Approximation Algorithms for Network Flow and Minimum Cut Problems

Calum MacRury

Master of Science

Mathematics and Statistics

McGill University

Montreal,Quebec

November 15, 2018

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Masters of Science

DEDICATION

In loving memory of Jacob Hope.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Professor Jakobson for his guidance, support and understanding throughout my time at McGill. I would also like to thank Professor Norin for his willingness to discuss various topics in graph theory with me. It was a pleasure working with both of you.

ABSTRACT

In this thesis, a number of optimization problems are presented from algorithmic graph theory. This includes the multi-commodity flow problem, whose motivation lies in the single source-sink pair maximum flow problem. Combinatorial and linear programming techniques are applied to these problems. In the case of a single source-sink pair, a polynomial time exact algorithm is reviewed, famously known as the Edmonds-Karp algorithm. A primal-dual algorithm is later analyzed for the case of multiple source-sink pairs, however it specifically deals with noncyclic graphs. The other main class of problems studied in this thesis are known as minimum cut problems. In particular, the single source-sink pair minimum cut problem is seen to have an exact algorithm. Generalizations of this problem are later analyzed, including the multiway cut problem and the multicut problem. Approximation algorithms are developed for both these problems, where randomized rounding algorithms from linear programming are the main focus.

ABRÉGÉ

Dans cette thèse, nous présentons plusieurs problèmes d'optimization qui viennent de théorie algorithmique des graphes. Notamment le problème de distribution multiple de flots dont la motivation provient du problème de flot maximum a source et puits uniques. Nous appliquons des techniques combinatoires et de programmation linéaire a ces problèmes. Pour le problème de flot maximum a source et puits uniques, il existe un algorithme qui calcule une solution exacte en temps polynomial, l'algorithme d'Edmonds-Karp. Nous analyserons aussi un algorithme de programmation linéaire dans le cas de sources et puits multiples, mais uniquement pour les graphes non cycliques. L'autre catégorie de problèmes que nous étudions dans cette these est celle des problèmes de coupes minimum. En particulier le problème de coupe minimum dans un graphe a source et puits unique admet un algorithme exact. Nous verrons plus tard des généralisations de ce problème.

TABLE OF CONTENTS

CHAPTER 0
Introduction

This thesis serves an as overview of a number of topics from algorithmic graph theory. In particular, it mainly focuses on developing algorithms for network flow and minimum cut problems. Broadly speaking, both classes of problems are optimization problems whose inputs contain edge weighted graphs, where the interpretation of these weights depends on the problem at hand.

In the case of a minimum cut problem, the goal is typically to separate a collection of vertex pairs in a graph, by removing a number of edges. The cost of an edge subset is defined as the sum of its weights, and an exact solution to the problem involves finding such a subset of minimum cost.

On the other hand, a network flow problem involves routing an abstract flow commodity between pairs of vertices. This flow is routed using the paths of the graph, and is subjected to a number of constraints. These restrictions are derived from the edge weights of the graph, where the amount of flow passing through an edge cannot exceed its capacity (weight). An exact solution to this problem therefore involves maximizing the amount of flow passing through the graph.

The majority of the problems covered by the thesis are computationally challenging. That is, they are unlikely to have polynomial time algorithms which generate exact solutions. As consequence of these restrictions, the thesis is mostly concerned with algorithms which generate approximate solutions in polynomial

time. The performance of such an algorithm is then determined by how closely it approximates exact solutions. Algorithms with these properties are referred to as approximation algorithms.

The first chapter of the thesis serves as an overview for some elementary concepts from complexity theory. This includes a discussion of various complexity classes, as well as formal definitions of optimization problems and approximation algorithms. The last section of the chapter includes a review of some basic concepts from linear programming. This includes material from duality theory, as well as definitions of integrality gaps and rounding algorithms. The material from this chapter is mainly adapted from the book "Computational Complexity" by Arora and Barak [AB16], and the journal article "a Short Guide to Approximation Preserving Reductions" by Crescenzi [Cp97]. That being said, the books "Approximation Algorithms" by Vazirani [Vv11], and "the Design of Approximation Algorithms" by Williamson, and Shmoys [WS11] were also used.

The second chapter introduces the network flow problem for a single vertex pair. In particular, a number of frameworks for this problem are developed, which are later shown to be equivalent. In the later sections, exact algorithms are developed to solve this problem, most notably the Edmonds-Karp algorithm which appeared in the 1972 journal article by Edmonds and Karp [EK72]. The chapter concludes with an overview of the maximum-flow minimum-cut theorem, together with some applications of the Edmonds-Karp algorithm. The book "Algorithm Design" by Kleinberg and Tardos [KT14] is used as the main source in this chapter.

In the next chapter, the multiway cut problem for graphs is studied extensively. This problem is a generalization of the minimum cut problem, and is known to be computationally challenging; that is, **NP**-hard. A variety of combinatorial and linear programming based algorithms are developed throughout the chapter. The effectiveness of these algorithms are compared and contrasted, with performance guarantees being the primary focus. The material of this chapter is based on the books "Approximation Algorithms" [Vv11] and "the Design of Approximation Algorithms" [WS11].

The final chapter of the thesis primarily studies a further generalization of the multiway cut problem, known as the multicut problem. This problem is first examined on trees, and is shown to be **NP**-hard even for trees of height one. The dual of this problem is shown to be a maximum flow type problem, in which flow is routed through many pairs of vertices concurrently. A primal-dual algorithm is then reviewed from the book "Approximation Algorithms" [Vv11], and shown to simultaneously approximate both problems. The chapter ends with the development of rounding based algorithms for solving the multicut problem on general graphs. This includes a number of randomized algorithms, together with a deterministic algorithm adapted from "Approximation Algorithms" [Vv11] and "the Design of Approximation Algorithms" [WS11].

CHAPTER 1
Review of Relevant Topics

This chapter contains a brief review of a number of important topics from linear programming and complexity theory. As the main goal of the thesis is to examine approximation algorithms on graphs, the primarily focus is on the theory which serves as background to this area. The majority of the examples throughout this chapter pertain to problems on graphs. In particular, the vertex cover problem is used as a recurring example, as it is a simple problem which highlights many of the topics discussed.

The first section involves a review of decision problems and some common complexity classes. It also introduces the notion of **NP**-hardness of a problem, where the vertex cover decision problem is seen as a prime example. The majority of this section is based upon the book "Computational Complexity" by Arora and Barak [AB16].

In the following section, the basic theory surrounding optimization problems is introduced. This includes a formal definition of **NP**-optimization problems, as well as a definition of what it means for such a problem to be **NP**-hard. Later on, both deterministic and randomized approximation algorithms are introduced for optimization problems which are **NP**-hard, and their performance guarantees are discussed. This includes a definition of randomized algorithms based on general probability spaces, which I designed to fit the applications of the thesis. The

section concludes with a review of approximation preserving reductions. Most of the material of this section is adapted from the book "Approximation Algorithms" by Vazirani [Vv11], and the journal article "A Short Guide to Approximation Preserving Reductions" by Crescenzi [Cp97].

The final section of this chapter reviews a number of standard linear programming concepts. In particular, the encodability of an optimization problem as an integer program (IP) is discussed, together with its relaxation as a linear program (LP). The integrality gap of an encoding is discussed at length, where I have provided mathematically precise defintions of this concept. The section also discusses the solvability of linear programs using polynomial time separation oracles. This includes a general discussion of LP rounding algorithms, and their relations to integrality gaps. The section concludes with a discussion of LP duality, and a proposition involving the complementary slackness conditions of a linear program. The book "the Design of Approximation Algorithms" by Williamson and Shmoys [WS11] is used as a resource, though this is primarily cited in the context of polynomial time separation oracles.

1.1 Decision Problems and Complexity Classes

In this section, we begin by introducing the notion of a decision problem, and the complexity classes **P** and **NP**. From here, we consider the notion of a polynomial time reduction. This naturally leads us to consider what it means for a decision problem to be **NP**-hard. We conclude by reviewing the widely believed conjecture that $\mathbf{P} \neq \mathbf{NP}$.

In broad terms, a *decision problem* is a type of problem in which the answer is always "yes" or "no". One of the most famously known decision problems is the *boolean satisfiability problem*. In this problem, we are passed an arbitrary *boolean function* $f : \{0,1\}^k \to \{0,1\}$, where $k \geq 1$. The function f is referred to as an *instance* of the problem. The goal is to determine whether or not there exists an assignment $s \in \{0,1\}^k$, for which $f(s)$ is equal to one. If such an assignment exists, f is called *satisfiable*, and is classified as a "yes" instance. Otherwise, f is referred to as *nonsatisfiable*, and is classified as a "no" instance.

We now introduce a number of concrete examples which make use of some basic concepts from graph theory. Suppose that we are given an *undirected graph* $G = (V, E)$. The set V forms the *vertices* of G, whereas the set E consists of the *undirected edges* of G. As G is *undirected*, each member of E is a set of the form $\{u, v\}$ for $u, v \in V$, where $u \neq v$. If $e \in E$, then we refer to e as an *undirected* edge, and the *ends* of e as the vertices it contains. When we work specifically with undirected graphs, we typically refer to G as a *graph*, and the members of E as simply *edges*.

A fundamental concept in graph theory is the notion of *connectivity*. Let us suppose that $G = (V, E)$ is an undirected graph. Moreover, suppose we are given vertices $v_1, \ldots, v_l$ in G, for which $\{v_i, v_{i+1}\} \in E$ for $i = 1, \ldots, l-1$, where $l \geq 2$. In this case, we say that these vertices form a *undirected path* betweeen v_1 and v_l, and refer to v_1 and v_l as being *connected*. The graph G itself is said to be *connected*, provided any two vertices are connected (that is, a path between them exists). The problem of determining whether or not a graph is connected is an example of a decision problem.

Problem 1.1.1 (Graph Connectivity Problem)**.** *Suppose we are given an undirected graph G as an instance of the problem. We may ask may the question: Is the graph G connected? If the answer is yes, then G is a "yes" instance. If the answer is no, then G is a "no" instance.*

If we are passed a graph $G = (V, E)$, then it is often useful to introduce a *weight function* $w : V \to \mathbb{Q}_{\geq 0}$, which can be thought of as associating a nonnegative rational weight to each vertex of G. Let us suppose that U is a subset of vertices of G, where for each edge e in E, at least one of the ends of e lies in U. In this case, we refer to U as a *vertex cover* of G. Moreover, the weight of U, denoted $w(U)$, is defined as the sum of the weights of its vertices. Rather,

$$w(U) := \sum_{v \in U} w(v).$$

Consider the following problem, known as the *vertex cover decision problem*:

Problem 1.1.2 (Vertex Cover Decision Problem)**.** *Suppose we are passed a rational number $q \in \mathbb{Q}_{\geq 0}$, an undirected graph $G = (V, E)$ with a weight function*

$w : E \to \mathbb{Q}_{\geq 0}$. In this case, the tuple (G, w, q) forms an instance of the problem. We may ask the question: Does G have a vertex cover of weight less than or equal to q? If the answer is yes, then (G, w, q) is a "yes" instance. Otherwise, (G, w, q) is a "no" instance.

It is useful to introduce a formal mathematical object which we can use to model each of these specific decision problems. Observe that in each of the problems listed, the problem instances can be encoded using finite strings of binary numbers. This is why when we considered the vertex cover decision problem, we ensured that our weight function took on rational numbers, as opposed to arbitrary real numbers. While there may be multiple ways to encode a problem using binary strings, the significance is that at least one encoding exists.

Let us now introduce our mathematical definition for decision problems. We first define $\{0,1\}^*$ to be the set of all $\{0,1\}$ strings of finite length. In light of the preceding observations, we can identify each decision problem as a subset of $\{0,1\}^*$. More specifically, let us suppose $L \subseteq \{0,1\}^*$. We refer to each string $x \in \{0,1\}^*$ as *problem instance*. If x is a member of L, then x is referred to as a *"yes" instance*. Otherwise, x is referred to as a *"no" instance*. Thus, the set L indicates which problem instances have an affirmative answer. We refer to L as a *language*, or as an *abstract decision problem*, where either term is used interchangeably. That being said, the second term is nonstandard. Typically the term "decision problem" is used instead to describe L. We precede this term with the word "abstract", as it helps to differentiable L from the informal notion of a decision problem.

As an example, observe how we can use an abstract decision problem to model Problem 1.1.2:

Example 1.1.1 (Vertex Cover Decision Problem)**.** *Let us suppose that* $q \in \mathbb{Q}_{\geq 0}$*, and* $G = (V, E)$ *is a graph with weight function* $w : V \to \mathbb{Q}_{\geq 0}$*. We denote* $(G, w, q)_b$ *as a* $\{0, 1\}$ *encoding of the problem instance* (G, w, q)*. We may now model Problem 1.1.2 using the language* L*, where*

$$L := \{(G, w, q)_b : G \text{ has a vertex cover } U \text{ with } w(U) \leq q\}.$$

Observe that since there are many different ways to encode graphs, there are many different languages which serve as models for Problem 1.1.2.

Now that we have an abstract model for decision problems, we may consider a natural question associated with them: If L is an arbitrary language, and $x \in \{0, 1\}^*$, can we compute whether x is a member of L in a finite amount of time? In other words, we wish to know whether there exists an *algorithm* $\mathcal{A}$, which given the input x, ouputs a *correct* $\{0, 1\}$ classification in a finite number of steps. That is, if $\mathcal{A}(x)$ denotes the value outputed by the algorithm, then $\mathcal{A}(x)$ should be one, if and only if $x \in L$ (x is a "yes" instance).

While we have an intuitive notion of what an algorithm is, there exist a number of computational models that allow us to formalize this intuition. In this text, we focus specifically on the *Turing machine model*. We assume that the reader is familiar with this construction, as well as how to identify algorithms with Turing machines. In particular, we assume familiartiy with the Church-Turing

thesis. The majority of the following definitions and results will be stated using this terminology.

Let us now return to the problem of classifying the members of the language L. We wish to find a *deterministic Turing machine* $\mathcal{M}$, for which if $x \in \{0,1\}^*$ is present in the *input tape* of $\mathcal{M}$, then after a finite number of steps, $\mathcal{M}$ will *terminate* with the correct $\{0,1\}$ classification of x in its output tape. Rather, if $\mathcal{M}(x)$ denotes the string present in the output tape at this point, then $\mathcal{M}(x) = 1$, if and only if $x \in L$. If the Turing machine $\mathcal{M}$ has this property, then we say that $\mathcal{M}$ *decides* the language L. Moreover, we refer to the abstract decision problem L as *decidable*.

It is clear that all of the decision problems we have introduced so far are decidable (when encoded as languages of $\{0,1\}^*$). This is easily seen, as there are simple algorithms which solve each of them. The Church-Turing thesis ensures that we can then encode each of these algorithms as Turing machines.

We now introduce a definition which allows us to measure how efficiently a given Turing machine operates:

Definition 1.1.2. *Suppose $\mathcal{M}$ is a Turing machine and $\tau : \mathbb{N} \to \mathbb{N}$. Moreover, for each $x \in \{0,1\}^*$, $\mathcal{M}$ returns the string $\mathcal{M}(x)$ in at most $\varepsilon\,\tau(|x|)$ steps, where $|x|$ is the length of x, and $\varepsilon > 0$ is a fixed constant. In this case, we say that the Turing machine $\mathcal{M}$ operates (executes) in time τ, or that $\mathcal{M}$ is a τ-time Turing Machine.*

We can relate Definition 1.1.2 to the decidability of languages as follows: Let L be an arbitrary language, and $\mathcal{M}$ be any Turing machine which decides L. If

$\tau : \mathbb{N} \to \mathbb{N}$, then we say that $\mathcal{M}$ *decides* L *in time* τ, provided $\mathcal{M}$ operates in time τ.

Using this definition, we can classify decision problems based on how efficiently they can be solved. Of particular importance is the complexity class **P**, which is defined as the set of all languages which are decidable in polynomial time. Rather, if L is an arbitrary language, then $L \in \mathbf{P}$, provided there exists a Turing Machine $\mathcal{M}$, and a polynomial $p : \mathbb{N} \to \mathbb{N}$, which decides L in time p (as in Definition 1.1.2). Observe that the graph connectivity problem (Problem 1.1.1) is a member of **P**.

Example 1.1.3. *Given a graph $G = (V, E)$ and a vertex $v \in V$, we may execute a depth-first-search about v, yielding a subset of vertices $U \subseteq V$. If G is connected, then U will contain all the vertices of G. Otherwise, U will be a strict subset of V. As this depth-first-search can be done in time $O(|V| + |E|)$, we have a simple algorithm for deciding connectivity in polynomial time.*

We also define the complexity class **NP**, which is rougly speaking, the set all of languages which can be *verified efficiently*. Formally, if L is an arbitrary language, then we include L in **NP**, provided there exists a *polynomial time* Turing machine $\mathcal{M}$ which satisfies the properties listed below. We assume that x is an arbitrary member of $\{0, 1\}^*$:

- There exists a polynomial p, for which if $x \in L$, then there is some string $u \in \{0, 1\}^*$ with $|u| \leq p(|x|)$, such that the output after passing (x, u) to $\mathcal{M}$ is one. That is, $\mathcal{M}(x, u) = 1$ in the notation introduced earlier. In this case, we refer to u as a *certificate* for x.

- If $x \notin L$, then for every $y \in \{0,1\}^*$ with $|y| \leq p(x)$, the output after passing (x, u) to $\mathcal{M}$ is zero; that is, $\mathcal{M}(x, y) = 0$.

If such a Turing machine $\mathcal{M}$ exists, then we refer to $\mathcal{M}$ as a *verifier* for L, and include L in the class **NP**.

It is clear that $\mathbf{P} \subseteq \mathbf{NP}$. To see this, observe that if L is decidable by $\mathcal{M}$ in polynomial time, then we can take p to be the zero polynomial whereas $\mathcal{M}$ can be used as the verifier for L. In particular, the empty string serves as a certificate for strings contained in L.

As a result of this observation, the graph connectivity problem is a member of **NP** by the work done in Example 1.1.3. We now show that the vertex cover decision problem (Problem 1.1.2) is a nontrivial member of **NP**.

Example 1.1.4. *Let us suppose that $q \geq 0$ is a rational number, and G is a graph with vertex weight $w : V \to \mathbb{Q}_{\geq 0}$. The tuple (G, w, q) forms a problem instance of the vertex cover decision problem. Our verification algorithm works as follows: Given a subset of vertices $U \subseteq V$, first check whether each edge of G is covered by U. If this is not the case, then U is not a valid vertex cover, so we reject U and return the output zero. Otherwise, we acknowledge U as a vertex cover, and compute its weight, namely, $w(U)$. We next compare $w(U)$ with the rational number q. If $w(U) < q$, then we once again reject U, and return the ouput zero. On the other hand, if $w(U) \leq q$, then we accept U, and return the output one. In this case, U serves as a certificate for the problem instance (G, w, q).*

It is clear that this algorithm can be implemented in time $O(|E| + |V|)$. Moreover, the size of U is at most $O(|V|)$, which is polynomial in the size of G.

The algorithm is therefore a verifier for the vertex cover decision problem, thus proving the problem's membership in ***NP****.*

1.1.1 NP-hardness and the Cook-Levin Theorem

We conclude the section by introducing the notion of **NP**-hardness. Intuitively, this is a means to characterize the most computationally difficult problems present in the class **NP**. In order to formalize this notion, we introduce the concept of *polynomial time reductions* between languages.

