σ is an *isometric* ℓ_1-*embedding* for metric $(V, \boldsymbol{d})$ if it preseves the ℓ_1 lengths of all edges, i.e.,

$$\forall u, v \in V, d(u, v) = \|\sigma(u) - \sigma(v)\|_1.$$

As shown below, in general, the metric computed by solving the dual program may not be isometrically ℓ_1-embeddable. So, we will relax this notion – we will ensure that the mapping does not *stretch* any edge, but we will allow it to *shrink* edges up to a specified factor. For $\beta \geq 1$, we will say that σ is a β-*distortion* ℓ_1-*embedding* for metric (V, d) if

$$\forall u, v \in V : \ \frac{1}{\beta} d(u, v) \leq \|\sigma(u) - \sigma(v)\|_1 \leq d(u, v).$$

Next, we show that the question of finding an approximate cut packing for a metric is intimately related to that of finding a low distortion ℓ_1 embedding for it.

Lemma 18.9 *Given mapping $\sigma : V \to \mathbf{R}^m$ for some m, there is a cut packing $y : 2^V \to \mathbf{R}^+$ such that each edge feels as much cut under y as its ℓ_1 length under σ. Moreover, the number of non-zero y_S's is at most $m(n-1)$.*

Proof : First consider the case when $m = 1$. Let the n vertices of V be mapped to $u_1 \leq u_2 \leq \ldots \leq u_n$. W.l.o.g. assume that the vertices are also numbered in this order. For each $i, 1 \leq i \leq n-1$, let $y_{\{v_1,\ldots,v_i\}} = u_{i+1} - u_i$. Clearly, this cut packing satisfies the required condition.

For arbitrary m, we observe that since the ℓ_1 norm is additive, we can define a cut packing for each dimension independently, and the sum of these packings satisfies the required condition. □

Lemma 18.10 *Given a cut packing $y : 2^V \to \mathbf{R}^+$ with m non-zero y_S's, there is a mapping $\sigma : V \to \mathbf{R}^m$ such that for each edge, its ℓ_1 length under σ is the same as the amount of cut it feels under y.*

Proof : We will have a dimension corresponding to each set $S \subseteq V$ such that $y_S \neq 0$. For vertices in S, this coordinate will be 0, and for vertices in $\overline{S}$, this coordinate will be y_S. Thus, this dimension contributes exactly as much to the ℓ_1 length of an edge as the amount of cut felt by this edge due to y_S. Hence this mapping satisfies the required condition. □

Lemmas 18.9 and 18.10 give:

Theorem 18.11 *There exists a β-distortion ℓ_1-embedding for metric $(V, \boldsymbol{d})$ if and only if there exists a β-approximate cut packing for it. Moreover, the number of non-zero cuts and the dimension of the ℓ_1-embedding are polynomially related.*

Corollary 18.12 *Metric $(V, \boldsymbol{d})$ is isometrically ℓ_1-embeddable iff there exists an exact cut packing for it.*

We have already shown that metric obtained for the instance in Example 18.3 does not have an exact cut packing, so, it is not isometrically ℓ_1-embeddable. However, we will show that any metric has an $O(\log n)$-distortion ℓ_1-embedding; this fact lies at the heart of the approximation algorithm for the sparsest cut problem.

Low distortion ℓ_1-embeddings for metrics

First consider the following one dimensional embedding for metric $(V, \boldsymbol{d})$: pick a set $S \subseteq V$, and define the coordinate of vertex v to be $\sigma(v) = \min_{s \in S} d(s, v)$, i.e., the length of the shortest edge from v to S. This mapping does not stretch any edge:

Lemma 18.13 *For the one dimensional embedding given above,*

$$\forall u, v \in V, \quad |\sigma(u) - \sigma(v)| \leq d(u, v).$$

Proof : Let s_1 and s_2 be the closest vertices of S to u and v respectively. W.l.o.g. assume that $d(s_1, u) \leq d(s_2, v)$. Then, $|\sigma(u) - \sigma(v)| = d(s_2, v) - d(s_1, u) \leq d(s_2, v) - d(s_2, u) \leq d(u, v)$. The last inequality follows by triangle inequality. □