Definition 1.1.5. *Given languages L_1 and L_2, we say that L_1 reduces to L_2, denoted $L_1 \leq L_2$, provided there exists a Turing machine $\mathcal{M}$, with the properties listed below. We assume now that $x \in \{0,1\}^*$:*

- *Given x as input, $\mathcal{M}$ outputs a string $y \in \{0,1\}^*$, such that $x \in L_1$, if and only if $y \in L_2$. We denote this string $\mathcal{M}$ for convenience.*
- *There exists a polynomial $p : \mathbb{N} \to \mathbb{N}$, such that $\mathcal{M}$ operates in time p. In other words, $\mathcal{M}$ is a polynomial time algorithm.*

We remark that unlike the preceding definitions, the Turing machine $\mathcal{M}$ in Definition 1.1.5 may write nontrivial strings to its output tape. Moreover, we observe that since $\mathcal{M}$ can only write one character at a time to its output tape,

$$|\mathcal{M}(x)| \leq p(|x|)$$

for each $x \in \{0,1\}^*$, as $\mathcal{M}$ operates in time p. In light of this observation, the claims below follow:

Proposition 1.1.6. *Suppose we are given languages L_1 and L_2, for which L_1 is reducible to L_2; that is, $L_1 \leq L_2$. In this case, if L_2 is decidable in polynomial time, then so is L_1. Similarly, if L_2 if a member of **NP**, then so is L_1.*

*If we consider $\leq$ to be a relation on **NP**, then it is clear that $\leq$ is transitive. Rather, if $L_1, L_2, L_3 \in$ **NP**, then $L_1 \leq L_2$ and $L_2 \leq L_3$, imply that $L_1 \leq L_3$.*

To conclude, we define a language $L \subseteq \{0,1\}^*$ to be **NP**-*hard*, provided every language in the class **NP** reduces to L. If we also know that the language L is a member of **NP**, then we refer to L as **NP**-*complete*. This turms out to be a nontrivial definition, as there exist many languages which are known to be **NP**-complete. That being said, the proof of the existence of such a language is a nontrivial result. This existence was first proven in Cook-Levin Theorem, which showed that the boolean satisfiability problem is **NP**-complete.

Theorem 1.1.7 (Cook-Levin Theorem)**.** *The boolean satisfiability problem is **NP**-complete.*

Moreover, there is a simple polynomial time reduction between the boolean satisfiability problem, and the vertex cover decision problem (when encoded as languages). Thus, as a result of Proposition 1.1.6, we also have the following corollary:

Corollary 1.1.8. *The vertex cover decision problem (Problem 1.1.2) is **NP**-complete.*

We may think of the **NP**-complete languages of **NP** as the hardest problems to decide within this complexity class. It it widely believed that **NP**-complete

problems *cannot* be solved in polynomial time. This belief is equivalent to the famous conjecture that $\mathbf{P} \neq \mathbf{NP}$.

Conjecture 1.1.1. *The complexity class* $\boldsymbol{P}$ *is strictly contained in* $\boldsymbol{NP}$.

1.2 Optimization Problems and Approximation Algorithms

In this section, we consider a class of problems known as optimization problems. The majority of the problems encountered in the thesis are of this form, so it is essential to understand how they are characterized. Specifically, we provide both formal and informal definitions of what optimization problems are. We then discuss what it means for optimization problems to be **NP**-hard, and describe how this affects our ability to solve such problems exactly. This naturally leads us to define approximation algorithms, and to consider their various properties. In particular, we look at the performance guarantees, correctness and the efficiency of such algorithms.

Informally speaking, an optimization problem Π contains a number of *problem instances*, each of which has its own collection of *feasible solutions*. These feasible solutions each have their own nonnegative rational numbers associated to them, which are referred to as their *values*. The goal of the problem is to find a feasible solution of *optimum value*. More specifically, if the optimization problem is classified as a *minimization problem*, then the goal is find a a feasible solution of *minimum value*. If it is a *maximization problem*, then the goal is find a feasible solution of *maximum value*.

In order to demonstrate these concepts, we return to the *vertex cover problem* on graphs, denoted Π. As in the decision version variant introduced in the previous section (see Problem 1.1.2), we are given an undirected graph $G = (V, E)$, with a weight function $w : V \to \mathbb{Q}_{\geq 0}$. Each such pair (G, w) makes up an instance

of the problem Π. We denote the set of all such instances by D_Π; that is,

$$D_\Pi := \{(G, w) : G = (V, E) \text{ an undirected graph and } w : V \to \mathbb{Q}_{\geq 0}\}.$$

If we fix an instance (G, w) of D_Π, then a feasible solution to (G, w), is a vertex cover U of G. In this way, we may denote $S_\Pi(I)$ as the collection of all vertex covers of G, where $I := (G, w)$. For each vertex cover U of G, we associate a nonnegative rational number to it, denoted $w(U)$. This number is defined to be the sum of the weights of the vertices of U. Rather,

$$w(U) := \sum_{v \in U} w(u).$$

As in our informal definition of optimization problems, $w(U)$ is the *value* associated to U. Finally, the problem is a minimization problem. That is, given (G, w), we wish to find a vertex cover U of G that is of *minimum value*. In the above notation,

$$w(U) = \min_{U' \in S_\Pi(G,w)} w(U').$$

We summarize this problem below for future reference:

Problem 1.2.1 (Vertex Cover Problem)**.** *Given an undirected graph $G = (V, E)$ with weight function $w : V \to \mathbb{Q}_{\geq 0}$, we define a vertex cover U of G, to be a subset $U \subseteq V$, where each edge of G has an end in U. We define the value or weight of U, to be the sum of the weights of its vertices. Rather,*

$$w(U) := \sum_{v \in U} w(v).$$

The goal of the problem is to ensure that U has minimum value. That is,

$$w(U) = \min\{w(U') : U' \text{ a vertex cover of } G\},$$

If U has this property, then it is an optimum solution to the problem instance (G, w).

Now that we have an idea of what optimzation problems are, we provide a formal set-theoretic definition that generalizes the above example, while incorporating the informal properties outlined at the beginning of the section. The definition is fairly technical, implicitly citing a number of notions from complexity theory. It can be skipped if the reader is already comfortable with their understanding of optimization problems.

An *optimization problem* Π is a four-tuple $(D, S, \text{obj}, \text{type})$, for which the properties below hold. We remark that Π is often referred to as a **NP**-*optimization problem*, though we typically drop this prefix when the context is clear.

1. D is a set consisting of the *problem instances* of Π. We require that the members of D are *encodable* as binary strings. Once an encoding is fixed, we define the size of I, denoted $|I|$, to be the number of characters (bits) required to represent I. Moreover, it must be possible to check whether an instance belongs to D in polynomial time.
2. Given an instance I of D, $S(I)$ denotes the set of *feasible solutions* to I. Once again, the collection of all possible feasible solutions, namely $\bigcup_{I \in D} S(I)$, must be encodable as binary strings. More, we require that the

size of these solutions be polynomial in the size of I. That is, there exists a fixed polynomial p, such that for each $s \in S(I)$, $|s| \leq p(|I|)$. Finally, there is a polynomial time algorithm, which given a pair (I, s), decides whether $s \in S(I)$.

3. The term obj is a function that assigns a nonnegative rational number to each pair (I, s), where $I \in D$ and $s \in S(I)$. It is typcially referred to as the *objective function* of problem Π. We require that it is *computable in polynomial time*. Rather, there exists an algorithm $\mathcal{A}$ and a polynomial q, which given (s, I), computes $\text{obj}(s, I)$ in at most $q(|(s, I)|)$ many steps.
4. The final term of Π is simply an element of set $\{\min, \max\}$; that is, $\text{type} \in \{\min, \max\}$. We use it to indicate whether Π is a minimization or maximization problem.

Given an instance $I \in D$, we say that $s \in S(I)$ is an *optimum solution* for I, provided,

$$\text{opt}(I, s) = \text{type}_{s' \in S(I)} \text{opt}(I, s').$$

The goal of the optimization problem is to find a feasible solution which is *optimum*, given an arbtirary problem instance. Moreover, we typically denote D,S, and opt with the subscript "Π", to avoid overlaps in notation in other contexts. Rather, these symbols are replaced by D_Π, S_Π and opt_Π when necessary.

Before we continue, we clarify some of the technical assumptions implicit in the above definition. Firstly, the notion of *encodability* of the problem instances can be taken to mean that there exists a bijection ϕ from D into $\{0, 1\}^*$. When we then make statements involving the size of an instance I of D, this can defined

to be the number of characters in $\phi(I)$. A similar definition suffices for the case of the total collection of feasible solutions; rather $\bigcup_{I \in D} S(I)$. We shall see later in the section how these algorithmic assumptions explain why we refer to Π as a **NP**-*optimization problem*.

We may state the algorithmic requirements of this definition in terms of Turing machines. In particular, the first requirement above effectively says that the language D (when encoded as binary strings) is decidable in polynomial time (a similar claim is true for the second requirement). The third requirement may be viewed as enforcing the efficiency at which the objective value can be computed (see Definition 1.1.2 in the previous section).

Now that we have a formal definition of optimization problems, we consider a subclass of them known as **NP**-*hard optimization problems*. To understand this classification, first recall the general goal of an optimization problem Π. We are given an instance of Π, and asked to find an optimum solution for the instance. That is, we wish to find an algorithm $\mathcal{A}$, which given an instance $I \in D_\Pi$, computes a feasible solution $s \in S(I)$ of optimum value. We refer to such an algorithm as an *exact algorithm* for Π.

While it is typically possible to find exact algorithms for optimization problems, we of course are most interested in ones which perform efficiently. More specifically, we wish to find algorithms which operate in *polynomial time*, and yet are also exact. In turns out that for the majority of the problems we consider, finding algorithms with both of these properties is most likely not an achieveable goal. This is because the existence of such an algorithm would allow us

to conclude that **P**=**NP**; a statement that is most likely false. As a example of an optimization problem which exhibits these shortcomings, we once again consider the vertex cover problem:

Example 1.2.1. *We consider a hypothetical situation in which an exact polynomial time algorithm for Problem 1.2.1 exists.*

Let $\mathcal{A}$ be an algorithm for the vertex cover problem, and p be a polynomial. We assume that for each graph $G = (V, E)$ with vertex weight $w : V \to \mathbb{Q}_{\geq 0}$, $\mathcal{A}$ returns an optimum vertex cover U of G. Moreover, this computation takes at most, $p(|G| + |w|)$ many steps, where $|G|$ is the size of G and $|w|$ is the size of w.

*Observe how algorithm $\mathcal{A}$ can be used to solve the vertex cover decision problem (Problem 1.1.2). Recall that an instance of this problem involves a rational number q, together with a graph $G = (V, E)$ and a vertex weight $w : V \to \mathbb{Q}_{\geq 0}$. We may first use $\mathcal{A}$ to compute a vertex cover U of G, for which $w(U)$ is minimum. We then compare $w(U)$ with q, and classify (G, w, q) as a "yes" instance, if and only if $w(U) \leq q$. Since U was a minimum weight vertex cover, this algorithm will correctly decide the problem instance in polynomial time. As consequence, the vertex cover decision problem must be a member of **P**. That being said, this problem is **NP**-complete (see Corollary 1.1.8), so we may conclude that **P** = **NP**.*

The main observation used in this example is that decision problem version of vertex cut is **NP**-hard. It turns out that regardless of which optimization problem Π we are given, we can always relate a decision problem of this kind to Π. To see this, consider the abstract decision problem L_Π defined as follows:

For each instance $I \in D_{\Pi}$ and $q \in \mathbb{Q}_{\geq 0}$, we add the instance (I, q) to L_{Π}, if and only if there exists some $s \in S(I)$ for which,

$$\mathrm{opt}_{\Pi}(s, I) \leq q,$$

assuming Π is a minimization problem (otherwise, the inequality is reversed). We typically refer to L_{Π} as the *decision problem version* of Π. A detailed examination of our definition of Π as a **NP**-hard optimization, should convince the reader that the language L_{Π} is a member of the complexity class **NP**. This should help explain the use of some of the technical assumptions regarding **NP**-optimization problems, as well as why the term is preceded by **NP**. The construction also allows us to extend the notion of **NP**-*hardness* from abstract decision problems to **NP**-optimization problems.

Definition 1.2.2. *We say that an* ***NP****-optimization problem* Π *is* ***NP****-hard, provided the decision problem* L_{Π} *is* ***NP****-hard.*

We may mimic the work done in Example 1.2.1 to yield the following proposition:

Proposition 1.2.3. *Assuming* $\boldsymbol{P \neq NP}$*, there does not exist an* ***NP****-hard optimization problem which has an exact polynomial time algorithm.*

In light of this proposition, we focus on finding algorithms which don't always return optimum solutions, yet are guaranteed to run efficiently. That is, in polynomial time, they return solutions to problem instances which are feasible, but not necessarily optimum. Broadly speaking, algorithms of this nature are known

as *approximation algorithms*. The majority of the thesis is focused on finding algorithms of this kind for **NP**-hard optimization problems.

Let us suppose that Π is an **NP**-hard optimization problem, for which we are given an *approximation algorithm* $\mathcal{A}$. We take this to mean that $\mathcal{A}$ runs in polynomial time, and always returns solutions which are feasible; that is, $\mathcal{A}$ is *correct*. In order to assess the performance of the algorithm, we would like to somehow measure the quality of the solutions it returns. As $\mathcal{A}$ cannot be exact, these solutions will not always be optimum. Instead, we attempt to make them as *close* to optimum as possible.

Assume for now that Π is a minimization problem, and that $\mathcal{A}$ is an approximation algorithm for it. We also assume that there exists some real number $\alpha > 0$, such that for all $I \in D_\Pi$,

$$\mathrm{opt}_\Pi(I, \mathcal{A}(I)) \leq \alpha \, OPT_\Pi(I),$$

where $OPT_\Pi(I)$ is the value of an optimum solution of I, and $\mathcal{A}(I)$ is the feasible solution of I returned by $\mathcal{A}$. In this case, we say that algorithm $\mathcal{A}$ has a *performance gurantee of* α, or that it achieves an *approximation guarantee of* α. Clearly, $\alpha \geq 1$, as

$$\mathrm{opt}_\Pi(I, \mathcal{A}(I)) \leq OPT_\Pi(I)$$

for all $I \in D_\Pi$.

On the other hand, if we assume that Π is a *maximization* problem, then we say that $\mathcal{A}$ has a *performance guarantee of* α. provided,

$$\text{opt}_{\Pi}(I, \mathcal{A}(I)) \geq \alpha \, OPT_{\Pi}(I)$$

for all $I \in \Pi$. In this case, it is clear that $\alpha \leq 1$, as

$$\text{opt}_{\Pi}(I, \mathcal{A}(I)) \geq OPT_{\Pi}(I)$$

for all $I \in D_{\Pi}$. Approximation algorithms which achieve performance guarantees as close to one as possible are of course the most desirable.

Before we continue, we make a remark regarding this definition. Observe that we can generalize the above definitions by allowing α to be function of the size of the input I. That is, $\alpha : \mathbb{N} \to \mathbb{R}$, such that $\alpha(n) \geq 1$ for all $n \in \mathbb{N}$, in the case of minimization problems, and $\alpha(n) \leq 1$ for all $n \in \mathbb{N}$, in the case of maximization problems. We shall see later in the thesis that sometimes performance guarantees of this kind are the best we can hope for.

We now show that there exists an approximation algorithm with a performance guarantee of two for the *cardinality vertex cover problem*. This problem is a special case of the vertex cover problem, in which all of the graphs we consider have weight functions which are identically one. That is, if $G = (V, E)$ is a graph with weight function $w : V \to \mathbb{Q}_{\geq 0}$, then $w(v) = 1$ for all $v \in V$. As all the weight functions have this property, we may assume that the cardinality vertex cover problem has only undirected graphs as its problem instances. Moreover, the value

(weight) of a vertex cover is simply the *cardinality* of the set. An optimum solution is therefore a vertex cover of minimum size.

Let us now describe how this algorithm operates. Suppose that we are passed a graph $G = (V, E)$. We begin by computing a *matching* M of G, which is a subset of edges of E which are mutually disjoint (no two edges of M share a vertex). We also ensure that M is *maximal*, thus implying that each edge $e \in E$ has a nontrivial intersection with at least one edge of M. Once we have computed this maximal matching M, we collect the vertices present in the edges of M, and return them as our output U.

Algorithm 1.2.1 Greedy Cardinality Vertex Cover Algorithm

Let $G = (V, E)$ be an undirected graph.
1: Initialize $M \leftarrow \emptyset$.
2: Initialize $U \leftarrow \emptyset$.
3: **for** each edge $e = \{u, v\} \in E$ **do**
4: **if** e is disjoint from all members of M **then**
5: Add e to M.
6: Add u and v to U.
7: Return U

It is clear that this algorithm runs in polynomial time. Moreover, the set U is a vertex cover, so the algorithm is correct. This is easily seen, for if $e \in E$, then e has a nontrivial intersection with a member of M, by the maximality of M. Since we add all the vertices which are present in the edges of M, this observation implies that U must be a vertex cover for G. We now prove a bound on the performance guarantee of the algorithm.

Proposition 1.2.4. *Algorithm 1.2.1 achieves an approximation guarantee of two for the cardinality vertex cover problem.*

Proof. Let us suppose we are given an arbitrary graph $G = (V, E)$, for which the algorithm computes the matching M and the vertex cover U. We denote the term $OPT(G)$ to be the size of a minimum vertex cover of G. Observe that,

$$|M| \leq OPT(G),$$

rather the number of edges in M is less than or equal to the value of $OPT(G)$. This is because the edges of M are disjoint, and so require at least M vertices of G in order to cover them all.

On the other hand, we know that $|U| = 2\,|M|$, by construction. Thus,

$$|U| \leq 2\,|M| \leq 2\,OPT(G).$$

As this inequality is true for every graph G, we know that the algorithm achieves an approximation guarantee of two. The claim thus follows.

□

It will also sometimes be useful to incorporate *randomness* into the approxmiation algorithms we design. We say that $\mathcal{A}$ is a *randomized approximation algorithm for optimization problem* Π, provided for each fixed $I \in D_\Pi$, the output $\mathcal{A}(I)$ forms a *random element* taking values in $S_\Pi(I)$. That is, there exists some *probability space* $(\Omega, \mathcal{B}, \mathbb{P})$, such that for each $I \in D_\Pi$, $\mathcal{A}(I)$ is a *measurable* map from Ω into $S_\Pi(I)$; rather, $\mathcal{A}(I) : \Omega \to S_\Pi(I)$. Observe that since we place a restriction on the size of the members of $S_\Pi(I)$ in terms of $|I|$, the size of $S_\Pi(I)$ is itself finite. In particular, we can associate a trivial *measure space* to $S_\Pi(I)$.

While the solutions returned by the algorithm are random, we typically require that $\mathcal{A}$ deterministically operates in polynomial time. That is, there is some polynomial p, such that for all $I \in D_{\Pi}$ and $\omega \in \Omega$, the algorithm takes at most $p(|I|)$ many steps to return the output $\mathcal{A}(I)(\omega)$.

As in the case of deterministic approximation algorithms, we devise performance guarantees for randomized approximation algorithms as well. This is done by comparing the value $OPT_{\Pi}(I)$ with the *expected value* of the random solution $\mathcal{A}(I)$ for each $I \in D_{\Pi}$. If Π is a minimization problem, then we say that $\mathcal{A}$ achieves a performance guarantee of $\alpha \geq 1$, provided

$$\mathbb{E}\,\mathrm{opt}_{\Pi}(I, \mathcal{A}(I)) \leq \alpha\, OPT_{\Pi}(I)$$

for each $I \in D_{\Pi}$. A similar definition holds for maximization problems, and for the case when α depends on the input size of I; that is, $\alpha : \mathbb{N} \to \mathbb{R}$.

1.2.1 Approximation Preserving Reductions

We conclude the section by introducing the definition of an *approximation preserving reduction* between pairs of *minimization* problems. This object shares similar characteristics to those of polynomial reductions between languages, however its definition is slightly more complicated.

Definition 1.2.5 (Approximation Preserving Reduction)**.** *Let Π_1 and Π_2 be two **NP**-minimization problems. Moreover, suppose that $\mathcal{A}_1$ and $\mathcal{A}_2$ are two algorithms, with the properties outlined below. We assume that $I_1 \in D_{\Pi_1}$:*

- *The algorithm $\mathcal{A}_1$ takes in the problem instance I_1, and returns an instance I_2 of Π_2, which we denote by $\mathcal{A}_1(I_1)$. Moreover, we have that,*

$$OPT_{\Pi_2}(I_2) \leq OPT_{\Pi_1}(I_1).$$

  In other words, the optimum value of I_2 is less than or equal to the optimum value of I_1.

- *If we take any feasible solution $s_2 \in S_{\Pi_2}(I_2)$, then given the inputs I_1 and s_2, the algorithm $\mathcal{A}_2$ returns a feasible solution $s_1 \in S_{\Pi_1}(I_1)$, which we denote by $\mathcal{A}_2(I_1, s_2)$. Moreover, the objective value of s_1 is less than the objective value of s_2. Rather,*

$$opt_{\Pi_1}(I_1, s_1) \leq opt_{\Pi_2}(I_2, s_2).$$

- *The algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ operate in polynomial time.*

If the algorithms satisfy these properties, then we refer to the pair $(\mathcal{A}_1, \mathcal{A}_2)$ as an approximation preserving reduction from Π_1 into Π_2.

If $(\mathcal{A}_1, \mathcal{A}_2)$ is an approximation preserving reduction between minimization problems Π_1 and Π_2, then it is clear from the above definition that

$$OPT_{\Pi_2}(\mathcal{A}_1(I_1)) \leq OPT_{\Pi_1}(I_1),$$

for each $I_1 \in D_{\Pi_1}$. It turns out that the second property of $(\mathcal{A}_1, \mathcal{A}_2)$ in the above definition allows us to conclude the reverse inequality as well. Rather,

$$OPT_{\Pi_1}(I_1) \leq OPT_{\Pi_2}(\mathcal{A}_1(I_1)).$$

Thus,

$$OPT_{\Pi_1}(I_1) = OPT_{\Pi_2}(\mathcal{A}_1(I_1)).$$

for all $I_1 \in D_{\Pi_1}$. We conclude the section by summarizing a number of useful properties of these types of reductions:

Proposition 1.2.6. *Let Π_1 and Π_2 be two **NP**-minimization problems, for which an approximation preserving reduction exists from Π_1 into Π_2. We observe the following claims hold:*

- *If Π_1 is **NP**-hard, then so is Π_2.*
- *Let $\alpha \geq 1$. If Π_2 has an algorithm with an approximation guarantee of α, then so will Π_1.*

While the results developed involving approximation presevering reductions are for pairs of minimization problems, similar definitions and results hold for pairs of maximization problems.

1.3 Linear Programming Techniques and Applications

To conclude the chapter, we introduce a special class of optimization problems, known as integer programs. We then examine how we can model many **NP**-optimization problems using this framework. Moreover, we introduce a relaxation of integer programs, know as linear programs. We consider some basic properties of these programs, and explore some techniques that will help us develop approximation algorithms for **NP**-hard optimization problems.

Let us suppose that $n, m \geq 1$ are integers, and we are given an n-vector $\boldsymbol{c} = (c_1, \ldots, c_n) \in \mathbb{Q}^n$, an m-vector $\boldsymbol{b} = (b_1, \ldots, b_m) \in \mathbb{Q}^m$, together with an $m \times n$ matrix $A = (a_{i,j}) \in \mathbb{Q}^{m \times n}$. We can use these parameters to define an *integer program* (IP), specified as either a minimization or maximization problem. We first consider the case under the former specification:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{n} c_j \, x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{i,j} \, x_j \geq b_j \qquad \forall i = 1, \ldots, m, \\ & x_j \in \mathbb{Z}_{\geq 0} \quad \forall j = 1, \ldots, n. \end{aligned} \tag{1.3.1}$$

Similarly, we may construct an integer program, whose objective is maximization:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} c_j \, x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{i,j} \, x_j \leq b_j \qquad \forall i = 1, \ldots, m, \\ & x_j \in \mathbb{Z}_{\geq 0} \quad \forall j = 1, \ldots, n. \end{aligned} \tag{1.3.2}$$

In either context, we refer to the vector $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{Z}^n_{\geq 0}$ as the *variable* of the integer program. Moreover, any assignment to $\boldsymbol{x}$ which satisfies the above *constraints* (inequalities) is referred to as a *feasible solution*. If $\boldsymbol{x}$ is feasible and *minimizes (maximizes)* the value of the linear *objective function* $\sum_{j=1}^{n} c_j \, x_j$, then it is referred to as an *optimum solution*.

It will sometimes be convenient to specify additional constraints on our integer programs. For example, we may wish to place additional bounds on the variable $\boldsymbol{x}$, such as restricting its coordinates to the set $\{0, 1\}$, or allowing its coordinates to take negative values. We may also wish to add constraints which impose exact equalities opposed to inequalities. Rather, constraints of the form, $\sum_{j=1}^{n} a_{i,j} = b_i$, where $1 \leq i \leq m$. It terms out that the framework above is sufficiently general to encode all these extra restrictions. As a result, integer programs which mimic the forms of IP (1.3.1) and IP (1.3.2) are often said to be in *canonical form*. We mostly concern ourselves with *canonical integer programs* throughout the section.

Let us now once again consider the vertex cover problem (Problem 1.2.1). Our goal is to build an integer program that encodes the restrictions of the problem. In order to do this, let us suppose we are given a graph $G = (V, E)$, together with

a weight function $w : E \to \mathbb{Q}_{\geq 0}$. We first label order the vertices and edges of G, $v_1, \ldots, v_n$ and $e_1, \ldots, e_m$ respectively, where $n := |V|$ and $m := |E|$.

For each $j = 1, \ldots, n$, we introduce the variable x_j associated to the vertex v_j, whose values are restricted to the set $\{0, 1\}$. Clearly, any assignment to the variable $\boldsymbol{x} := (x_1, \ldots x_n)$ corresponds to a subset of vertices $U \subseteq V$, where $v_j \in U$, if and only if $x_j = 1$, for each $j = 1, \ldots, n$.

In order to ensure that the feasible solutions of the program are in bijection with the valid vertex covers of G, we introduce m inequalities involving the variable $\boldsymbol{x}$. Rather, for each edge $i = 1, \ldots, m$, we impose the restriction

$$x_{j_1} + x_{j_2} \geq 1,$$

where $e_i = \{v_{j_1}, v_{j_2}\}$, and $1 \leq j_1 < j_2 \leq n$. Let us once again denote the vertex set associated to the assignment of $\boldsymbol{x}$ as U. Clearly, U will include at least one vertex from every edge $e \in E$, precisely when $\boldsymbol{x}$ is feasible. This observation is sufficient to establish the desired bijection.

Let us now define the m-vector $\boldsymbol{b} := (1, \ldots, 1)$, and the $m \times n$ matrix $A = (a_{i,j})$, where

$$a_{i,j} = 1, \text{ if and only if vertex } v_j \text{ is in edge } e_i.$$

Finally, we define the n-vector $\boldsymbol{c}$, where $\boldsymbol{c} := (w(v_1), \ldots, w(v_n))$, so that the objective function $\sum_{j=1}^{n} c_j \, x_j$ properly encodes the value of the vertex set corresponding to $\boldsymbol{x}$. Using the parameters A, $\boldsymbol{b}$ and $\boldsymbol{c}$, consider the following integer program:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{n} c_j \, x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{i,j} \, x_j \geq b_j \qquad \forall i = 1, \ldots, m, \\ & x_j \in \{0, 1\} \quad \forall j = 1, \ldots, n. \end{aligned} \tag{1.3.3}$$

The program is clearly not in canonical form, however this can be rectified by encoding each upper bound on the variables as an additional inequality in the matrix A. We leave this for the reader to verify.

While we ordered the vertices and edges of G to state our integer program in the notation of IP (1.3.1), we typically ignore this formality. Instead, we state our integer program in the notation of the graph G. That is, for each $v \in V$, we introduce the variable x_v, whose values are restricted to the set $\{0, 1\}$. In this notation, we observe the following formulation of the vertex cover problem on G:

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} w(v) \, x_v \\ \text{subject to} \quad & x_u + x_v \geq 1 \qquad \forall e = \{u, v\} \in E \\ & x_v \in \{0, 1\} \quad \forall v \in V. \end{aligned} \tag{1.3.4}$$

Now that we've witnessed the expressability of integer programs, it is natural to wonder whether there exists a polynomial time algorithm for solving them. Namely, we wish to find an algorithm, which given an *arbitrary* canonical integer program in the form of IP (1.3.1) or IP (1.3.2), returns optimum solution $\boldsymbol{x}^*$ in time polynomial in the size of A,$\boldsymbol{b}$, and $\boldsymbol{c}$.

While algorithms exist that execute efficiently for *specific* classes of integer programs, no such algorithm is known to exist for *all* integer programs. In light of the **NP**-hardness of the vertex cover problem, together with IP (1.3.4), it is unlikely that such an algorithm exists.

Theorem 1.3.1. *The problem of finding an optimum solution to an arbitrary integer program does not have a polynomial time solution, provided* $\boldsymbol{P \neq NP}$.

While there is likely no way of solving general integer programs in polynomial time, there exists a class of optimization problems which *are* efficiently solvable, and closely resemble integer programs. We refer to these optimization problems of this kind as *linear programs*. Broadly speaking, they differ only from integer programs in that they allow their solutions to take on *real* numbers.

As in the case of integer programs, let us suppose that $n, m \geq 1$ are integers, and that we are given an n-vector $\boldsymbol{c} = (c_1, \ldots, c_n) \in \mathbb{Q}^n$, as well as an m-vector $\boldsymbol{b} = (b_1, \ldots, b_m) \in \mathbb{Q}^m$. Moreover, assume that A is an $m \times n$ matrix $A = (a_{i,j}) \in \mathbb{Q}^{m \times n}$. We can use these parameters to formulate a *linear program* (LP), whose goal is minimization or maximization. In this case of minimization, we have the following program:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{n} c_j \, x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{i,j} \, x_j \geq b_j \quad \forall i = 1, \ldots, m, \\ & x_j \geq 0 \quad \forall j = 1, \ldots, n. \end{aligned} \tag{1.3.5}$$

Similarly, we may construct a linear program, whose objective is maximization:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} c_j \, x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{i,j} \, x_j \leq b_j \quad \forall i = 1, \ldots, m, \\ & x_j \geq 0 \quad \forall j = 1, \ldots, n. \end{aligned} \tag{1.3.6}$$

The terminology of linear programs mirrors that of integer programs. In paritcular, we refer to the vector $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{Q}^n$ as a *variable*, and assignments of $\boldsymbol{x}$ as *feasible solutions*, provided they satisfy the above *constraints* (inequalities). Moreover, an assignment of $\boldsymbol{x}$ is referred to as *optimum*, provided it *minimizes (maximizes)* the value of the linear *objective function* $\sum_{j=1}^{n} c_j \, x_j$, while remaining feasible. Finally, we refer to linear programs that conform to the structure of LP (1.3.5) and LP (1.3.6) as being *canonical*. The structure of canonical linear programs is sufficiently general to account for a number of natural modifications, as in the case of integer programs (see the discussion after IP (1.3.2)).

There exists a number of polynomial time algorithms that imply the following theorem:

Theorem 1.3.2. *There exists an algorithm Ψ and a polynomial p, which given any canonical linear program parameterized by $A, \boldsymbol{b}$, and $\boldsymbol{c}$, returns an optimum solution $\boldsymbol{x}^*$ in at most $p(|A| + |\boldsymbol{b}| + |\boldsymbol{c}|)$ steps. Moreover, the optimum solution $\boldsymbol{x}^*$ is guaranteed to take on rational values in all its coordinates; that is, $\boldsymbol{x}^* \in \mathbb{Q}^n$.*

.

The fact that the solution $\boldsymbol{x}^*$ takes on rational numbers is convenient in that we can store its coordinates exactly. We do not have to worry about approximating these values with rational numbers when using $\boldsymbol{x}^*$ in computations.

Let us now consider the minimization linear program formed by restricting our feasible solutions to rational numbers:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{n} c_j \, x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{i,j} \, x_j \geq b_j \qquad \forall i = 1, \ldots, m, \\ & x_j \in \mathbb{Q}_{\geq 0} \quad \forall j = 1, \ldots, n. \end{aligned} \tag{1.3.7}$$

Clearly, the feasible solutions of this linear program remain feasible solutions to LP (1.3.5). Moreover, as a result of Theorem 1.3.2, we also have the following corollary:

Corollary 1.3.3. *If $\boldsymbol{x}^*$ is an optimum solution to LP* (1.3.5)*, then there is an optimum solution $\boldsymbol{x}^{**}$ to LP* (1.3.7)*, for which,*

$$\sum_{j=1}^{n} c_j \, x_j^* = \sum_{j=1}^{n} c_j \, x_j^{**}.$$

That is, $\boldsymbol{x}^*$ and $\boldsymbol{x}^{**}$ have equal value.

As a result of this claim, we refer to LP (1.3.5) and LP (1.3.7) as being *equivalent* as linear programs. We may work with either formulation, depending on which is more convenient. Throughout the thesis, we work with both variations, however the form of LP (1.3.5) is most useful from a notational perspective,

wheras our algorithms employ the rational solutions of LP (1.3.7) in practise. In any case, we consider LP (1.3.7) to also be in canonical form. Of course, we can develop similar definitions and claims for linear programs stated as maximization problems.

Let us now reconsider the vertex cover problem in the structure of a linear program. Given IP (1.3.4), we form a linear program, known as the *linear relaxation* of IP (1.3.4), by allowing its variables to take on real numbers. This leaves us with the linear program below, where $G = (V, E)$ is a graph with weight function $w : V \to \mathbb{Q}_{\geq 0}$:

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} w(v)\, x_v \\ \text{subject to} \quad & x_u + x_v \geq 1 \qquad \forall e = \{u, v\} \in E \\ & x_v \in [0, 1] \quad \forall v \in V. \end{aligned} \tag{1.3.8}$$

Observe that we can forgo the upper bounds on these variables, without decreasing the value of an optimum solution. This is easily seen, as the function $w : V \to \mathbb{Q}_{\geq 0}$ is guaranteed to be nonnegative, and so its optimum solutions have their coordinates as *small* as possible. We therefore have the following *canonical* linear program, after restricting our attention to rational solutions:

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} w(v)\, x_v \\ \text{subject to} \quad & x_u + x_v \geq 1 \qquad \forall e = \{u, v\} \in E \\ & x_v \in \mathbb{Q}_{\geq 0} \quad \forall v \in V. \end{aligned} \tag{1.3.9}$$

We may think of the assignment $\boldsymbol{x} = (x_v)_{v \in V}$ as specifying a *fractional vertex cover* for G, where each value x_v indicates the *fractional* extent to which v is included in the vertex cover. In light of Theorem 1.3.2, there is an optimum solution $\boldsymbol{x}^*$ to LP (1.3.9), which we refer to as an *optimum fractional vertex cover*. The value of this optimum solution, namely $\sum_{j=1}^{n} c_j \, x_j^*$, is denoted by $OPT_f(I)$, where $I := (G, w)$ is the problem instance of the vertex cover problem. We now employ a linear programming technique known as *rounding* to attain an approximation algorithm for the *general* vertex cover problem.

Broadly speaking, the rounding algorithm begins with a problem instance (G, w) of the vertex cover problem. We then encode the problem as IP (1.3.4), using G and w as parameters. From here, relax this integer problem to form LP (1.3.9). We use Theorem 1.3.2 to invoke an algorithm which returns an optimum (fractional) solution $\boldsymbol{x}^*$ to LP (1.3.9), in time polynomial in $|G|$ and $|w|$. For each vertex $v \in V$, we examine the coordinate x_v^*, and round it up to one, provided $x_v^* \geq 1/2$. Otherwise, we round the coordinate x_v^* down to zero. The resulting assignment $\boldsymbol{x}^*$ is no longer fractional, but is instead integral, and thus a solution of IP (1.3.4). In particular, we may associate to it a vertex cover U of G, which is returned as the output of the algorithm. Let us now summarize this procedure:

Algorithm 1.3.1 Vertex Cover Algorithm

Let $G = (V, E)$ be an undirected graph.
Let $w : V \to \mathbb{Q}_{\geq 0}$ be a weight function.
1: Initialize $U \leftarrow \emptyset$.
2: Initialize vector $\boldsymbol{x}^* = (x_v^*)_{v \in V}$.
3: Solve LP (1.3.9), with (G, w) as inputs.
4: Store the result in $\boldsymbol{x}^*$.
5: **for** each vertex $v \in V$ **do**
6: **if** $x_v^* \geq \frac{1}{2}$ **then**
7: Add v to U.
8: Return U

We first observe that the algorithm is guaranteed to execute in polynomial time, as we can solve LP (1.3.9) in polynomial time. Moreover, the algorithm is correct; that is, U is always a vertex cover of G. To see this, suppose that e is an edge of G, where $e = \{u, v\}$. As the solution $\boldsymbol{x}^*$ is feasible, we know that

$$x_u^* + x_v^* \geq 1.$$

In particular, at least of one of x_u^* and x_v^* has value greater or equal to $1/2$. As a result, U will contain at least one of u and v, and thus will cover e. This implies that U is a valid vertex cover, as e was arbitrary.

Theorem 1.3.4. *Algorithm 1.3.1 achieves an approximation guarantee of* 2.

Proof. We assume that $G = (V, E)$ is an arbitrary graph with a weight function $w : V \to \mathbb{Q}_{\geq 0}$, each of which is passed to the algorithm. Let us suppose that U is the vertex cover of G returned by the algorithm. If $OPT(G, w)$ denotes the value of an optimum vertex cover of G with respect to w, then our goal is to show that,

$$w(U) \leq 2\, OPT(G, w).$$

In other words, we must bound the weight of U by twice the weight of an optimum vertex cover for G.

Towards this goal, let us suppose that $\boldsymbol{x}^*$ is the optimum solution to LP (1.3.9), as used in the algorithm. As $\boldsymbol{x}^*$ is an optimum fractional vertex cover, we denote its value by $OPT_f(G, w)$. Observe that since all solutions to IP (1.3.4) remain solutions to LP (1.3.9), we may conclude that $OPT_f(G, w) \leq OPT(G, w)$. Thus, if we can show that $w(U) \leq 2\, OPT_f(G, w)$, the proof will be complete.

Let us denote the integral solution of IP (1.3.4) corresponding to U by $\boldsymbol{x}^{**}$. That is, $v \in U$, if and only if $x_v^{**} = 1$, for each $v \in V$. It is clear that $\boldsymbol{x}^{**}$ is the *rounded* version of $\boldsymbol{x}^*$. That is, for each $v \in V$, $x_v^{**} = 1$, if and only if $x_v^* \geq 1/2$. In light of these observations, we have that

$$\begin{aligned} w(U) &= \sum_{v \in V} w(v)\, x_v^{**} \\ &\leq \sum_{v \in V} 2\, w(v)\, x_v^* \\ &= 2\, OPT_f(G, w). \end{aligned}$$

Thus, $w(U) \leq 2\, OPT_f(G, w)$, so the claim holds.