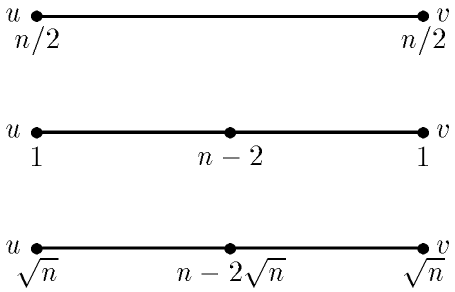
More generally, consider the following m-dimensional embedding: Pick m subsets of V, $S_1, \ldots, S_m$, and define the i^{th} coordinate of vertex v to be $\sigma_i(v) = (\min_{s \in S_i} d(s, v))/m$; notice the scaling factor of m used. The additivity of ℓ_1 metric, together with Lemma 18.13, imply that this mapping also does not stretch any edge.

Ensuring that a single edge is not over-shrunk

The remaining task is to choose the sets in such a way that no edge shrinks by a factor of more than $O(\log n)$. It is natural to use randomization for picking the sets. Let us first ensure that a single edge (u, v) is not over-shrunk. For this purpose, define the *expected contribution of set S_i* to the ℓ_1 length of edge (u, v) to be $E(|\sigma_i(u) - \sigma_i(v)|)$.

For simplicity, assume that n is a power of 2; let $n = 2^l$. For $2 \leq i \leq l + 1$, set S_i is formed by picking each vertex of V with probability $1/2^i$. The embedding w.r.t. these sets works for the single edge (u, v), with high probability. The proof of this fact involves cleverly taking into consideration the expected contribution of each set. For different metrics, different sets have large contribution. In order to develop intuition for the proof, we first illustrate this through a series of examples.

Example 18.14 In the following three metrics, $d(u, v) = 1$. Relative to u and v, the remaining $(n - 2)$ vertices are placed as shown in the figure below.



For each metric, the expected contribution of one of the sets is $\Omega(d(u,v)/l)$. For the first metric, this set is S_l, since it will be a singleton with constant probability. For the second metric, this set is S_2, since it will contain exactly one of u and v with constant probability. For the third metric, this set is $S_{\lceil l/2 \rceil}$, since with constant probability, it will contain exactly one vertex of the $2\sqrt{n}$ vertices bunched up with u and v. □

In the next lemma, we encapsulate the basic mechanism for establishing a lower bound on the expected contribution of a set S_i. For any vertex x and non-negative real r, let $B(x,r)$ denote the *ball* of radius r around x, i.e., $B(x,r) = \{s \in V | d(x,s) \leq r\}$.

Lemma 18.15 *If for some choice of $r_1 \geq r_2 \geq 0$, and constant c,*

$$\mathbf{Pr}[(S_i \cap B(u,r_1) = \emptyset) \textit{ and } (S_i \cap B(v,r_2) \neq \emptyset)] \geq c,$$

then the expected contribution of S_i is $\geq c(r_1 - r_2)/l$.

Proof : Under the event described, $d(u,S_i) \geq r_1$ and $d(v,S_i) \leq r_2$. If so, $\sigma_i(u) \geq r_1/l$ and $\sigma_i(v) \leq r_2/l$. Therefore, $|\sigma_i(u) - \sigma_i(v)| \geq (r_1 - r_2)/l$, and the lemma follows. □

The remaining task is to define suitable radii r_1 and r_2 for each set S_i such that the probabilistic statement of Lemma 18.15 holds. We will need the following simple probabilistic fact:

Lemma 18.16 *For $1 \leq t \leq l-1$, let A and B be disjoint subsets of V, such that $|A| < 2^t$ and $|B| \geq 2^{t-1}$. Form set S by picking each vertex of V independently with probability $p = 1/(2^{t+1})$. Then,*

$$\mathbf{Pr}[(S \cap A = \emptyset) \textit{ and } (S \cap B \neq \emptyset)] \geq \frac{7}{64}.$$