□

1.3.1 Integrality Gaps and Rounding Algorithms

In the preceding argument, the main proof technique involved comparing the value of U with the value of $OPT_f(G, w)$. This is typical of all linear programming based algorithms, which employ rounding techniques to approximate optimum vertex covers. It turns out that it is possible to understand how well these

algorithms can perform, by considering a value known as the *integrality gap* of IP (1.3.4). If we denote the vertex cover problem by Π, then we define this to be

$$\sup_{(G',w')\in D_\Pi} \frac{OPT(G',w')}{OPT_f(G',w')},$$

where $OPT(G',w')$ is the value of an optimum *integral* vertex cover of G', and $OPT_f(G',w')$ is the value of an optimum *fractional* vertex cover of G'. We may denote this value by Λ_Π for convenience. Clearly, $\Lambda_\Pi \geq 1$, as all solutions to IP (1.3.9) are solutions to LP (1.3.9). We claim that the analysis done in the proof of Theorem 1.3.4 implies that $\Lambda_\Pi \leq 2$.

Let us suppose that G and w are passed to Algorithm 1.3.1. As we saw, a vertex cover U of G is returned, for which

$$w(U) \leq 2\, OPT_f(G,w). \tag{1.3.10}$$

By definition, we know that $OPT(G,w) \leq w(U)$. Thus, we have that,

$$\frac{OPT(G,w)}{OPT_f(G,w)} \leq 2,$$

for each graph G and vertex weight w passed to Algorithm 1.3.1. We typically refer to the ratio on the left side of this equation as the *integrality gap of instance* (G,w). In any case, we may conclude that $\Lambda_\Pi \leq 2$, after taking a supremum over the instances of Π with respect to Equation 1.3.10. The desired claim therefore holds. Moreover, it is clear that *any* approximation algorithm whose analysis employs a comparison in the form of Equation 1.3.10 cannot improve upon the approximation guarantee of Λ_Π.

In the later chapters, we often use linear programming rounding techniques to approximate solutions to optimization problems. It is therefore useful to consider a general definition of the integrality gap of an optimization problem Π.

Let us suppose that Π is classified as a minimization problem; similar definitions hold for maximization problems. We may assume that there is an *IP-encoding* of Π, which is defined as a pair of functions (ϕ, ψ) satisfying the properties listed below:

The function ϕ is typically referred to as an *IP-encoding map* for Π. It has the property that it injectively maps instances of Π to minimization integer programs. Typically, these minimization programs are assumed to be canonical form. We refer to the function ψ as a *IP-conversion map for* Π. Its inputs include an instance I of Π, together with a feasible solution s of $S_\Pi(I)$. This function satisfies a number of properties:

- For each $I \in \Pi$, and $s \in S_\Pi(I)$, $\psi(I, s)$ is a feasible solution of $\phi(I)$, whose *value* is equal to the value of s. In light of this property, we refer to ψ as *value-preserving*.
- For each $I \in \Pi$, $\psi(I, \cdot)$ forms a bijection from $S_\Pi(I)$ into the set of all feasible solutions of $\phi(I)$.

When the context is clear, we typically drop the prefix "IP", and refer to (ϕ, ψ) as an *encoding*, where ϕ is an *encoding map* and ψ is a *conversion map*.

For each instance I of Π, observe that the existence of (ϕ, ψ) implies that the value of an optimum solution of I is equal to the value of an optimum solution of $\phi(I)$. In each case, we denote this value by $OPT(I)$. Let us now suppose

that $\phi(I)_f$ is the relaxation of $\phi(I)$ as a canonical linear program. We may denote the value of an optimum solution of $\phi(I)_f$ by $OPT_f(I)$. It is clear that $OPT_f(I) \leq OPT(I)$. We refer to the ratio,

$$\frac{OPT(I)}{OPT_f(I)},$$

as the *integrality gap of instance I with respect to the encoding* (ϕ, ψ). Moreover, we refer to the supremum,

$$\sup_{I \in D_\Pi} \frac{OPT(I)}{OPT_f(I)},$$

as the *integrality gap of Π with respect to the encoding* (ϕ, ψ). We shall see later in the text that there often exists multiple ways of encoding a single optimization problem, thus leading to multiple integrality gaps for the problem. Of course, if there is no ambiguity, then we may refer to this supremum as the *integrality gap of Π*, or in a slight abuse of notation, as the *integrality gap of $\phi(I)$*.

From an algorithmic perspective, the most useful IP encodings are the ones whose functions can be evaluated in polynomial time. We saw this in the vertex cover problem, as it was the main tool used in the approximation algorithm we built. It turns out that similar claims are true of any optimization problem Π that uses the *rounding algorithm* below as a template. Once again, we assume that (ϕ, ψ) is an encoding of the problem Π, and that both these functions are computable:

Algorithm 1.3.2 General Rounding Algorithm

Let I be an instance of Π.
1: Store the IP $\phi(I)$, after computing it from I.
2: Store the LP $\phi(I)_f$, after relaxing it from $\phi(I)$.
3: Store an optimum solution $\boldsymbol{x}$ of LP $\phi(I)_f$, by passing $\phi(I)_f$ to an LP-solving algorithm (See Theorem 1.3.2).
4: Store a solution $\boldsymbol{x}^*$ of IP $\phi(I)$, by rounding the coordinates of $\boldsymbol{x}$ to integer values.
5: Compute the solution $\psi^{-1}(I, \boldsymbol{x}^*)$.
6: Set $s \leftarrow \psi^{-1}(I, \boldsymbol{x}^*)$.
7: Return s.

It is clear that the algorithm explicitly stores both the programs $\phi(I)$ and $\phi(I)_f$. It then solves $\phi(I)_f$, yielding an optimum solution $\boldsymbol{x}^*$, whose values are then rounded to an integral solution $\boldsymbol{x}$ of $\phi(I)$. This rounding process is of course dependent on the specific problem at hand. In any case, Algorithm 1.3.2 will certainly have a runtime *at least* polynomial in the size of $\phi(I)$. In particular, if $\phi(I) \geq \Omega(2^{|I|})$ for all $I \in D_{\Pi}$, then the runtime of this algorithm will also be exponential in the size of its inputs.

Conversely, if the encoding of I to $\phi(I)$ can be done in polynomial time, together with the conversion of $\boldsymbol{x}^*$ into s, then the algorithm itself will run in time polynomial in the size of I. The only caveat being that the rounding of $\boldsymbol{x}$ to $\boldsymbol{x}^*$ must also be doable in time polynomial in the size of $\boldsymbol{x}$. This is typically a reasonable assumption to make when considering rounding algorithms.

As consequence of the above remarks, it is clear that the runtime of Algorithm 1.3.2 is intimately connected to the efficiency for which ϕ and ψ can be computed. In the case of the map ψ, this is not an issue, as all the problems we consider throughout the text have efficient conversion algorithms. On the other hand, a

number of the problems in the later chapters have encoding maps which cannot be computed efficiently. It is therefore desirable to consider an algorithmic template which is similar to Algorithm 1.3.2, yet whose runtime is less reliant on the complexity of the encodings used.

In order to achieve this goal, we must first enforce some additional conditions on the IP-encoding (ϕ, ψ) of Π. Let us suppose that p and q are polynomials, and that $I \in D_\Pi$. We enforce the following conditions:

- The feasible solutions of IP $\phi(I)$ have at most $p(|I|)$ many coordinates.
- Each rational number of $\phi(I)$ can be represented in at most $q(|I|)$ many characters (bits).

If the IP-encoding (ϕ, ψ) satisfies these properties, then we say that it *compactly represents* Π. Notice that in this case, each inequality of $\phi(I)$ can be represented using at most $p(|I|)\, q(|I|)$ many characters. That being said, the number of *inequalities* of $\phi(I)$ may still be exponential in the size of I.

If we assume that our encoding (ϕ, ψ) compactly represents Π, then we may consider a class of algorithms which help us decide whether solutions of $\phi(I)_f$ are in fact feasible. Rather, assume that $I \in D_\Pi$, and that $\boldsymbol{x}$ is a *potentially* feasible solution to LP $\phi(I)_f$. We refer to an algorithm $\mathcal{A}$ as a *polynomial time separation oracle*, provided the properties below hold:

- Given I and $\boldsymbol{x}$ as inputs to $\mathcal{A}$, the algorithm operates in time polynomial in the size of I and $\boldsymbol{x}$.
- If $\boldsymbol{x}$ is a feasible solution to the LP $\phi(I)_f$, then the algorithm classifies $\boldsymbol{x}$ as a "yes" instance.

- If $\boldsymbol{x}$ is not a feasible solution to the LP $\phi(I)_f$, then $\boldsymbol{x}$ is classified as a "no" instance, and a *violated* inequality of $\phi(I)$ is returned by $\mathcal{A}$.

The existence of polynomial time separation oracles help us solve linear programs in polynomial time. This is truly an amazing result, as these linear programs may still have exponentially many inequalities.

Theorem 1.3.5. *Let Π be an optimization problem, with an IP-encoding (ϕ, ψ) that compactly represents it. If there exists a polynomial separation oracle $\mathcal{A}$ for Π and (ϕ, ψ), then there exists an algorithm Ψ with the following properties:*

- *Ψ takes in an instance I of Π as input, and is granted access to $\mathcal{A}$ as a subroutine.*
- *Ψ returns an optimal solution $\boldsymbol{x}$ of $\phi(I)_f$, in time polynomial in the size of I.*

In light of this theorem, we can modify our general rounding algorithm (Algorithm 1.3.2) to account for encodings that aren't computable in polynomial time, yet can still be compactly represented. That is, we assume that Π is an optimization problem with a compactly represented encoding (ϕ, ψ), together with a polynomial time separation oracle $\mathcal{A}$.

Algorithm 1.3.3 General Oracle Rounding Algorithm

Let I be an instance of Π.

1: Store an optimum solution $\boldsymbol{x}$ of LP $\phi(I)_f$, by passing I to a linear program solver with access to subroutine $\mathcal{A}$ (see Theorem 1.3.5).
2: Store a solution $\boldsymbol{x}^*$ of IP $\phi(I)$, by rounding the coordinates of $\boldsymbol{x}$ to integer values.
3: Compute the solution $\psi^{-1}(I, \boldsymbol{x}^*)$.
4: Set $s \leftarrow \psi^{-1}(I, \boldsymbol{x}^*)$.
5: Return s.

Observe that because of Theorem 1.3.5, the solution $\boldsymbol{x}$ is guaranteed to be polynomial in the size of I. If we also assume that the conversion map ψ is computable in polynomial time, then it is clear that the entire algorithm runs in polynomial time. We shall see in the later chapters that the existence of this algorithm is essential in the design of linear programming based rounding algorithms. With that being said, encoding which can be computed in polynomial time are still the most desirable, as Algorithm 1.3.2 is more efficient than Algorithm 1.3.3 in practise.

1.3.2 The Dual of a Linear Program

We conclude this chapter with an overview of the most important results from the duality theory of linear programming. In particular, we define the dual of a linear program, and review the Strong Duality Theorem. These results will be used in the later chapters to help us design approximation algorithms.

Let us suppose that $m, n \geq 1$ are integers, and that we are given a *minimization* linear program, parameterized by an $m \times n$ matrix A, together with an m-vector $\boldsymbol{b}$, and an n-vector $\boldsymbol{c}$. The matrix A and the m-vector $\boldsymbol{b}$ are used to write the m inequalities of the linear program, whereas the n-vector $\boldsymbol{c}$ is used to represent the program's objective funcction.

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{n} c_j \, x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{i,j} \, x_j \geq b_j \quad \forall i = 1, \ldots, m, \\ & x_j \geq 0 \quad \forall j = 1, \ldots, n. \end{aligned} \tag{1.3.11}$$

We may once again think of the n-vector $\boldsymbol{x} := (x_1, \ldots, x_n)$ as the variable of the linear program. Let us refer to LP (1.3.11) as the *primal*.

Suppose we now compute the *transpose* of the matrix A, namely A^T. This is an $n \times m$ matrix, formed by constructed the rows and columns of A. If we swap the roles of $\boldsymbol{b}$ and $\boldsymbol{c}$ from that of the primal, then we may form a *maximization* linear program. It uses A^T and $\boldsymbol{c}$ to form its inequalties, and $\boldsymbol{b}$ to form its objective function.

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{m} b_i \, y_i \\ \text{subject to} \quad & \sum_{i=1}^{m} a_{i,j} \, y_i \leq c_j \quad \forall j = 1, \ldots, n, \\ & y_i \geq 0 \quad \forall i = 1, \ldots, m. \end{aligned} \tag{1.3.12}$$

In this context, the m-vector $\boldsymbol{y} := (y_1, \ldots, y_m)$ is referred to as the variable of the linear program. We refer to LP (1.3.12) as the *dual* of LP (1.3.11), or simply the *dual* when the context is clear.

It turns out that the values taken on by the objective functions of the primal and dual are closely related. We summarize this relation in the theorem below:

Theorem 1.3.6 (Weak Duality Theorem)**.** *If $\boldsymbol{x}$ is a feasible solution to the primal (LP* (1.3.11)*), and $\boldsymbol{y}$ is a feasible solution to the dual (LP* (1.3.12)*), then we have that,*

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

In particular, a maximum solution to the dual has value less than or equal to a minimum solution to the primal.

It turns out that this inequality is in fact tight. That is, *optimum primal solutions* and *optimum dual solutions* have equal value.

Theorem 1.3.7 (Strong Duality Theorem)**.** *If $\boldsymbol{x}^*$ is a optimum solution to the primal (LP* (1.3.11)*), and $\boldsymbol{y}^*$ is an optimum solution the dual (LP* (1.3.12)*), then we have that,*

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

As a corollary, we can discern when pairs of solutions are each optimum. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be feasible solutions to the primal and dual respectively. Consider the following sets of equations:

Primal complementary slackness conditions

For each $1 \leq j \leq n$, either $x_j = 0$, or $\sum_{i=1}^{m} a_{i,j} \, y_i = c_j$.

Dual complementary slackness conditions

For each $1 \leq i \leq m$, either $y_i = 0$, or $\sum_{j=1}^{m} a_{i,j} \, x_j = b_i$.

Corollary 1.3.8 (Complementary Slackness Conditions)**.** *The solutions $\boldsymbol{x}$ and $\boldsymbol{y}$ are both optimum, if and only if they satisfy the primal and dual complementary slackness conditions.*

If we fix $\alpha, \beta \geq 1$, then we can also form an approximate version of this corollary. Let us once again assume that $\boldsymbol{x}$ and $\boldsymbol{y}$ are feasible solutions to the primal and dual. We may modify the above conditions, such that we no longer require exactness:

Approximate primal complementary slackness conditions

For each $1 \leq j \leq n$, either $x_j = 0$, or $c_j/\alpha \leq \sum_{i=1}^{m} a_{i,j} \, y_i = \ c_j$.

Approximate dual complementary slackness conditions

For each $1 \leq i \leq m$, either $y_i = 0$, or $b_i \leq \sum_{j=1}^{m} a_{i,j}\, x_j = \beta \cdot b_i$.

We shall often make use of the following proposition when designing *primal-dual* approximation algorithms in the later chapters.

Proposition 1.3.9. *If* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *satisfy the above conditions, then we have that,*

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

CHAPTER 2
Introduction to Network Flow Problems

In this chapter, two network flow problems on graphs are introduced, namely the path network flow problem and the edge network flow problem. Later on, these problems are related to the single source-sink pair minimum cut problem. This includes a number of applications to graph theory, where both directed and undirected networks are considered.

In the first section, the edge and path flow problems are related to each other. In particular, the section develops efficient algorithms for converting solutions from one problem to the other. While the relations between these problems have been studied before in the book "Approximation Algorithms" by Vazirani [Vv11], the specific algorithms presented are developed and analyzed independently by myself.

The next section introduces the Ford-Fullkerson algorithm, whose analysis is adapted from the book "Algorithm Design" by Kleinberg and Tardos [KT14]. This naturally leads to a specific implementation of this algorithm, known as the Edmonds-Karp algorithm, which was established in a journal article by Edmonds and Karp in 1972 [EK72]. While this algorithm is of course not original, I wrote its analysis independently of any sources.

The final section of the chapter includes a number of important applications of this algorithm, most notably the max-flow min-cut theorem. These applications

are cited from the book "Algorithm Design" [KT14], though I developed the terminology and analysis independently.

2.1 Path and Edge Network Flow

Our goal in this section is to introduce two network flow problems, and to relate them to each other algorithmically. We mainly work with *directed graphs* when studying these problems, so we review some of the terminology regarding these objects.

Suppose we are given a *directed graph* $G = (V, E)$, where V denotes the *vertices* of G, and E denotes the *directed edges* of G. A *directed edge* e between vertices $u, v \in V$, is an ordered pair, $e = (u, v)$. The position of the vertices in this pair encodes the direction of the edge. In particular, we say that u *points* to v; referring to u as the *head* and v as the *tail* of e. If the context is clear, we refer to directed graphs as graphs, and directed edges as edges.

If $u \in V$ is a fixed vertex of G, then we define the *in-neighbourhood of* u, denoted $N^-(u)$, as the vertices of G which point to u. Similarly, we define the *out-neighbourhood of* u, denoted $N^+(u)$, where $N^+(u) := \{v \in V : (u, v) \in E\}$. If $S \subseteq V$, then we denote $\delta^-(S) := \{e = (u, v) : v \in S \text{ and } e \in E\}$ and $\delta^+(S) := \{e = (u, v) : u \in S \text{ and } e \in E\}$.

Let us also suppose that we are also given a *capacity function*, $c : E \to \mathbb{Q}_{\geq 0}$. For each edge $e \in E$, we refer to the value $c(e)$ as the *capacity* of e.

Finally, suppose we are given a pair of vertices (s, t) of G. We refer to s as the *source* node, and t as the *sink* node.

The purpose of the network flow problem is to route an abstract commodity between s and t. In order to do so, let us suppose Γ is the set of all simple directed paths from s to t. Moreover, for each $\gamma \in \Gamma$, let f_γ be the amount of flow routed

through γ. We wish to assign a value to each variable f_γ, such that the amount of flow passing through any given edge does not exceed its capacity. Formally, we wish that

$$\sum_{\gamma \in \Gamma : e \in \gamma} f_\gamma \leq c(e),$$

for each $e \in E$.

If an assignment $\boldsymbol{f} = (f_\gamma)_{\gamma \in \Gamma}$ satisfies all these constraints, then we refer to it as a *path flow* through G. We wish to find a path flow which maximizes the amount of flow leaving the source. We formally outline this problem below:

Problem 2.1.1 (Path Network Flow)**.** *We are given a directed graph $G = (V, E)$ with capacity function $c : E \to \mathbb{Q}_{\geq 0}$, and source-sink pair (s, t). Let us suppose Γ refers to the set of all directed paths from s to t. The objective is to find a path flow through G, denoted $\boldsymbol{f} = (f_\gamma)_{\gamma \in \Gamma}$, which maximizes the value of $\sum_{\gamma \in \Gamma} f_\gamma$. We refer to this quantity as the value of the path flow $\boldsymbol{f}$; denoted $val(\boldsymbol{f})$ for convenience. We remark that when storing $\boldsymbol{f}$, we adopt the convention that only paths with nonzero flow are kept.*

While the above network flow problem is easy to state, it does not lend itself well to algorithmic design. Rather, for a typical graph G, the number of paths between s and t may be superpolynomial in size. Thus, working directly with the variables $(f_\gamma)_{\gamma \in \Gamma}$ is difficult to do efficiently. As we wish to find a polynomial time algorithm to solve Problem 2.1.1, this limitation is undesirable.

In order to get around this issue, we introduce another network flow problem. For each edge $e = (u, v) \in E$, let ζ_e be the amount of flow which passes from u to v. As before, we require that ζ_e is bounded above by the capacity of the edge e.

In addition to these constraints, we also require that most of the vertices of G satisfy a *conservation condition*. Rather, we have that,

$$\sum_{u \in N^-(v)} \zeta_{(u,v)} = \sum_{w \in N^+(v)} \zeta_{(v,w)},$$

for all $v \in V$, where $v \neq s, t$. In other words, for each vertex v (excluding s and t), the amount of flow that enters v is equal to the amount of flow that leaves it.

Finally, we impose some special conditions on the source and sink. Rather, for each $v \in V$, if $(v, s) \in E$ then we have that $\zeta_{(v,s)} = 0$. Similarly, for each $w \in V$, if $(t, w) \in E$, then we have that $\zeta_{(t,w)} = 0$. In other words, we require that there is no flow entering the source, and no flow leaving the sink.

If an assignment $\boldsymbol{\zeta} = (\zeta_e)_{e \in E}$ satisfies these constraints, then we refer to it as an *edge flow* through G. As before, we wish to maximize the total flow that leaves the source. Let us now formally outline this problem:

Problem 2.1.2 (Edge Network Flow)**.** *We are given a directed graph $G = (V, E)$ with capacity function $c : E \to \mathbb{Q}_{\geq 0}$, and source-sink pair (s, t). Let us suppose Γ refers to the set of all simple paths from s to t. The objective is to find an edge flow through G, denoted $\boldsymbol{\zeta} = (\zeta_e)_{e \in E}$, which maximizes the value of $\sum_{u \in V : (s,u) \in E} \zeta_{(s,u)}$. We refer to this quantity as the value of the edge flow $\boldsymbol{f}$; denoted $val(\boldsymbol{f})$ for convenience.*

It turns out that Problems 2.1.2 and 2.1.1 are in a strong sense equivalent. Rather, any edge flow can be converted to a path flow of equal value, and vice versa. Before we describe this conversion, we first focus on some properties of edge flows in their own right.

Suppose that $G = (V, E)$ and $c : E \to \mathbb{Q}_{\geq 0}$ are as above, and $(\zeta_e)_{e \in E}$ is an edge flow through G. Given a vertex v of G, let us define $Z^+(v) := \sum_{u \in N^+(v)} \zeta_{(v,u)}$, and $Z^-(v) := \sum_{w \in N^+(v)} \zeta_{(w,v)}$. We refer to these values as the *outflow* and *inflow* of v, respectively. Given source sink pair (s, t), we observe that the constraints of ζ on s and t reduce to $Z^-(s) = 0$ and $Z^+(t) = 0$. Similarly, the conservation constraints reduce to $Z^-(v) = Z^+(v)$, for all $v \in V$, $v \neq s, t$.

We may extend the definitions of Z^+ and Z^- to subsets of vertices as well. Rather, if $S \subseteq V$, then we define $Z^+(S)$ to be the total edge flow leaving S, and $Z^-(S)$ to be the total edge flow entering S. Formally, we have that $Z^+(S) := \sum_{e \in \delta^+(S)} \zeta_e$ and $Z^-(S) := \sum_{e \in \delta^-(S)} \zeta_e$. We refer to these quantities as the *outflow* and *inflow* of S, respectively.

Recall that if $S \subseteq V$ has $s \in S$ and $t \in V \setminus S$, then we refer to $(S, V \setminus S)$ as an *s-t cut*. Observe the following proposition regarding inflows and outflows of (s, t) cuts:

Proposition 2.1.1. *Let $(\zeta_e)_{e \in E}$ be an edge flow through G. If $(S, V \setminus S)$ is an (s, t) cut, then we have that $val(\boldsymbol{\zeta}) = Z^+(S) - Z^-(S)$.*

Proof. We first observe that $val(\boldsymbol{\zeta}) = Z^+(s) - Z^-(s)$, as $\boldsymbol{\zeta}$ is assumed to not send any flow into s.

Moreover, as $\boldsymbol{\zeta}$ satisfies the conservation conditions, we know that in particular, $Z^+(v) - Z^-(v) = 0$ for all $v \in S$, $v \neq s$.

It follows that, $val(\boldsymbol{\zeta}) = \sum_{v \in S} Z^+(s) - Z^-(s)$.

Let us consider this sum on the right-hand side of the above equation. Observe that if $e = (u, v) \in E$, whose head and tail lie in S, then ζ_e will occur in terms $Z^+(u)$ and $Z^-(v)$ exclusively. Similarly, if only the head u of e is in S, then ζ_e will occur exclusively in term $Z^+(u)$. If only the tail v of e is in S, then ζ_e will only occur in term $Z^-(v)$. When neither end of e is in S, ζ_e will not occur in any term of the sum.

Using these above observations, we notice that,

$$\sum_{v \in S} Z^+(v) - Z^-(v) = Z^+(S) - Z^-(S).$$

It follows that,

$$val(\boldsymbol{\zeta}) = Z^+(S) - Z^-(S).$$

This completes the proof of the proposition. □

We now return to studying the conversion between edge and path flows. In particular we introduce two algorithms; one for each side of the conversion. Let us begin with the conversion of edge flows to path flows. From an efficiency perspective, this conversion can be done in polynomial time.

At a high level, the algorithm searches for simple paths which are *fully nonzero* (paths which have nonzero edge flow at each of their edges). When such

a path γ' is found, the minimum edge flow $\min_{e\in\gamma'} \zeta_e$ is assigned to $f_{\gamma'}$. This value is then subtracted from ζ_e for each each edge e of p'. The algorithm then continues the search for paths of nonzero edge flow, until no such path exists. At this point, the path flow $(f_\gamma)_{\gamma\in\Gamma}$ is returned.

Algorithm 2.1.1 Edge Flow to Path Flow Conversion Algorithm

Let (G, E) be a directed graph.
Let $c : E \to \mathbb{Q}_{\geq 0}$, and (s, t) a pair of distinct vertices.
Let $(\zeta_e)_{e\in E}$ be an edge flow.
1: Initialize an identically zero path flow $(f_\gamma)_{\gamma\in\Gamma}$.
2: **while** a fully nonzero simple path γ' from s to t exists in G **do**
3: Initialize $\varepsilon \leftarrow 0$.
4: Set $\varepsilon \leftarrow \min_{e\in\gamma'} \zeta_e$.
5: Set $f_{\gamma'} \leftarrow \varepsilon$.
6: **for** $e \in \gamma'$ **do**
7: Set $\zeta_e \leftarrow \zeta_e - \varepsilon$.
8: Return $(f_\gamma)_{\gamma\in\Gamma}$.

We first observe that the above algorithm runs in polynomial time. In order to show this, it is enough to bound the number of iterations of the "while loop". Consider the following proposition:

Proposition 2.1.2. *The number of iterations of the "while loop" in Algorithm 2.1.1 is at most $|E|$, the number of edges of G.*

Proof. Let us suppose that simple path γ' in computed in the "while loop". If ε is the minimum edge flow of γ', then at least one edge e of γ' has $\zeta_e = \varepsilon$. In particular, by the end of the "while loop", e will have been removed from G. Since the "while loop" terminates when no fully nonzero path from s to t exists in G,

at most $|E|$ edges will need to be removed from G to cause this termination. The result thus holds.

□

We now make a claim regarding the correctness of the above algorithm.

Proposition 2.1.3. *Let $\boldsymbol{\zeta} = (\zeta_e)_{e\in E}$ be an edge flow through G. The assignment $\boldsymbol{f} = (f_\gamma)_{\gamma\in\Gamma}$ returned by the algorithm, is a path flow for which $val(\boldsymbol{f}) = val(\boldsymbol{\zeta})$. Moreover, this computation is done in time $O(|E|^2 + |E||V|)$.*

Proof. Let us suppose $l \geq 0$ is the number of iterations of the "while loop" in Algorithm 2.1.1. By the above proposition, we know that $l \leq |E|$. Moreover, each iteration involves computing a depth first search started at s, so each iteration takes time $O(|E| + |V|)$. It follows that the algorithm takes time $O(|E|^2 + |E||V|)$. We now analyze the current edge flow and path flow at each of these iterations.

For clarity, let us set $\boldsymbol{\zeta^*}$ to denote the initial edge flow $\boldsymbol{\zeta}$. We then set $\boldsymbol{f}^0 = \mathbf{0}$ and $\boldsymbol{\zeta}^0 = \boldsymbol{\zeta}^*$. Similarly, for $i = 1, \dots l$, we set $\boldsymbol{f}^i$ to be the path flow and $\boldsymbol{\zeta}^i$ to be the edge flow, *after* i iterations of the "while loop".

Our first goal will be to show that the final path flow, $\boldsymbol{f}^l$, is valid. Towards, this goal, let $e_0 \in E$. We claim that,

$$\sum_{\gamma\in\Gamma} f^i_\gamma \leq \zeta^*_{e_0} - \zeta^i_{e_0}$$

for each $0 \leq i \leq l$.

In order to prove this claim, we use induction on i. In particular, if $i = 0$, then the statement is trivial, as $\boldsymbol{f}^0$ is identically zero.

Let us assume that it is true for $i \geq 0$. We now show that it holds for $i+1$.

Consider iteration $i+1$ of the "while loop". As we are concerned with the edge e_0, we consider two cases:

Let us first assume that the path γ' chosen by the algorithm does not include edge e_0. In this scenario, for each path $\gamma \in \Gamma$ with $e_0 \in \gamma$, its path flow will not change value. In other words, $f_\gamma^{i+1} = f_\gamma^i$.

Similarly, the edge flow of e_0 will also remain the same. We therefore have that $\zeta_{e_0}^{i+1} = \zeta_{e_0}^i$. It follows that the inequality will continue to hold trivially.

We now assume that the path γ' does include edge e_0. As in the algorithm, let us denote ε to be the minimal edge flow through γ'. Rather, $\varepsilon := \min_{e \in \gamma'} \zeta_e^i$.

Observe that after this iteration of the "while loop", the edge flow through e_0 will have been decreased by ε. Formally, we have that $\zeta_e^{i+1} = \zeta_e^i - \varepsilon$.

On the other hand, the path flow through γ' will have been increased by ε. It follows that $f_{\gamma'}^{i+1} = f_{\gamma'}^i + \varepsilon$.

Thus, by the induction hypothesis,

$$\begin{aligned}
\sum_{\gamma \in \Gamma : e_0 \in \gamma} f_\gamma^{i+1} &= \sum_{\gamma \in \Gamma : e_0 \in \gamma} f_\gamma^i + \varepsilon \\
&\leq \zeta_{e_0}^* - \zeta_{e_0}^i + \varepsilon \\
&= \zeta_{e_0}^* - \zeta_{e_0}^{i+1}.
\end{aligned}$$

It follows that the claim holds for all $e_0 \in E$ and $0 \leq i \leq l$. In particular, for $i = l$, we have that

$$\sum_{\gamma \in \Gamma : e_0 \in \gamma} f_\gamma^l \leq \zeta_{e_0}^* - \zeta_{e_0}^l \leq c(e_0),$$

for all $e_0 \in E$. The rightmost inequality follows from the feasibility of $\boldsymbol{\zeta}^*$.

As the algorithm returns the path flow $\boldsymbol{f}^l$, this implies that the algorithm returns a correct solution.

It remains to show that $val(\boldsymbol{f}^l) = val(\boldsymbol{\zeta^*})$.

Let us consider the quantities $val(\boldsymbol{f}^i)$ and $val(\boldsymbol{\zeta}^i)$ after iteration i of the "while loop", where $0 \leq i \leq l-1$. We observe that during iteration $i+1$ of the "while loop", $val(\boldsymbol{f}^i)$ increases by the same amount that $val(\boldsymbol{\zeta}^i)$ decreases. As this is true for all iterations of the "while loop", we know that $val(\boldsymbol{f}^l) = val(\boldsymbol{\zeta}^*) - val(\boldsymbol{\zeta}^l)$. Thus, if we can show that $val(\boldsymbol{\zeta}^l) = 0$, the result will follow.

Let us consider the final edge flow $\boldsymbol{\zeta}^l$. We define a subset S of vertices, where $v \in S$, provided there is a fully nonzero path from s to v. We include s in this set as well.

As the algorithm terminates with this edge flow, this edge flow must fail the condition of the "while loop". Rather, any path from s to t in G will have at least one edge with zero flow. It follows that t will not be included in S.

If we consider the cut $(S, V \setminus S)$, then it must have outflow $Z^+(S) = 0$. To see this, observe that if there were some $u \in S$ and $v \in V \setminus S$, such that $\zeta^l(u,v) > 0$, then v would be included in S.

On the other hand, we know that by Proposition 2.1.1, $val(\boldsymbol{\zeta}^l) = Z^+(S) - Z^-(S)$. This implies that $val(\boldsymbol{\zeta}^l) \leq 0$. As edge flows always have nonnegative value, we know that $val(\boldsymbol{\zeta}^l) = 0$.

It follows that $val(\boldsymbol{f}^l) = val(\boldsymbol{\zeta}^*)$, thus completing the proof.

□

We now describe an algorithm for converting path flows to edge flows. Unlike the previous procedure, this algorithm does not always run in polynomial time. That being said, we shall see that if the path flow is restricted, then we may circumvent this issue.

Given a path flow $(f_\gamma)_{\gamma\in\Gamma}$ through $G = (V, E)$, the algorithm outputs an edge flow $(\zeta_e)_{e\in E}$. Initially, we set this edge flow to be identically zero. We then process the paths between s and t, which route nonzero flow. For each such path $\gamma \in \Gamma$, we assign the value f_γ to each of the edges of γ; updating the edge flows in $\boldsymbol{\zeta}$ to account for this change. When all the nonzero path flows have been processed, we return the final edge flow.

Algorithm 2.1.2 Path Flow to Edge Flow Conversion Algorithm

Let (G, E) be a directed graph.
Let $c : E \to \mathbb{Q}_{\geq 0}$, and (s, t) a pair of distinct vertices.
Let $(f_\gamma)_{\gamma\in\Gamma}$ be a path flow.
1: Initialize an identically zero edge flow $(\zeta_e)_{e\in E}$.
2: **while** there exists $\gamma' \in \Gamma$ with $f_{\gamma'} \neq 0$ **do**
3: **for** each $e' \in \gamma'$ **do**
4: Set $\zeta_{e'} \leftarrow \zeta_{e'} + f_{\gamma'}$
Set $f_{\gamma'} \leftarrow 0$.
5: Return $(\zeta_e)_{e\in E}$.

We first observe that we can easily characterize the runtime of the above algorithm. Given a path $\gamma \in \Gamma$, we define the *length* of γ, denoted $|\gamma|$, to be its number of edges. With this notation, suppose $\boldsymbol{f} = (f_\gamma)_{\gamma\in\Gamma}$ is the inputed path

flow. We define the *size* of $\boldsymbol{f}$, denoted $|\boldsymbol{f}|$, to be the sum of the lengths of its nonzero paths. That is,

$$|\boldsymbol{f}| = \sum_{\gamma \in \Gamma : f_\gamma \neq 0} |\gamma|.$$

Using this notion of size, we observe the following claim on the runtime of the above algorithm:

Proposition 2.1.4. *The runtime of the above algorithm is $O(|\boldsymbol{f}| + |E|)$.*

Proof. It is clear that at most $O(|E|)$ steps are needed to initialize the edge flow.

Moreover, the runtime of the remaining algorithm is clearly $O(|\boldsymbol{f}|)$.

The result therefore holds.

□

This proposition immediately allows us to characterize when the conversion to edge flow can be done efficiently.

Corollary 2.1.5. *Algorithm 2.1.2 runs in polynomial time in the size of G, if and only if $|\boldsymbol{f}|$ is polynomial in the size of G.*

We now verify the correctness of Algorithm 2.1.2.

Proposition 2.1.6. *Let $\boldsymbol{f} = (f_\gamma)_{\gamma \in \Gamma}$ be a path flow through G. The assignment $\boldsymbol{\zeta} = (\zeta_e)_{e \in E}$ returned by the algorithm, is an edge flow for which $val(\boldsymbol{\zeta}) = val(\boldsymbol{f})$.*

Proof. For clarity, we denote the initial path flow passed to the algorithm as $\boldsymbol{f}^* = (f^*_\gamma)_{\gamma \in \Gamma}$. Similarly, we denote the final edge flow returned by the algorithm as $\boldsymbol{\zeta}^{**} = (\zeta^{**}_e)_{e \in E}$. Moreover, we use $\boldsymbol{f}$ and $\boldsymbol{\zeta}$ to refer to the edge and path flows

as they change throughout the algorithm. The path flow $\boldsymbol{f}$ begins equal to $\boldsymbol{f}^*$ and the edge flow $\boldsymbol{\zeta}$ begins identically zero.

Let us now show that the edge flow returned is in fact valid. We first check that s and t satisfy the necessary conditions. In particular, we claim that both $Z^-(s)$ and $Z^+(t)$ remain zero throughout the algorithm. To see this, observe edge flow is only ever assigned to the edges of the paths of Γ. Since the paths in Γ are directed and simple, they never contain edges directed into s or out of t. It follows that these edges avoid edge flow, thus implying the claim.

We now wish to show that the conversation conditions are satisfied by $\boldsymbol{\zeta}$ throughout the algorithm. These conditions are trivially satisfied at the beginning of the algorithm, as $\boldsymbol{\zeta}$ starts identically zero.

Now consider when the algorithm processes a path $\gamma \in \Gamma$. If a vertex $v \in \gamma$ is neither s nor t, then exactly two of its adjacent edges will be assigned flow. In particular, one will be directed into v, and one will be directed out of it. Moreover, the amount of flow assigned to each of these edges will be equal. This ensures that the edge flow $\boldsymbol{\zeta}$ will continue to satisfy the conversation conditions throughout each iteration of the "while loop". The final edge flow must therefore satisfy the conservation conditions.

We now verify that the capacity conditions hold. Let us consider an edge $e \in E$. By the time the algorithm finishes, it is clear that the flow assigned to ζ_e^{**} will be precisely $\sum_{\gamma \in \Gamma : e \in \gamma} f_\gamma^*$. This implies,

$$\zeta_e^{**} = \sum_{\gamma \in \Gamma : e \in \gamma} f_\gamma^* \leq c(e),$$

as the path flow $\boldsymbol{f}^*$ is assumed to valid. It follows that $\boldsymbol{\zeta}^{**}$ will satisfy all of the relevant capacity conditions.

It remains to check that $\boldsymbol{\zeta}^{**}$ and $\boldsymbol{f}^*$ have the same value. We first observe that at the beginning of the algorithm, $val(\boldsymbol{\zeta}) = 0$. Moreover, each time a path $\gamma \in \Gamma$ is processed, an outgoing edge of s is assigned the path flow f^*_γ. Now, the "while loop" of the algorithm concludes when each path flow has been set to zero. This tells us that,

$$\sum_{e \in \delta^+(s)} \zeta^{**}_e = \sum_{\gamma \in \Gamma} f^*_\gamma.$$

We observe that the left-hand side of this equation is precisely $val(\boldsymbol{\zeta}^{**})$, whereas the right-hand side is exactly $val(\boldsymbol{f}^*)$. This implies that $val(\boldsymbol{\zeta}^{**}) = val(\boldsymbol{f}^*)$, therefore completing the proof.

□

We finish the section with the observation that because of these conversion algorithms, the value of an optimum solution to the edge flow problem is equal to the value of an optimum solution to the path flow problem.