Proof :

$$\mathbf{Pr}[S \cap A = \emptyset] = (1-p)^{|A|} \geq (1 - p|A|) \geq \frac{1}{2},$$

where the first inequality follows by taking the first two terms of the binomial expansion.

$$\mathbf{Pr}[S \cap B = \emptyset] = (1-p)^{|B|} \leq 1 - p|B| + p^2 \left(\frac{|B|^2}{2}\right);$$

this time around, we used the first three terms of the binomial expansion. Therefore,

$$\mathbf{Pr}[S \cap B \neq \emptyset] = 1 - (1-p)^{|B|} \geq p|B|(1 - \frac{p|B|}{2}) = \frac{1}{4}(1 - \frac{1}{8}) = \frac{7}{32}.$$

Finally observe that since A and B are disjoint, the two events $(S \cap A = \emptyset)$ and $(S \cap B \neq \emptyset)$ are independent. The lemma follows. □

For $0 \leq t \leq l$, define $\rho_t = \min\{\rho \geq 0 : \ |B(u,\rho)| \geq 2^t \text{ and } |B(v,\rho)| \geq 2^t\}$, i.e., ρ_t is the smallest radius such that the ball around u and the ball around v each has at least 2^t vertices. Clearly, $\rho_0 = 0$ and $\rho_l \geq d(u,v)$. Let $\hat{t} = \max\{t : \ \rho_t < d(u,v)/2\}$; clearly, $\hat{t} \leq l-1$. Finally, for any vertex x and non-negative real r, let $B^\circ(x,r)$ denote the *open ball* of radius r around x, i.e., $B^\circ(x,r) = \{s \in V | d(x,s) < r\}$.

Lemma 18.17 *For $1 \leq t \leq \hat{t}$, the expected contribution of S_{t+1} is*

$$\geq \frac{7}{64} \frac{(\rho_t - \rho_{t-1})}{l},$$

and for $t = \hat{t}+1$, the expected contribution of S_{t+1} is

$$\geq \frac{7}{64l}(\frac{d(u,v)}{2} - \rho_{t-1}).$$

Proof : First consider t such that $1 \leq t \leq \hat{t}$. By the definition of ρ_t, for at least one of the two vertices u and v, the open ball of radius ρ_t contains fewer than 2^t vertices. W.l.o.g. assume this happens for vertex u, i.e., $|B^\circ(u,\rho_t)| < 2^t$. Again, by definition, $|B(v,\rho_{t-1})| \geq 2^{t-1}$. Since $\rho_{t-1} < \rho_t < d(u,v)/2$, the two sets $B^\circ(u,\rho_t)$ and $B(v,\rho_{t-1})$ are disjoint. So, by Lemma 18.16, the probability that S_{t+1} is disjoint from the first set and intersects the second is least $7/64$. Now, the first claim follows from Lemma 18.15.

Next, let $t = \hat{t}+1$. By the definition of $\hat{t}$, for at least one of the two vertices u and v, the open ball of radius $d(u,v)/2$ contains fewer than 2^t vertices. As before, w.l.o.g. assume this happens for vertex u, i.e., $|B^\circ(u,d(u,v)/2)| < 2^t$. Clearly, $|B(v,\rho_{t-1})| \geq 2^{t-1}$. Since $\rho_{t-1} < d(u,v)/2$, the two sets $B^\circ(u,d(u,v)/2)$ and $B(v,\rho_{t-1})$ are disjoint. The rest of the reasoning is same as before. □

Lemma 18.18 *The expected contribution of all sets $S_2, \ldots, S_{l+1}$ is*

$$\geq \frac{7}{128} \frac{d(u,v)}{l}.$$

Proof : By Lemma 18.17, the expected contribution of all sets $S_2, \ldots, S_{l+1}$ is at least the following telescoping sum:

$$\frac{7}{64l}\left((\rho_1 - \rho_0) + (\rho_2 - \rho_1) + \ldots + (\frac{d(u,v)}{2} - \rho_{\hat{t}})\right) = \frac{7}{128}\frac{d(u,v)}{l}.$$

□

For convenience, let $c = 7/128$.