Corollary 2.1.7. *Suppose we are given a directed graph $G = (V, E)$ with capacity function $c : E \to \mathbb{Q}_{\geq 0}$, and source sink pair (s, t). If $\boldsymbol{\zeta}$ is an optimum edge flow, and $\boldsymbol{f}$ is an optimum path flow, then we have that $val(\boldsymbol{\zeta}) = val(\boldsymbol{f})$.*

2.2 Maximum Network Flow and the Ford-Fulkerson Algorithm

In this section, we develop an efficient algorithm for computing maximum edge flows. In particular, we study the Ford-Fulkerson Aglorithm; a famous algorithm based on local search techniques. We then see how we can use the algorithm to also find optimum solutions to the path flow problem; making extensive use of the results from the previous section. Finally, we investigate a variation of the Ford-Fulkerson algorithm, and conclude with a statement of the max-flow min-cut theorem.

Suppose we are given a graph $G = (V, E)$, with capacity function $c : E \to \mathbb{Q}_{\geq 0}$, together with an edge flow $\boldsymbol{\zeta}$. We saw in the previous section that given an (s, t) cut $(S, V \setminus S)$, we may measure the outflow and inflow of S. In particular, we defined $Z^+(S) := \sum_{e \in \delta^+(S)} \zeta_e$ and $Z^-(S) := \sum_{e \in \delta^-(S)} \zeta_e$, to be the outflow and inflow of S respectively.

In addition to inflow and outflow, we can also quantity the *capacity* of an (s, t) cut. Rather, given an (s, t) cut $(S, V \setminus S)$, we define the *capacity* of $(S, V \setminus S)$, denoted $c(S, V \setminus S)$, to be the total capacity of its outgoing edges. Formally, we set $c(S, V \setminus S) = \sum_{e \in \delta^+(S)} c(e)$.

It terms out that given an edge flow $\boldsymbol{\zeta} = (\zeta_e)_{e \in E}$ through G, the capacity of a cut forms an upper bound on the value of $\boldsymbol{\zeta}$.

Proposition 2.2.1. *Let $\boldsymbol{\zeta} = (\zeta_e)_{e \in E}$ be an edge flow through G. If $(S, V \setminus S)$ is an (s, t) cut, then we have that $val(\boldsymbol{\zeta}) \leq c(S, V \setminus S)$.*

Proof. We observe that if $(S, V \setminus S)$ is an (s, t) cut, then by Proposition 2.1.1, we know that $val(\boldsymbol{\zeta}) = Z^+(S) - Z^-(S)$.

Observe that for each edge $e \in \delta^+(S)$, we know that $\zeta_e \leq c(e)$. Thus, we have that, $Z^+(S) \leq c(S, V \setminus S)$.

It follows that $val(\boldsymbol{\zeta}) \leq c(S, V \setminus S)$.

□

Our goal now will be to establish an algorithm for computing edge flows of maximum value. Let us suppose we fix an arbitrary (s, t) cut $(S, V \setminus S)$. As a consequence of the above proposition, we know that *any* edge flow through G will have its value bounded above by $c(S, V \setminus S)$. Thus, if we can compute an edge flow $\boldsymbol{\zeta}$ for which $val(\boldsymbol{\zeta}) = c(S, V \setminus S)$, we will know that $\boldsymbol{\zeta}$ is of maximum value.

We now describe an algorithm for computing maximum edge flows, known as the Ford-Fulkerson algorithm. At a high level, the algorithm begins with an identically zero edge flow, say $\boldsymbol{\zeta}$, which it continually augments. The purpose of each augmentation is to increase the amount of flow that leaves the source. Eventually the algorithm will no longer be able to perform anymore augmentations, at which point the procedure will terminate, and $\boldsymbol{\zeta}$ will be optimum.

The most natural way to perform an augmentation is to choose a path from s to t, for which nonzero flow may be routed through. It turns out that this greedy approach will not always work: there may be no paths for which we can route flow through, despite $\boldsymbol{\zeta}$ not being optimum. As this is the case, we need a way to undo previous augmentations; that is, reduce the amount of flow currently passing through an edge.

It will be useful to design a data structure which we can use to track the status of the edges of G. In other words, if we have already constructed an edge

flow $\boldsymbol{\zeta}$, then for each edge $e \in E$, we record the amount that ζ_e can be changed. Toward this goal, we must store two quantities for each $e \in E$: the amount that we can increase the flow through e, namely $c(e) - \zeta_e$; as well as ζ_e, the amount we can decrease the flow through e. As a consequence of these requirements, it will be convenient to use a *multigraph* data structure to store the information regarding these edges.

Let us denote $G_{\boldsymbol{\zeta}} = (V_{\boldsymbol{\zeta}}, E_{\boldsymbol{\zeta}})$ to be the *residual* multigraph of G and $\boldsymbol{\zeta}$. Each edge of $G_{\boldsymbol{\zeta}}$ is defined as a 2-tuple: the first coordinate contains a pair of vertices, whereas the second coordinate contains an integer. We use this second coordinate to keep track of the multiple edges that may arise between a vertex pair. Observe the following definition of $G_{\boldsymbol{\zeta}}$:

- $G_{\boldsymbol{\zeta}}$ has the same vertex set as G.
- For each edge $e = (u, v) \in E$, if $\zeta_e < c(e)$, then we add $(e, 1)$ to $E_{\boldsymbol{\zeta}}$. An edge of $G_{\boldsymbol{\zeta}}$ is referred to it as a *forward edge*, when it originates from G in this way. Clearly, these types of edges run in the same direction as in the original graph G.
- For each edge $e = (u, v) \in E$, if $\zeta_e > 0$, then we let $e' = (v, u)$ and add the edge $(e', -1)$ to $G_{\boldsymbol{\zeta}}$. Edges of $G_{\boldsymbol{\zeta}}$ added in this manner are referred to as *backward edges*. They run in the opposite direction of the edges in which they originate from.

In addition to specifying the structure of the residual graph, we also associate a *residual* capacity function, denoted $c_{\boldsymbol{\zeta}}$, to the edges of $G_{\boldsymbol{\zeta}}$. We define this function as follows:

- If $(e, l) \in E_{\boldsymbol{\zeta}}$ is a forward edge ($l = 1$), then we set $c_{\boldsymbol{\zeta}}(e, l) = c(e) - \zeta_e$. We may think of this quantity as specifying the remaining potential for flow through e.
- If $(e, l) \in E_{\boldsymbol{\zeta}}$ is a backward edge ($l = -1$), then let e' be the reversal of e. We set $c_{\boldsymbol{\zeta}}(e, l) = \zeta_{e'}$. This quantity may be thought of as the amount we can reduce the flow going through e.

Let us now define a subroutine critical to the Ford-Fulkerson algorithm. The subroutine takes in the currently defined edge flow $\boldsymbol{\zeta}$, the residual capacity function $c_{\boldsymbol{\zeta}}$, and a simple path γ from s to t in $G_{\boldsymbol{\zeta}}$. The *bottleneck* of γ, namely $\min_{(e,l)\in\gamma} c_{\boldsymbol{\zeta}}(e, l)$, is then computed and stored in a variable ε. We then use the path γ, together with ε, to augment the flow $\boldsymbol{\zeta}$ through G. For each edge $(e, l) \in \gamma$, if (e, l) is a forward edge, then ζ_e is increased by ε. Similarly, if $e = (u, v)$ and (e, l) is a backward edge, then $\zeta_{(v,u)}$ is reduced by ε. The subroutine then returns the updated flow $\boldsymbol{\zeta}$.

Let us now formally describe this subroutine. We refer to executions of this subroutine as *augmentations*.

Algorithm 2.2.1 Augmentation Algorithm

Let γ be a simple path from s to t in $G_{\boldsymbol{\zeta}}$.
Let $c_{\boldsymbol{\zeta}} : E \mapsto \mathbb{Q}_{\geq 0}$ be the residual capacity of G.
Let $\boldsymbol{\zeta}$ be an edge flow through G.
1: Initialize a copy $\boldsymbol{\zeta}'$ of $\boldsymbol{\zeta}$.
2: Set $\varepsilon \leftarrow \min_{(e,l)\in\gamma} c_{\boldsymbol{\zeta}}(e,l)$.
3: **for** $(e,l) \in \gamma$ **do**
4: **if** $l = 1$ **then**
5: Set $\zeta_e' \leftarrow \zeta_e' + \varepsilon$.
6: **else**
7: Set $e = (u,v)$.
8: Set $\zeta_{(v,u)}' \leftarrow \zeta_{(v,u)}' - \varepsilon$
9: Return $\boldsymbol{\zeta}'$.

We now formulate a result involving the augmentation algorithm.

Proposition 2.2.2. *Let $\boldsymbol{\zeta}$ be the edge flow passed to Algorithm 2.2.1. The algorithm takes $O(|E|)$ time to return a valid edge flow, $\boldsymbol{\zeta}'$. Moreover, if $\boldsymbol{\zeta} = (\zeta_e)_{e\in E}$ and $c : E \to \mathbb{Q}_{\geq 0}$ take on integral values, then so will $\boldsymbol{\zeta}'$.*

Let us now introduce the entirety of the Ford-Fulkerson algorithm. We observe that each path in $G_{\boldsymbol{\zeta}}$ has *nonzero residual capacity*; rather, all its edges have nonzero capacity, with respect to $c_{\boldsymbol{\zeta}}$. We also denote $\Gamma_{\boldsymbol{\zeta}}$ to be the simple paths between s and t in $G_{\boldsymbol{\zeta}}$.

Algorithm 2.2.2 Ford-Fulkerson Maximum Flow Algorithm

Let (G, E) be a directed graph.
Let $c : E \to \mathbb{Q}_{\geq 0}$, and (s, t) a pair of distinct vertices.
1: Initialize an identically zero edge flow $\boldsymbol{\zeta} = (\zeta_e)_{e \in E}$.
2: Compute $G_{\boldsymbol{\zeta}}$ and $c_{\boldsymbol{\zeta}} : E \to \mathbb{Q}_{\geq 0}$.
3: **while** there exists a path from s to t in $G_{\boldsymbol{\zeta}}$ **do**
4: Choose $\gamma \in \Gamma_{\boldsymbol{\zeta}}$ using an arbitrary algorithm.
5: Run Algorithm 2.2.1 with inputs γ, $c_{\boldsymbol{\zeta}}$ and $\boldsymbol{\zeta}$.
6: Set $\boldsymbol{\zeta}'$ to be the result of this algorithm.
7: Update $G_{\boldsymbol{\zeta}}$ and $c_{\boldsymbol{\zeta}}$ with the new edge flow $\boldsymbol{\zeta}'$.
8: Return $\boldsymbol{\zeta} = (\zeta_e)_{e \in E}$.

Observe that in Step 4 of the above algorithm, we do not explicitly specify how the computation of γ is done. In this sense, we may think of the Ford-Fulkerson algorithm as specifying a general framework for a whole collection of algorithms. These algorithms all make use of the same general structure, but differ in how they choose which paths to augment. When we prove claims about the Ford-Fulkerson algorithm, we are really proving these claims for *any* algorithm within this collection. That being said, we state all of these claims in terms of Algorithm 2.2.2.

We begin by proving that Algorithm 2.2.2 terminates with a valid edge flow. In order for this claim to be true, we need to ensure that the capacity function, $c : E \to \mathbb{Q}_{\geq 0}$, takes on integral values. Let us refer to an edge flow which takes on integral values as an *integral edge flow*.

Proposition 2.2.3. *Let $G = (V, E)$ be a directed graph with capacity function $c : E \to \mathbb{Z}_{\geq 0}$. Moreover, let $\alpha := \sum_{e \in \delta^+(s)} c(e)$. Algorithm 2.2.2 returns an integral edge flow in time $O(\alpha\,|E|)$.*

Proof. For clarity, we denote $\boldsymbol{\zeta}^* := \mathbf{0}$ as the initial edge flow. Moreover, we refer to $\boldsymbol{\zeta}$ as the edge flow as it changes throughout the algorithm. Clearly, $\boldsymbol{\zeta}$ begins as equal to $\boldsymbol{\zeta}^*$. Our goal will be to show that at most α iterations of the "while loop" occur.

We first observe that when $\boldsymbol{\zeta}$ is augmented by the "while loop", its value increases by a positive integer. This is because at every step of the algorithm, $\boldsymbol{\zeta}$ is an integral flow by Proposition 2.2.1. Moreover, the algorithm only performs augmentations when they are guaranteed to increase the value of the edge flow.

Let us consider the (s,t) cut S, where $S := \{s\}$. We observe that $(S, V \backslash S)$ has capacity $c(S, V \setminus S) = \sum_{e \in \delta^+(s)} c(e)$. Now, by Proposition 2.2.1, we know that at any point in the algorithm, $val(\boldsymbol{\zeta}) \leq c(S, V \setminus S)$. On the other hand, each iteration of the "while loop" increases the value of $\boldsymbol{\zeta}$ by a positive integer. It follows that there are at most α iterations of the "while loop". As each iteration by "while loop" takes time $O(|E|)$, we know that the running time of the algorithm will be $O(\alpha\,|E|)$. This completes the proof.

□

We are now ready to prove that the Ford-Fulkerson algorithm returns an optimum edge flow. As before, the capacity function $c : E \to \mathbb{Q}_{\geq 0}$ must take on integer values in order for the algorithm to terminate. We again denote $\alpha = \sum_{e \in \delta^+(s)} c(e)$.

Theorem 2.2.4. *Given a directed graph $G = (V, E)$ with capacity function $c : E \to \mathbb{Z}_{\geq 0}$, the Ford-Fulkerson algorithm returns an integral optimum edge flow $\boldsymbol{\zeta} = (\zeta_e)_{e \in E}$ in time $O(\alpha\,|E|)$.*

Proof. We have already proven that the solution returned by the algorithm is integral and takes $O(\alpha\,|E|)$ to compute. It remains to show that the solution returned is optimum.

Let us refer to $\boldsymbol{\zeta}$ as the final edge flow returned by algorithm. Moreover, we again denote $\Gamma_{\boldsymbol{\zeta}}$ to be the set of simple paths from s to t in the augmented graph $G_{\boldsymbol{\zeta}}$.

We now consider a subset S of the vertices of G. For each $v \in V$, if there is a path from s to v in $G_{\boldsymbol{\zeta}}$, then we place v in S. By convention, we also place s in S.

We observe that before the algorithm terminates, $\boldsymbol{\zeta}$ must fail the conditions of the "while loop". This means that no path from s to t will exist in $G_{\boldsymbol{\zeta}}$, so t will not be placed in S. Thus, $(S, V \setminus S)$ forms a partition of the vertex set of G.

Our goal now will be to show that the capacity of $(S, V \setminus S)$ is in fact equal to the value of $\boldsymbol{\zeta}$. Rather, we prove that $c(S, V \setminus S) = val(\boldsymbol{\zeta})$. By Proposition 2.2.1, all edge flows through G must have their value bounded above by $c(S, V \setminus S)$. As a consequence, this will prove that $\boldsymbol{\zeta}$ is optimum.

Suppose that $e \in E$ is an edge present in $\delta^+(S)$; that is, it runs from a vertex in S to a vertex in $V \setminus S$. We claim that the edge flow through e is equal to its capacity. To show this, assume otherwise. Rather, suppose that $\zeta_e < c(e)$. In this case, forward edge $(e, 1)$ must have been added to $E_{\boldsymbol{\zeta}}$ with residual capacity $c_{\boldsymbol{\zeta}}(e, 1) = c(e) - \zeta_e > 0$. If $e = (u, v)$, then since $u \in S$, there must be a path γ in $G_{\boldsymbol{\zeta}}$ between s and u. Observe that we may add edge $(e, 1)$ to γ, forming a path from s to v in $G_{\boldsymbol{\zeta}}$. As v was assumed to not be in S, this is a contradiction. The claim therefore holds.

Similarly, if $e \in E$ is an edge present in $\delta^-(S)$, then it must have edge flow equal to zero. To see this, assume that $\zeta_e > 0$. In this case, backward edge $(e', -1)$ must have been added to $E_{\boldsymbol{\zeta}}$, where e' is the reversal of e. Moreover, the residual capacity $c_{\boldsymbol{\zeta}}$ of $(e', -1)$ will have been set to $\zeta_e > 0$. Once again, assume that $e = (u, v)$. In this case, $v \in S$, as e is directed into s. In consequence, we may find a path from s to u in $G_{\boldsymbol{\zeta}}$, thus contradicting the membership of u. This concludes the proof of the claim.

These claims imply that the outflow of S, namely $Z^+(S)$, is equal to $c(S, V \setminus S)$. Moreover, the inflow of S, denoted by $Z^-(S)$, is equal to zero. By Proposition 2.1.1 this tells us that,

$$val(\boldsymbol{\zeta}) = Z^+(S) - Z^-(S) = c(S, V \setminus S).$$

This completes the proof of the theorem.

□

We now see how we can use this algorithm to simultaneously solve the path flow problem (Problem 2.1.1) in an efficient amount of time. We refer to a path flow which exclusively takes on integral values as an *integral path flow*.

Theorem 2.2.5. *Given a directed graph $G = (V, E)$, source sink pair (s, t) and a capacity function $c : E \to \mathbb{Z}_{\geq 0}$, an integral optimum path flow $\boldsymbol{f} = (f_\gamma)_{\gamma \in \Gamma}$ may be computed in time $O(\alpha\,|E| + |E|^2)$. In this context, Γ refers to the set of simple paths between s and t, and $\alpha = \sum_{e \in \delta^+(s)} c(e)$.*

Proof. In order to compute an optimum path flow, we first compute an optimum edge flow $\boldsymbol{\zeta}$ using Algorithm 2.2.2. By Theorem 2.2.4, this computation can be done in time $O(\alpha\,|E|)$.

Once we have established this edge flow, we use Algorithm 2.1.1 to convert $\boldsymbol{\zeta}$ to a path flow $\boldsymbol{f}$. By Proposition 2.1.3, this conversion may be done in time $O(|E|^2)$. Moreover, the value of $\boldsymbol{f}$ will be equal to the value of $\boldsymbol{\zeta}$, so $\boldsymbol{f}$ will be optimum (see Corollary 2.1.7). The result thus follows.

□

2.2.1 Edmonds-Karp Maximum Flow Algorithm

The above algorithms provide us with fairly efficient ways to solve each of our network flow problems. That being said, we measure the size of the capacity function as the total number of bits needed to store the numbers $(c(e))_{e\in E}$. If we assume that each positive integer $c(e)$ is stored in binary notation, then the size of $c(e)$ is $\lfloor \log_2(c(e)) \rfloor + 1$. This implies that the size of $c : E \to \mathbb{Z}_{\geq 0}$ is $\sum_{e\in E}(\lfloor \log_2(c(e)) \rfloor + 1)$.

If we once again denote $\alpha = \sum_{e\in\delta^+(s)} c(e)$, then suppose α is close to the value of $\sum_{e\in E} c(e)$. In this case, our algorithms will run in time *exponential* in the size of c, provided *alpha* many augmentations occur. Thus, if the outgoing edges of s have large capacity, then our algorithms may not perform well.

To rectify this issue, we consider a specific implementation of the Ford-Fulkerson algorithm. Currently, the algorithm maintains an edge flow $\boldsymbol{\zeta}$, and periodically searches for paths between s and t to augment. In particular, it chooses paths in Γ_ζ, as they can have nonzero flow routed through them. If there

exist multiple paths of this form, the algorithm does not explicitly say which path to choose. Our implementation will choose the candidate which has the fewest number of edges possible (breaking ties arbitrarily).