Lemma 18.19 $\quad$ $\mathbf{Pr}$[*contribution of all sets is* $\geq \frac{cd(u,v)}{2l}$] $\geq \frac{c}{2-c}$.

Proof : Denote the probability in question by p. Clearly, the total contribution of all sets $S_2, \ldots, S_{l+1}$ to the ℓ_1 length of edge (u,v) is at most $d(u,v)/2l$. This fact and Lemma 18.18 give:

$$p\frac{d(u,v)}{l} + (1-p)\frac{cd(u,v)}{2l} \geq \frac{d(u,v)}{l}.$$

Therefore, $p \geq c/(2-c)$. $\square$

Exercise 18.20 For each of the three metrics given in Example 18.14, one of the sets $S_2, \ldots, S_{l+1}$ has an expected contribution of $\Omega(d(u,v)/l)$. Give a metric for which each set has an expected contribution of $\Theta(d(u,v)/l^2)$.

Ensuring that no edge is over-shrunk

The above embedding does not over-shrink edge (u,v) with constant probability. In order to ensure that no edge is over-shrunk, we will first enhance this probability. The key idea is to repeat the entire process several times independently and use Chernoff bounds to bound the error probability. We will use the following statement of the Chernoff bound: Let $X_1, \ldots, X_n$ be independent Bernoulli trials with $\mathbf{Pr}[X_i] = p$, $0 < p < 1$, and let $X = \sum_{i=1}^{n} X_i$; clearly, $E(X) = np$. Then, for $0 < \delta \leq 1$,

$$\mathbf{Pr}[X < (1-\delta)np] < \exp(-\delta^2 np/2).$$

Pick sets $S_2, \ldots, S_{l+1}$, using probabilities specified above, independently $N = O(\log n)$ times each. Call the sets so obtained $S_i^j, 2 \leq i \leq l+1, 1 \leq j \leq N$. Consider the $Nl = O(\log^2 n)$ dimensional embedding of metric $(V, \boldsymbol{d})$ w.r.t. these Nl sets. We will prove that this is an $O(\log n)$-distortion ℓ_1-embedding for metric (V, d).

Lemma 18.21 *For $N = O(\log n)$, this embedding satisfies:*

$$\mathbf{Pr}[\|\sigma(u) - \sigma(v)\|_1 \geq \frac{pcd(u,v)}{4l}] \geq (1 - \frac{1}{2n^2}),$$

where $p = c/(2-c)$.

Proof : We will think of the process of picking sets $S_2, \ldots, S_{l+1}$ once as a single Bernoulli trial; thus we have N such trials. A trial succeeds if the contribution of all its sets is $\geq (cd(u,v))/2l$. By Lemma 18.19, the probability of success is at least p. Using the Chernoff bound with $\delta = 1/2$, the probability that at most $Np/2$ of these trials succeed is at most $\exp(pN/8)$. Clearly, this is bounded by $1/2n^2$ for $N = O(\log n)$. If at least $Np/2$ trials succeed, the ℓ_1 length of edge (u,v) will be at least $(pcd(u,v))/4l = d(u,v)/O(\log n)$. The lemma follows. $\square$

Adding the error probabilities for all $n(n-1)/2$ edges, we get:

Theorem 18.22 *The $Nl = O(\log^2 n)$ dimensional embedding given above is an $O(\log n)$-distortion ℓ_1-embedding for metric (V, d), with probability at least $1/2$.*

The algorithm

The reader can verify that Claim 18.5 and Theorems 18.6, 18.11 and 18.22 lead to an $O(\log n)$ factor approximation algorithm for the sparsest cut problem. In this section, we will improve the approximation guarantee to $O(\log k)$ where k is the number of source-sink pairs specified.

For this purpose, notice that Theorem 18.6 holds even for the following less stringent approximate cut packing: no edge is allowed to be over-packed, and the edges of the demand graph are not under-packed by more than a β factor (the rest of the edges are allowed to be under-packed to any extent). In turn, such a cut packing can be obtained from an ℓ_1-embedding that does not over-shrink edges of the demand graph only. Since these are only $O(k^2)$ in number, where k is the number of source-sink pairs, we can ensure that these edges are not shrunk by a factor of more than $O(\log k)$, thus enabling an improvement in the approximation guarantee.

Let $V' \subseteq V$ be the set of vertices that are sources or sinks, $|V'| \leq 2k$. For simplicity, assume $|V'|$ is a power of 2; let $|V'| = 2^l$. The sets $S_2, \ldots, S_{l+1}$ will be picked from V', and it is easy to verify from the proof of Lemma 18.21 that $N = O(\log k)$ will suffice to ensure that none of the $O(k^2)$ edges of the demand graph is shrunk by more than a factor of $O(\log k)$. The complete algorithm is:

Algorithm 18.23 (Sparsest cut)

1. Solve the dual LP (18.2) to obtain metric $(V, \boldsymbol{d})$.
2. Pick sets $S_i^j,\ 2 \leq i \leq l+1,\ 1 \leq j \leq N$, where set S_i^j is formed by picking each vertex of V' independently with probability $1/2^i$.
3. Obtain an ℓ_1-embedding of $(V, \boldsymbol{d})$ in $O(\log^2 k)$ dimensional space w.r.t. these sets.
4. Obtain an approximate cut packing for $(V, \boldsymbol{d})$ from the ℓ_1-embedding.
5. Output the sparsest cut used by the cut packing.

Theorem 18.24 *Algorithm 18.23 achieves an approximation guarantee of $O(\log k)$ for the sparsest cut problem.*

Corollary 18.25 *For a demands multicommodity flow instance with k source-sink pairs,*

$$\frac{1}{O(\log k)} \left(\min_{S \subset V} \frac{c(S)}{\operatorname{dem}(S)} \right) \leq \max_{\text{throughput } f} f \leq \min_{S \subset V} \frac{c(S)}{\operatorname{dem}(S)}.$$

Applications

We present below a number of applications of the sparsest cut problem.

Edge expansion

Expander graphs have numerous applications; for instance, see Example 17.11. We will obtain an $O(logn)$ factor algorithm for the problem of determining the edge expansion of a graph:

Problem 18.26 (Edge expansion) Given an undirected graph $G = (V, E)$, the *edge expansion of a set $S \subset V$* with $|S| \leq n/2$, is defined to be $|\delta(S)|$, i.e., the number of edges in the cut $(S, \overline{S})$. The problem is to find a minimum expansion set.

Consider the special case of demands multicommodity flow in which we have $n(n-1)/2$ distinct commodities, one for each pair of vertices. This is called the *uniform multicommodity flow problem.* For this problem, the sparsity of any cut $(S, \overline{S})$ is given by

$$\frac{c(S)}{|S||\overline{S}|}.$$

Let $(S, \overline{S})$, with $|S| \leq |\overline{S}|$, be the cut found by Algorithm 18.23 when run on G with uniform demands. Notice that $|\overline{S}|$ is known within a factor of 2, since $n/2 \leq |\overline{S}| \leq n$. So, S has expansion within an $O(\log n)$ factor of the minimum expansion set in G. Clearly, the generalization of this problem to arbitrary edge costs also has an $O(\log n)$ factor approximation algorithm.

Conductance

The conductance of a Markov chain characterizes its mixing rate, i.e., the number of steps needed to ensure that the probability distribution over states is sufficiently close to its stationary distribution. Let P be the transition matrix of a discrete-time Markov chain on a finite state space X, and let π denote the stationary probability distribution of this chain. We will assume that the chain is aperiodic, connected, and that it satisfies the detailed balance condition

$$\pi(x)P(x,y) = \pi(y)P(y,x) \quad \forall x, y \in X.$$

Define undirected graph $G = (X, E)$ on vertex set X; $(x, y) \in E$ iff $\pi(x)P(x,y) \neq 0$. The edge weights are defined to be $w(x,y) = \pi(x)P(x,y)$. The *conductance* of this chain is given by

$$\Phi = \min_{S \subset X, 0 < \pi(S) \leq 1/2} \frac{w(S, \overline{S})}{\pi(S)},$$

where $w(S, \overline{S})$ is the sum of weights of all edges in the cut $(S, \overline{S})$. For any set S, the numerator of the quotient defined above is the probability that the chain in equilibrium escapes from set S to $\overline{S}$ in one step. So the quotient gives the conditional probability of escape, given that the chain is initially in S, and Φ measures the ability of the chain to not get trapped in any small region of the state space.