Algorithm 2.2.3 Edmonds-Karp Maximum Flow Algorithm

Let (G, E) be a directed graph.
Let $c : E \to \mathbb{Q}_{\geq 0}$, and (s, t) a pair of distinct vertices.
1: Initialize an identically zero edge flow $\boldsymbol{\zeta} = (\zeta_e)_{e \in E}$.
2: Compute $G_{\boldsymbol{\zeta}}$ and $c_{\boldsymbol{\zeta}} : E \to \mathbb{Q}_{\geq 0}$.
3: **while** there exist a path from s to t in $G_{\boldsymbol{\zeta}}$ **do**
4: Choose $\gamma \in \Gamma_{\boldsymbol{\zeta}}$, with the fewest possible edges.
5: Run Algorithm 2.2.1 with inputs γ,$c_{\boldsymbol{\zeta}}$ and $\boldsymbol{\zeta}$.
6: Set $\boldsymbol{\zeta}'$ to be the result of this algorithm.
7: Update $G_{\boldsymbol{\zeta}}$ and $c_{\boldsymbol{\zeta}}$ with the new edge flow $\boldsymbol{\zeta}'$
8: Return $\boldsymbol{\zeta} = (\zeta_e)_{e \in E}$.

Our goal will be to show that the number of augmentations the algorithm performs is polynomial in the size of G. Let us begin by recalling some definitions from graph theory:

If $G = (V, E)$ is a directed graph, and (s, t) is a pair of vertices of G, then we define the *distance* between s and t, denoted $dist_G(s, t)$, as the number of edges in the shortest (directed) path between s and t. If s is not connected to t in G, then this value is set to positive infinity.

Let us suppose that γ is a shortest path between s and t. We may form a new path γ^R, by reversing every edge of γ. This path is known as the *reversal* of γ. If G is a multigraph, then each edge (e, l) of γ is replaced by $(e^R, -l)$ in γ^R.

Now consider graph $G' = (V', E')$, where $G' := G \cup \gamma^R$. Rather, G' has the same vertex set as G, but its edges include E, together with the edges found in γ^R.

Lemma 2.2.6. *Suppose that $u^*, v^* \in \gamma$ and $dist_G(s, u^*) \leq dist_G(s, v^*)$. It follows that $dist_G(u^*, v^*) = dist_{G'}(u^*, v^*)$.*

Proof. Let us first consider the case when $u^* = s$ and $v^* = t$.

We know that $G \subseteq G'$ by definition. Thus, any path in G is present in G', so we know that $dist_{G'}(s, t) \leq dist_G(s, t)$.

We now prove the reverse inequality. Let us suppose that γ is the path between s and t used to construct G'. Moreover, let us suppose that γ^R is the reversal of γ. That is, each edge of γ^R is the reversal of an edge of γ.

If p is a shortest path between s and t in G', then we claim that p will not contain any edges of γ^R. To see this, let us assume otherwise. We denote $e^R = (v, u)$ as the first edge of p which lies in γ^R. Observe that the edge $e = (u, v)$ lies in the path γ.

Let us split the path p into three parts. We denote p_1 as the piece between s and v, p_2 as the piece between v and u, and p_3 as the piece between u and s.

Observe that since e^R is the first edge of p in γ^R, we know that p_1 is entirely contained in G. This means that $|p_1| = dist_G(s, v)$, as the path p is of minimum length.

On the other hand, if we denote γ_1 as the piece of γ between s and v, then $|\gamma_1| = dist_G(s, u)$. This is immediate, as γ is a shortest path in G.

If $\gamma_1 p_3$ is the concatenation of γ_1 followed by p_3, then we observe that,

$$|\gamma_1 p_3| = dist_G(s, u) + |p_3| < dist_G(s, v) + |p_2| + |p_3| = |p|$$

as $dist_G(s, v) = dist_G(s, u) + 1$, and $|p_1| = dist_G(s, v)$.

However, $\gamma_1 p_3$ is a path from s to t in G'. Since p was assumed to be the shortest path from s to t in G', this is a contradiction.

It follows that p contains no edges of γ', and so is contained in G. Thus, $dist_G(s,t) \leq |p| = dist_{G'}(s,t)$. Together with the first inequality proven, this implies that,

$$dist_G(s,t) = dist_{G'}(s,t). \tag{2.2.1}$$

We now consider the general case when u^* and v^* are arbitary vertices of γ. It is obvious once again that, $dist_{G'}(u^*,v^*) \leq dist_G(u^*,v^*)$, as $G \subseteq G'$. Let us assume that this inequality is in fact strict. In this case, we see that,

$$\begin{aligned} dist_{G'}(s,t) &= dist_{G'}(s,u^*) + dist_{G'}(u^*,v^*) + dist_{G'}(v^*,t) \\ &< dist_G(s,u^*) + dist_G(u^*,v^*) + dist_G(v^*,t) \\ &= dist_G(s,t), \end{aligned}$$

as γ is a shortest path between s and t in both G and G'. This contradictions Equation 2.2.1, so the general case for u^* and v^* must hold.

□

We shall also need a strengthening of Lemma 2.2.6. In the notation used above, let $G = (V,E)$ be a directed graph and (s,t) be a pair of distinct vertices of G. Moreover, let $\gamma_1, \ldots \gamma_k$ be k simple paths between s and t of minimal length. We may form $G' := G \cup \bigcup_{i=1}^{k} \gamma_i$. Observe the following lemma:

Lemma 2.2.7. *Suppose that $u^*, v^* \in \gamma_i$ where $1 \leq i \leq k$, and $dist_G(s,u^*) \leq dist_G(s,v^*)$. It follows that $dist_G(u^*,v^*) = dist_{G'}(u^*,v^*)$.*

Proof. Let us first prove the lemma for vertices s and t. We observe that by applying Lemma 2.2.6 to G and γ_1, we have that $dist_G(s,t) = dist_{\gamma_1^R \cup G}(s,t)$. Similarly, we may the lemma to $G \cup \gamma_1$ and γ_2, to achieve a similar inequality. The final inequality, $dist_G(s,t) = dist_{G'}(s,t)$, can be easily seem after k applications of Lemma 2.2.6 in this manner.

In order to extend this argument to $u^*, v^* \in \gamma_i$, for which

$$dist_G(s,u^*) \leq dist_G(s,v^*),$$

we may apply the same argument as seen in the second part of the proof of Lemma 2.2.6.

□

Let us now suppose that we pass a directed graph G, a capacity function $c : E \to \mathbb{Z}_{\geq 0}$, and a source sink pair (s,t) to Algorithm 2.2.3. Algorithm 2.2.3 is a specific implementation of the Ford-Fulkerson algorithm, so it must terminate by Proposition 2.2.3. Thus, we may assume that Algorithm 2.2.3 performs $q \geq 0$ augmentations when passed the above inputs.

It will be convenient to label the edge flow as it is processed throughout the algorithm. Let $\boldsymbol{\zeta}^0 := \mathbf{0}$, be the edge flow after it is initialized. Moreover, let $\boldsymbol{\zeta}^i$ be the edge flow after i augmentations (iterations of the algorithm's "while loop"). The edge flow $\boldsymbol{\zeta}^q$ will be the output of the algorithm.

We use $G_{\boldsymbol{\zeta}^i}$ to denote the residual multigraph of G with respect to $\boldsymbol{\zeta}^{\boldsymbol{i}}$, where $0 \leq i \leq q$. Similarly, we denote $c_{\boldsymbol{\zeta}^i}$ as the residual capacity function. For each

$0 \leq i \leq q$, we denote the distance between s and t in $G_{\boldsymbol{\zeta}^i}$ by $dist_{G_{\boldsymbol{\zeta}^i}}(s,t)$. We shorten the notations of $G_{\boldsymbol{\zeta}^i}$ and $c_{\boldsymbol{\zeta}^i}$ to G_i and c_i when the context is clear.

For each $0 \leq i \leq q-1$, we refer to γ_i as the path chosen by the algorithm in iteration $i+1$ of the "while loop". In this notation, γ_0 is the first path chosen by the algorithm, and γ_{q-1} is the final path chosen by the algorithm.

Lemma 2.2.8. *For $0 \leq i \leq q-1$, let us denote γ_i^R as the reversal of γ_i; G_i is a multigraph, so the labellings of γ_i are negated in γ_i^R. If we form the multigraph $G_i \cup \gamma_i^R$, then*

$$G_{i+1} \subseteq G_i \cup \gamma_i^R.$$

Proof. Given graph G_i, we form G_{i+1} by first choosing a path γ_i between s and t in G_i. We then augment the graph (Algorithm 2.2.1), using G_i, γ_i and c_i as inputs. This is done as follows:

Let $\varepsilon := \min_{(e,l)\in\gamma_i} c_i(e,l)$ be the bottleneck of γ_i. For each edge (e,l) present in γ_i, if (e,l) is a forward edge ($l=1$) then set $\zeta_e^{i+1} = \zeta_e^i + \varepsilon$. Similarly, if (e,l) is a backward edge, set $\zeta_{e^R}^{i+1} = \boldsymbol{\zeta}_i(e^R) - \varepsilon$, where e^R is the reversal of e. The augmentation algorithm then returns the updated flow $\boldsymbol{\zeta}^{i+1}$.

We observe that once the flow $\boldsymbol{\zeta}^{i+1}$ is returned, the residual graph G_{i+1} is updated from $\boldsymbol{\zeta}^{i+1}$. We may process the edges of γ_i to see how these graphs differ.

First, suppose (e,l) is a forward edge of γ_i. If $\zeta_e^{i+1} = c(e)$, then (e,l) will not be present in G_{i+1}. On the other hand, if $\zeta_e^i = 0$, then $\zeta_e^{i+1} > 0$, so backward edge $(e^R, -l)$ will be present in G_{i+1}, where e^R is the reversal of e.

Let us now consider when (e, l) is a backward edge of γ_i. We denote e^R as the reversal of e. If $\zeta_{e^R}^{i+1} = 0$, then (e, l) will be not be present in G_{i+1}. However, if $\zeta_{e^R}^{i} = c(e^R)$, then $\zeta_{e^R}^{i+1} < c(e^R)$, so forward edge $(e^R, -l)$ will be present in G_{i+1}.

Using these observations, it is clear that $G_{i+1} \subseteq G_i \cup \gamma_i^R$.

□

Proposition 2.2.9. *Let $0 \leq i \leq q-1$. It follows that, $dist_i(s,t) \leq dist_{i+1}(s,t)$.*

Proof. Let γ_i and γ_i^R be as above. We form the graph $G' := G_i \cup \gamma_i^R$

By Lemma 2.2.6, since γ_i was a shortest path in G_i, we know that,

$$dist_{G'}(s,t) = dist_{G_i}(s,t)$$

On the other hand, Lemma 2.2.8 implies that $G_{i+1} \subseteq G'$. Together with the above equation, this implies that,

$$dist_{G_i}(s,t) \leq dist_{G_{i+1}}(s,t).$$

The result therefore holds.

□

As a consequence of this proposition, it follows that $(dist_i(s,t))_{i=1}^q$ forms a nondecreasing sequence of numbers. In particular, we may assume that β distinct values are assumed by this sequence, where $0 \leq \beta \leq q$. Let us denote these values as $d_1 < d_2 < \ldots < d_\beta$.

We may partition the iterations of the "while loop" by the values in which they take on. For $j = 1, \ldots \beta$, we denote $I_j := \{0 \leq i \leq q : dist_i(s,t) = d_j\}$. We

refer to the iterations in I_j as *phase* j of the algorithm. In this way, β phases of the algorithm occur. We first observe the following proposition regarding the size of β.

Proposition 2.2.10. *There are at most* $|V| - 1$ *phases in the execution of Algorithm 2.2.3. Rather, in the above notation,* $\beta \leq |V| - 1$.

Proof. We may assume that $|V| \geq 2$, as G must have at least two vertices in order to have a distinct source sink pair (s, t).

If $\beta < 2$, then the claim holds trivially, as $|V| \geq 2$. Let us now assume that $\beta \geq 2$.

Fix $1 \leq j \leq \beta - 1$. During phase j, the algorithm invokes augmentations using simple paths of length d_j. When the algorithm moves into phase $j + 1$, the algorithm will instead use simple paths of lenth d_{j+1}. As the longest path used in any augmentation is $|V| - 1$, there may be at most $|V| - 1$ phases. Rather, $\beta \leq |V| - 1$.

The result therefore follows.

□

Given $1 \leq j \leq \beta$, we may refer to the *length of phase* j as $|I_j|$. Our next goal will be to bound the length of each phase of the algorithm. We first prove a fact that will be essential in achieving these bounds.

Proposition 2.2.11. *Let* $i^*, i \in I_j$, *where* $1 \leq j \leq \beta$. *If* $i < i^*$, *then* γ_{i^*} *will be edge disjoint from* γ_i^R.

Proof. Let us assume that γ_{i^*} and γ_i^R are *not* edge disjoint. This implies there is an edge $e = (u, v)$ of γ_i, such that $e^R = (v, u)$ is in γ_{i^*}. Now consider the graph $G' := G_i \cup \bigcup_{k=i+1}^{i^*} \gamma_k^R$. By repeated applications of Lemma 2.2.8, we know that G_{i^*} will be included in G', provided we ignore the labellings of each graph.

By Lemma 2.2.7, we know that,

$$dist_{G'}(s, u) = dist_{G_i}(s, u),$$

and,

$$dist_{G'}(v, t) = dist_{G_i}(v, t).$$

Moreover, as $G_{i^*} \subseteq G'$,

$$dist_{G'}(s, u) \leq dist_{G_{i^*}}(s, u),$$

and,

$$dist_{G'}(v, t) \leq dist_{G_{i^*}}(v, t).$$

We now split the path γ_{i^*} into pieces $\gamma_{i^*}^1, \gamma_{i^*}^2$ and $\gamma_{i^*}^3$. The path $\gamma_{i^*}^1$ is the portion from s to v, $\gamma_{i^*}^2$ is the portion from v to u, and $\gamma_{i^*}^3$ is the portion from u to t. We observe that since γ_{i^*} is a shortest path in G_{i^*}, $|\gamma_{i^*}^1| = dist_{G_{i^*}}(s, v)$, $|\gamma_{i^*}^2| = dist_{G_{i^*}}(v, u)$ and $|\gamma_{i^*}^3| = dist_{G_{i^*}}(u, t)$.

Applying the above inequalities to the pieces of γ_{i^*}, we observe that,

$$\begin{aligned}
|\gamma_{i^*}| &= |\gamma_{i^*}^1| + |\gamma_{i^*}^2| + |\gamma_{i^*}^3| \\
&\geq dist_{G'}(s,v) + dist_{G_{i^*}}(v,u) + dist_{G'}(u,t) \\
&= dist_{G_i}(s,u) + 1 + dist_{G^{i^*}}(v,u) + dist_{G_i}(u,t) \\
&= dist_{G'}(s,t) + 1 + dist_{G^{i^*}}(v,u) \\
&> dist_{G'}(s,t) \\
&= |\gamma_i|,
\end{aligned}$$

as $dist_{G'}(s,v) = dist_{G_i}(s,v) = dist_{G_i}(s,u) + 1$. This allows us to conclude that γ_{i^*} has greater length than γ_i.

On the other hand, we assumed that both iterations i and i^* are in the same phase. This means that γ_{i^*} and γ_i should have the same length. As we have shown they do not, this yields a contradiction. It follows that γ_{i^*} and γ_i^R are edge disjoint, proving the original claim.

□

Before we prove a claim which limits the length of the phases of the Edmonds-Karp Algorithm, we introduce some definitions. Given $0 \leq i \leq q-1$, we say that edge (e,l) is *removed at time* i, provided $(e,l) \in G_i$, yet $(e,l) \notin G_{i+1}$. By Lemma 2.2.8, it is necessary that path γ_i contains (e,l) in order for this to occur. Similarly, we say that edge (e,l) is *added at time* i, provided $(e,l) \notin G_i$, yet $(e,l) \in G_{i+1}$. It is necessary that path γ_i contains $(e^R,-l)$ in order for this to occur.

If we restrict ourselves to a specific phase, then it becomes easy to track which edges have been added and removed:

Lemma 2.2.12. *Let $1 \leq j \leq \beta$ and $i \in I_j$. Suppose that (e, l) is removed (added) at time i. If $i^* \in I_j$, and $i < i^*$, then (e, l) cannot be added (removed) at time i^*.*

Proof. We prove each claim in sequence.

If (e, l) is removed from G_i at time i, then it must have been a member of path γ_i by the above remarks. Similarly, in order for it to be added to G_{i^*}, its reversal, namely $(e^R, -l)$, must be a member of path γ_{i^*}. By Proposition 2.2.11, $(e^R, -l)$ cannot occur in γ_{i^*}, so this may not happen.

Now assume that (e, l) is added to G_i at iteration i. In this case, edge $(e^R, -l)$ must have been present in γ_i. In order for it to be removed at iteration i^*, (e, l) must have occured in γ_{i^*}. Once again by Proposition 2.2.11, this cannot occur. It follows that (e, l) cannot be removed at iteration i^*.

□

With this lemma, we are ready to bound the length of the phases of the algorithm.

Proposition 2.2.13. *Each phase of the algorithm has length at most $2|E|$. Rather, for all $1 \leq j \leq \beta$, $|I_j| \leq 2|E|$.*

Proof. For simplicity, we assume that we are working with the first phase of the algorithm ($j = 1$). The argument will naturally extend to any arbitrary phase.

For each $0 \leq i^* \leq |I_1| - 1$, let us denote $\mathcal{R}_{i^*}$ as the edges which are removed from G_i at times $i = 0, \ldots, i^*$. By Lemma 2.2.12, $(\mathcal{R}_{i^*})_{i^*=0}^{|I_1|-1}$ forms a nondecreasing sequence with respect to inclusion.

Similarly, we denote $\mathcal{A}_{i^*}$ as the edges which are added to G_{i^*} at times $i = 0, \ldots, i^*$. Lemma 2.2.12, implies that $(\mathcal{A}_{i^*})_{i^*=0}^{|I_j|-1}$ forms a nondecreasing sequence with respect to inclusion.

Observe that for each $i^* = 1, \ldots, |I_j| - 1$, we know that $e(G_{i^*}) = (e(G_0) \cup \mathcal{A}_{i^*-1}) \setminus \mathcal{R}_{i^*-1}$. That is, the edges of G_{i^*} includes the edges of G_0 which are not removed in the first i^* augmentations, together with the edges that are added after the first i^* augmentations.

We observe that if $0 \leq i^* \leq |I_1| - 1$, then $\mathcal{A}_{i^*} \subseteq \cup_{i=0}^{i^*} e(\gamma_i^R)$, by repeated applications of Lemma 2.2.8. On the other hand, if $i^{**} \leq |I_1| - 1$ and $i^* < i^{**}$, then we know that by Proposition 2.2.11, $\gamma_{i^{**}}$ is edge disjoint from $\cup_{i=0}^{i^*} e(\gamma_i^R)$. It follows that the edges of $\gamma_{i^{**}}$ are disjoint from $\mathcal{A}_{i^*}$ for all $0 \leq i^* < i^{**}$.

Let $1 \leq i' \leq |I_1| - 1$, and consider when path $\gamma_{i'}$ is chosen for augmentation. First observe that the edges of $\gamma_{i'}$ are contained in $e(G_i) = (e(G_0) \cup \mathcal{A}_{i'-1}) \setminus \mathcal{R}_{i'-1}$. Moreover, the edges of $\gamma_{i'}$ are disjoint from $\mathcal{A}_{i'-1}$. These two statements imply that,

$$e(\gamma_{i'}) \subseteq e(G_0) \setminus \mathcal{R}_{i'-1}.$$

Now, during the augmentation of $\gamma_{i'}$, at least one edge of $\gamma_{i'}$ will be removed. Moreover, any edge which is removed at this time *cannot* belong to $\mathcal{R}_{i'-1}$, by the preceding equation. This implies that $\mathcal{R}_{i'}$ will be strictly larger than $\mathcal{R}_{i'-1}$; rather,

$\mathcal{R}_{i'-1} \subsetneq \mathcal{R}_{i'}$. Thus, the sequence $(|\mathcal{R}_i|)_{i=0}^{|I_1|-1}$ is *strictly* increasing. In particular, $|\mathcal{R}_{|I_1|-2}| \geq |I_1| - 1$, as $|\mathcal{R}_0| \geq 1$.