Theorem 18.24 leads to an $O(\log n)$ factor approximation algorithm for computing conductance. First, observe that it suffices to approximate the following symmetrized variant of Φ:

$$\Phi' = \min_{S \subset X, 0 < \pi(S) \leq 1} \frac{w(S, \overline{S})}{\pi(S)\pi(\overline{S})}, \tag{18.3}$$

since Φ and Φ' are within a factor of 2 of each other (notice that if $0 < \pi(S) \leq 1/2$, then $1/2 \leq \pi(\overline{S}) < 1$).

Next, let us show that computing Φ' is really a special case of the sparsest cut problem. Consider graph $G = (X, E)$ with edge weights as defined above. For each pair of vertices $x, y \in X$, define a distinct commodity with a demand of $\pi(x)\pi(y)$. It is easy to see that the sparsity of a cut $(S, \overline{S})$ for this instance is simply the quotient defined in (18.3). Hence, the sparsity of the sparsest cut is Φ'.

Balanced cut

The following problem finds applications in partitioning problems, such as circuit partitioning in VLSI design. Furthermore, it can be used to perform the "divide" of a divide-and-conquer strategy on certain problems; for instance, see the algorithm for Problem 18.30 below.

Problem 18.27 (Minimum b-balanced cut) Given an undirected graph $G = (V, E)$ with non-negative edge costs, and a rational b, $0 < b \leq 1/2$, find a minimum capacity cut $(S, \overline{S})$ such that $bn \leq |S| < (1-b)n$.

A b-balanced cut for $b = 1/2$ is called a *bisection cut*, and the problem of finding a minimum capacity such cut is called the *minimum bisection problem*. No approximation algorithms are known for Problem 18.27. However, we can use Theorem 18.24 to obtain the following *pseudo-approximation algorithm*: we will find a 1/3-balanced cut whose capacity is within an $O(\log n)$ factor of the capacity of a minimum bisection cut.

For $V' \subset V$, let $G_{V'}$ denote the subgraph of G induced by V'. The algorithm is: Initialize $U \leftarrow \emptyset$ and $V' \leftarrow V$. Until $|U| \geq n/3$ do: find a minimum expansion set in $G_{V'}$, say W; $U \leftarrow U \cup W$ and $V' \leftarrow V' - W$. Finally, let $S \leftarrow U$, and output the cut $(S, V - S)$.

Claim 18.28 *The cut output be the algorithm is a 1/3-balanced cut whose capacity is within an $O(\log n)$ factor of the capacity of a minimum bisection cut in G.*

Proof : At the end of the penultimate iteration, $|U| < n/3$. So, at the beginning of the last iteration, $|V'| \geq 2n/3$. At most half of these vertices are added to U in the last iteration. Therefore, $|V - S| \geq n/3$, and so $n/3 \leq |S| < n/3$. Hence $(S, V - S)$ is a 1/3-balance cut.

Let $(T, \overline{T})$ be a minimum bisection cut in G. Since at the beginning of each iteration, $|V'| \geq 2n/3$, each of the sets $T \cap V'$ and $\overline{T} \cap V'$ has at least $n/6$ vertices. So, the expansion of a minimum expansion set in $G_{V'}$ in each iteration is at most $\frac{c(T)}{(n/6)}$. Since the algorithm finds a set having expansion within a factor of $O(\log n)$ of optimal, in any iteration, the set U found satisfies:

$$\frac{c(U)}{|U|} \leq O(\log n) \frac{c(T)}{n/6}.$$

Since the final set S has at most $2n/3$ vertices, summing up we get

$$c(S) \leq O(\log n) \frac{c(T)(2n/3)}{n/6},$$

thereby giving $c(S) \leq O(\log n) c(T)$. □

Exercise 18.29 Why can't the algorithm given above be converted to a true approximation algorithm, i.e., so that in the end, we compare the 1/3-balance cut found to the optimal 1/3-balance cut? Construct graphs for which the capacity of a minimum bisection cut is arbitrarily higher than that of a 1/3-balanced cut. Show that the algorithm given above extends to finding a b-balanced cut that is within an $O(\log n)$ factor of the best b'-balanced cut for $b \leq 1/3$ and $b < b'$. Where in the argument is the restriction $b \leq 1/3$ used?

Minimum cut linear arrangement

Problem 18.30 (Minimum cut linear arrangement) Given an undirected graph $G = (V, E)$ with non-negative edge costs, for a numbering of its vertices from 1 to n, define S_i to be the set of vertices numbered at most i, for $1 \leq i \leq n - 1$; this defines $n - 1$ cuts. The problem is to find a numbering that minimizes the capacity of the largest of these $n - 1$ cuts, i.e., it minimizes $\max\{c(S_i) | \ 1 \leq i \leq (n-1)\}$.

Using the pseudo-approximation algorithm obtained above for the 1/3-balanced cut problem, we will obtain a true $O(\log^2 n)$ factor approximation algorithm for this problem. A key observation is that in any arrangement, $S_{n/2}$ is a bisection cut, and so the capacity of a minimum bisection cut in G, say β, is a lower bound on the optimal arrangement. The reason we get a true approximation algorithm is that the 1/3-balanced cut algorithm compares the cut found to β.

The algorithm is recursive: Find a 1/3-balanced cut in G_V, say $(S, \overline{S})$, and recursively find a numbering of S in G_S using numbers from 1 to $|S|$, and a numbering of $\overline{S}$ in $G_{\overline{S}}$ using numbers from $|S| + 1$ to n. Of course, the recursion ends when the set is a singleton, in which case the prescribed number is assigned to this vertex.

Claim 18.31 *The algorithm given above achieves an $O(\log^2 n)$ factor for the minimum cut linear arrangement problem.*

Proof : The following binary tree T (not necessarily complete) encodes the outcomes of the recursive calls made by the algorithm: Each recursive call corresponds to a node of the tree. Suppose recursive call α ends with two further calls, α_1 and α_2, where the first call assigns smaller numbers and the second assigns larger numbers. Then, α_1 will be made the left child of α in T, and α_2 will be made the right child of α. If recursive call α was made with a singleton, then α will be a leaf of the tree.

To each non-leaf, we will assign the set of edges in the cut found during this call, and to each leaf we will assign its singleton vertex. Thus, the left to right ordering of leaves gives the numbering assigned by the algorithm to the vertices. Furthermore, the edge sets associated with non-leaf nodes define a partitioning of all edges of G. The cost of edges associated with any non-leaf is $O(\log n)\beta$ by Claim 18.28. Since each recursive call finds a 1/3-balanced cut, the depth of recursion, and hence the depth of T, is $O(\log n)$.

Following is a crucial observation: Consider any edge (u, v) in G. Let α be the lowest common ancestor of leaves corresponding to u and v in T. Then, (u, v) belongs to the set of edges associated with node α.

With respect to the numbering found by the algorithm, consider a cut $(S_i, \overline{S_i})$, $1 \le i \le n - 1$. Any edge in this cut connects vertices numbered j and k with $j \le i$ and $k \ge i + 1$. So, such an edge must be associated with a node that is a common ancestor of the leaves numbered i and $i+1$. Since the depth of T is $O(\log n)$, there are $O(\log n)$ such common ancestors. Since the cost of edges associated with any node in T is $O(\log n)\beta$, the cost of cut $(S_i, \overline{S_i})$ is bounded by $O(\log^2 n)\beta$. The claim follows, since we have already argued that β is a lower bound on the optimal arrangement. $\square$