Let us now consider the final augmentation of the first phase. As this point, the edges of path $\gamma_{|I_1|-1}$ are taken from $e(G_0) \setminus \mathcal{R}_{|I_1|-2}$. Now, $|e(G_0)| \leq 2|E|$, so $e(G_0) \setminus \mathcal{R}_{|I_1|-2}$ has size at most $2|E| - |\mathcal{R}_{|I_1|-2}| \leq 2|E| - |I_1| + 1$. Moreover, $\gamma_{|I_1|-1}$ takes at least one edge from $e(G_0) \setminus \mathcal{R}_{|I_1|-2}$, so

$$1 \leq |e(G_0) \setminus \mathcal{R}_{|I_1|-2}| \leq 2|E| - |I_1| + 1.$$

It follows that,

$$|I_1| \leq 2|E|.$$

As the length of phase one is defined as the size of I_1, the claim holds.

□

Theorem 2.2.14. *Algorithm 2.2.3 returns a maximum edge flow in time $O(|E|^2|V|)$.*

Proof. As Algorithm 2.2.3 is a specific implementation of the Ford-Fulkerson algorithm, we know that it returns an optimum edge flow. It remains to verify that this computation is done in $O(|E|^2|V|)$ many steps.

Observe that the algorithm has at most $|V| - 1$ phases, by Proposition 2.2.10. Moreover, each phase has at most $2|E|$ augmentations, by Proposition 2.2.13. It follows that the algorithm performs at most $2|E|(|V| - 1)$ augmentations.

We also know that each augmentation involves computing a shortest path from s to t. This computation can be specifically implemented using a breadth-first search, thus taking time at most $O(|E|)$.

Putting all these statements together, it is clear that the algorithm takes at most $O(|E|^2|V|)$ many steps. The result thus holds.

□

Recall that given a capacity function $c : E \to \mathbb{Q}_{\geq 0}$ on a graph $G = (V, E)$, we defined the capacity of a cut $(S, \overline{S})$, to be the sum of the capacities of its outgoing edges (from S into $\overline{S}$). Let us refer to the *minimum source-sink cut problem for directed graphs*, as the problem of finding a minimum capacity (s, t) cut for the graph $G = (V, E)$, with respect to the function $c : E \to \mathbb{Q}_{\geq 0}$. We conclude this section by stating the famous connection between minimum (s, t) cuts and maximum edge flows.

Theorem 2.2.15 (Max-flow Min-cut)**.** *Given a directed graph $G = (V, E)$ with capacity function $c : E \to \mathbb{Z}_{\geq 0}$, there exists a maximum edge flow $\boldsymbol{\zeta}$ and minimum (s, t) cut $(S, \overline{S})$, for which $val(\boldsymbol{\zeta}) = c(S, \overline{S})$.*

2.3 Applications of the Edmonds-Karp Algorithm

In this section we consider some applications of the Edmonds-Karp Algorithm to a number of important problems in graph theory. Unlike the previous sections, we primarily focus on problems pertaining to *undirected* graphs. In particular, we first investigate the minimum cut problem for undirected graphs. After this, we consider the problem of routing edge disjoint paths between pairs of vertices in undirected graphs. We remark that many of the proofs of the claims in this section are easily verified, and thus have been omitted.

Let us suppose that $G = (V, E)$ is an *undirected graph* with a *cost function* $c : E \to \mathbb{Q}_{\geq 0}$, and a source-sink pair (s, t) of vertices. If $S \subseteq V$, for which $(S, \overline{S})$ partitions s and t, then we refer to $(S, \overline{S})$ as an (s, t) *cut*. Moreover, we refer to the *cost* of $(S, \overline{S})$, denoted $c(S, \overline{S})$, as the total cost of the edges between S and $\overline{S}$. Rather,

$$c(S, \overline{S}) := \sum_{e \in \delta(S)} c(e),$$

where $\delta(S)$ contains the edges of G with exactly one end in S. We consider the *minimum cost problem*, as stated below:

Problem 2.3.1 (Source-Sink Minimum Cut for Undirected Graphs)**.** *Let $G = (V, E)$ be an undirected graph with cost function $c : E \to \mathbb{Q}_{\geq 0}$, and source-sink pair (s, t). The objective of this problem is to find an (s, t) cut $(S, \overline{S})$ of minimum cost.*

We remark that the decision to refer to $c : E \to \mathbb{Q}_{\geq 0}$ as a cost function, opposed to a capacity function, is purely syntactic. We do not explicitly compute edge flows on *undirected* graphs, so the cost terminology better describes the goal

of this problem. Let us now describe an algorithm which solves this problem in polynomial time. We make extensive use of the results from the previous section.

We first introduce a *directed* graph, denoted $G^* = (V^*, E^*)$, known as the *symmetrization* of G. This is constructed by replacing each edge e of G with directed edges pointed in both possible directions of e. Formally, we describe this process below:

- G^* has the same vertices as G.
- For each edge $e = \{u, v\} \in E$, add *directed* edges (u, v) and (v, u) to G^*.

Additionally, we define a capacity function $c^* : E^* \to \mathbb{Q}_{\geq 0}$ for G^*, where $c^*((u, v))$ takes on the value $c(\{u, v\})$ for each edge $(u, v) \in E^*$. We refer to c^* as the *symmetrization* of cost function c. Observe the following lemma:

Lemma 2.3.1. *Let $G = (V, E)$ be an undirected graph with cost function $c : E \to \mathbb{Q}_{\geq 0}$ and source sink pair (s, t). More, let $G^* = (V^*, E^*)$ and $c^* : E^* \to \mathbb{Q}_{\geq 0}$ be the symmetrizations of G and c respectively. If $(S, \overline{S})$ is an (s, t) cut of G, then it is also an (s, t) cut of G^* (and vice versa). Moreover,*

$$c(S, \overline{S}) = c^*(S, \overline{S}).$$

In particular, if $(S, \overline{S})$ is a minimum capacity cut for G^*, then it is a minimum cost cut for G^*.

The algorithm for Problem 2.3.1 first involves building G^* and c^* from G and c. It then computes an edge flow $\boldsymbol{\zeta}$ for G^*, of maximum value, using Algorithm 2.2.3 from the previous section. From here, the edge flow is used to build minimum

capacity (s,t) cut for G^*, which is then translated into a minimum cost (s,t) for G.

Before we describe the algorithm in detail, we recall some notation from the previous sections. Given an edge flow $\boldsymbol{\zeta}$ through G^*, we refer to $G^*_{\boldsymbol{\zeta}}$ as the *residual multigraph* of G^* with respect to $\boldsymbol{\zeta}$. Similarly, we refer to $c^*_{\boldsymbol{\zeta}}$ as the *residual capacity function* with respect to $\boldsymbol{\zeta}$. If γ is a path contained in $G^*_{\boldsymbol{\zeta}}$, then we know it has *nonzero residual capacity*; rather, all its edges are nonzero with respect to $c^*_{\boldsymbol{\zeta}}$.

Algorithm 2.3.1 Source-Sink Minimum Cut Algorithm

Let (G,E) be an undirected graph.
Let $c : E \to \mathbb{Q}_{\geq 0}$, and (s,t) a pair of distinct vertices.
1: Initialize $S \leftarrow \{s\}$
2: Compute G^* and c^*, the symmetrizations of G and c.
3: Compute an optimum edge flow $\boldsymbol{\zeta} = (\zeta_e)_{e \in E^*}$, using Algorithm 2.2.3.
4: **for** each $v \in V^*$ **do**
5: **if** there is a path p from s to v in $G^*_{\boldsymbol{\zeta}}$ **then**
6: $S \leftarrow S \cup \{v\}$.
7: Return $(S, \overline{S})$.

Theorem 2.3.2 (Source-Sink Minimum Cut)**.** *Algorithm 2.3.1 returns a minimum (s,t) cut in polynomial time.*

Proof. It is clear that the algorithm will run in polynomial time, as Theorem 2.2.14 proves that the Edmonds-Karp algorithm executes in polynomial time. Moreover, the for loop of the above algorithm can be implemented using a depth-first search on the graph $G^*_{\boldsymbol{\zeta}}$, where the vertices that are found are added to S.

Let us now consider the correctness of the algorithm. We first observe that in the proof of Theorem 2.2.4, the main goal was to show that $(S, \overline{S})$ has capacity equal to the value of $\boldsymbol{\zeta}$. Rather,

$$c^*(S, \overline{S}) = val(\boldsymbol{\zeta}).$$

As a consequence of the max-flow min-cut theorem (Theorem 2.2.15), this implies that $(S, \overline{S})$ is a minimum capacity cut. Moreover, Lemma 2.3.1 implies that $(S, \overline{S})$ is a minimum cost cut for G.

□

We now consider another problem in graph theory. Suppose $G = (V, E)$ is an undirected graph with source-sink pair (s, t). Moreover, suppose we wish to find paths between s and t which are edge disjoint. It is natural to wonder how many such paths exist in G. We summarize this problem below:

Problem 2.3.2 (Path Routing in Undirected Graphs)**.** *Let $G = (V, E)$ be an undirected graph with source-sink pair (s, t). If $l \geq 1$, then we wish to find a selection of simple paths from s to t, denoted $p_1, \ldots, p_l$, which are edge disjoint. Rather, for all $1 \leq i < j \leq l$, $e(p_i) \cap e(p_j) = \emptyset$. The goal of the problem is to find the largest selection of such paths.*

We wish to find an algorithm for this problem that runs in polynomial time. Given $G = (V, E)$, it will once again be useful to consider the symmetrization of G, denoted G^*. We can also associate a cost function $c : E \to \mathbb{Q}_{\geq 0}$ to G, where c is identically one. As before, we denote c^* as the symmetrization of c. Let us now outline the steps of this algorithm:

At a high level, we first compute a maximum edge flow through G^*. We then use the conversion algorithm from Section 2.1 (Algorithm 2.1.1), to build a maximum path flow through G^*. With this path flow, we can derive a selection of paths in G that are guaranteed to be edge disjoint. The optimality of the path flow will then ensure that the selection of paths in G is as large as possible.

Before we implement this algorithm in detail, it will be convenient to focus on flows through G^* that satisfy additional constraints than we've previously seen. Let us suppose that $\boldsymbol{\zeta}$ is an edge flow through G^*, in which for each each $\{u, v\} \in E$, at most one of $\zeta_{(u,v)}$ and $\zeta_{(v,u)}$ is nonzero. We refer to an edge flow $\boldsymbol{\zeta}$ with this property as being *canonical*. We also classify path flows through G. Let $\boldsymbol{f}$ be a path flow through G^*, such that for each edge $\{u, v\} \in E$, at most of one (u, v) or (v, u) is contained in a nonzero path of $\boldsymbol{f}$. In this case, we refer to $\boldsymbol{f}$ as a canonical path flow.

If we once again consider the conversion algorithm from Section 2.1, namely Algorithm 2.1.1, then it is clear that this algorithm maps canonical edge flows to canonical path flows.

Lemma 2.3.3. *Let $G = (V, E)$ be an undirected graph with an identically one cost function $c : E \to \mathbb{Q}_{\geq 0}$, and a source-sink pair (s, t). More, suppose G^* and c^* are the symmetrizations of G. If $\boldsymbol{\zeta}$ is a canonical edge flow through G^*, then the path flow $\boldsymbol{f}$ returned from Algorithm 2.1.1 is also canonical.*

We also observe the following lemma:

Lemma 2.3.4. *Given any edge flow $\boldsymbol{\zeta}$ through G^*, there exists a canonical edge flow $\boldsymbol{\zeta}'$ through G^*, such that $val(\boldsymbol{\zeta}) = val(\boldsymbol{\zeta}')$. Moreover, given $\boldsymbol{\zeta}$ as input, $\boldsymbol{\zeta}'$ may be computed from it in polynomial time.*

Let us now state a detailed description of how to build an optimum selection of paths from s to t. We denote Γ as the set of *directed* simple paths from s to t in G^*.

Algorithm 2.3.2 Path Routing Algorithm

Let (G, E) be an undirected graph.
Let $c : E \to \mathbb{Q}_{\geq 0}$, and (s, t) a pair of distinct vertices.
1: Initialize $\mathcal{P} \leftarrow \emptyset$.
2: Compute G^* and c^*, the symmetrizations of G and c.
3: Compute an optimum edge flow $\boldsymbol{\zeta} = (\zeta_e)_{e \in E^*}$, using Algorithm 2.2.3.
4: Update $\boldsymbol{\zeta}$ such that it is canonical.
5: Run Algorithm 2.1.1 with the above inputs.
6: Let $\boldsymbol{f}$ be the path flow returned from this subroutine.
7: **for** each $\gamma \in \Gamma$ **do**
8: **if** $f_\gamma \neq 0$ **then**
9: Convert γ to a undirected path p.
10: Add p to $\mathcal{P}$.
Return $\mathcal{P}$.

In the above algorithm, the optimality of $\boldsymbol{f}$ ensures that the collection of edges $\mathcal{P}$ will itself be optimum. This leaves us with the following theorem:

Theorem 2.3.5. *Algorithm 2.3.2 returns a selection of edge disjoint paths, denoted $\mathcal{P}$, which run from s to t in G. Moreover, the size of $\mathcal{P}$ is optimum, and is returned in polynomial time.*

CHAPTER 3
The Multiway Cut Problem

This chapter focuses on a problem known as the multiway cut problem, which is a generalization of the minimum cut problem on undirected graphs. The first section begins with a detailed introduction to the problem, culminating in the design of a combinatorial algorithm for finding multiway cuts. This approximation algorithm is based on a greedy approach, where it routinely makes use of the minimum cut algorithm as a subroutine. The algorithm is seen to attain a performance guarantee of $2 - 2/k$, where $k \geq 1$ is the number of source-sink pairs that are separted from the input graph $G = (V, E)$. The analysis of this algorithm is based on the material from the book "Approximation Algorithms" [Vv11].

In the following section, an IP formulation of the problem is presented, and relaxed to an LP formulation. This linear program has exponentially many constraints, and so a polynomial time separation oracle is designed to help solve the program. Using an optimum solution to this program, a randomized rounding algorithm is used to derive a (integral) solution to the IP formulation of the problem. This algorithm is proven to attain an expected approximation guarantee of $2 - 2/k$, using a proof technique typically used in rounding algorithms. A derandomization algorithm is then derived, and seen to attain an identical approximation guarantee deterministically. The section ends by matching the upper and lower bounds of the integrality gap of this integer program. While the

randomized algorithm of this section is referenced in the book "Approximation Algorithms" [Vv11], the derandomization of this algorithm is original, as is the analysis of both algorithms.

The final section of the chapter introduces a second IP formulation of the multiway cut problem, motivated by the shortcomings of the previous formulation. An LP relaxation of this integer program is taken, whose optimum solution is then used to design another randomized rounding algorithm. This algorithm is then seen to achieve an expected approximation guarantee of 3/2. Once again, a derandomization process is outlined, and argued to deterministically achieve an approximation guarantee of 3/2. The majority of the material from this section is adapted from the book "The Design of Approximation Algorithms" [WS11].

3.1 Introduction to Multiway Cut

Suppose we are given an undirected graph $G = (V, E)$ with a *cost* function $c : E \to \mathbb{Q}_{\geq 0}$. If we are also given a source-sink pair (s, t) of distinct vertices, then we may consider a *cut* $F \subseteq E$, whose removal leaves s disconnected from t in G. The *cost* of the cut F, denoted $c(F)$, is defined to be the sum of the costs of its edges. The problem of finding a cut of minimal cost is known as the *minimum cut problem*. We saw in the previous chapter that there exists polynomial time algorithms for computing such cuts (see Theorem 2.3.2 Section 2.3).

In this chapter, we will primarily be interested in a generalization of this problem. Instead of a single source sink pair, we are given k distinct source nodes $s_1, \ldots, s_k$, and asked to find a subset of edges $F \subseteq E$, whose removal disconnects the source nodes from each other. As before, we are interested in finding cuts that are of optimum cost. Let us formally state this problem below:

Problem 3.1.1 (Multiway Cut)**.** *Suppose we are given an undirected graph $G = (V, E)$, with cost function $c : E \to \mathbb{Q}_{\geq 0}$. Moreover, let $s_1, \ldots, s_k$ be $k \geq 2$ distinct vertices of G, which we refer to as source nodes of G. We wish to find a subset $F \subseteq E$, for which $s_1, \ldots, s_k$ are disconnected in $G \setminus F$. If F satisfies this condition, we refer to it as a multiway cut. We denote the cost of F as $c(F)$, where $c(F) := \sum_{e \in F} c(e)$. The goal of the problem is to find a multiway cut of minimum cost.*

Unlike the minimum cut problem, it is known that for fixed $k \geq 3$, an optimum cut with this property is **NP**-hard to find. As this is the case, we provide a number of algorithms throughout this chapter which efficiently approximate this

problem. We begin with a simple combinatorial algorithm, which is based on a greedy strategy. In the later sections, we shall see more sophisticated algorithms, which employ the use of linear programming techniques.

We now outline our greedy combinatorial algorithm. At a high level, this algorithm sequentially processes the source nodes of the graph in an arbitrary order. When it reaches source node s_i, it computes a cut $F_i \subseteq E$, which optimally separates s_i from the other sources. We shall see how this computation is done, but first we formalize this algorithm below:

Algorithm 3.1.1 Greedy Multiway Cut Algorithm

Let $G = (V, E)$ be an undirected graph.
Let $c : E \to \mathbb{Q}_{\geq 0}$, and $\{s_1, \ldots, s_k\} \subseteq V$.
1: Initialize $F \leftarrow \emptyset$.
2: **for** i=1 to k **do**
3: Compute a minimum cost cut F_i, which separtes s_i from the other source nodes.
4: Let $F \leftarrow F \cup F_i$
5: Compute the cut of greatest cost among the candidates, $F_1, \ldots, F_k$.
6: Denote F_j as this cut.
7: Return $F \setminus F_j$

In order to perform Step 3 of the above computation, we must first introduce a graph augmentation. Let us suppose s_1 is the selected vertex, and we wish to optimally separate it from $s_2, \ldots, s_k$; the other cases can be done analogously. We first define a new graph $G_1 = (V_1, E_1)$ with cost function $c_1 : E_1 \to \mathbb{Q}_{\geq 0}$:

- For each vertex $v \in V$, if $v \neq s_2, \ldots, s_k$, then add v to V_1.
- Add an *augmented vertex* with label ν_a to V_1.

- For each edge $e \in E$, if neither end of e is in $\{s_2, \ldots, s_k\}$, then add e to E_1 with cost $c_1(e) = c(e)$.
- If $e = \{s_i, v\}$, where $2 \leq i \leq k$, and $v \notin \{s_2, \ldots, s_k\}$, then add edge $\{\nu_a, v\}$ to G_1 with cost equal to $c(e)$.
- If e has both ends in $\{s_2, \ldots, s_k\}$, then we make no changes to G_1.

We refer to $G_1 = (V_1, E_1)$ as an *augmented graph*, with *augmented cost function* $c_1 : E_1 \to \mathbb{Q}_{\geq 0}$. Its use can be summarized in the following proposition:

Proposition 3.1.1. *Let G_1 and $c_1 : E_1 \to \mathbb{Q}_{\geq 0}$ be as above. Moreover, suppose F_1 is a minimum cost cut of s_1 and ν_a. Given F_1 as input, we may recover a minimum cost cut of s_1 and $s_2, \ldots, s_k$ in polynomial time.*

We may use Algorithm 2.3.1 to build a minimum cost cut of s_1 and the augmented vertex v_a. From here, we can use the above proposition to recover a minimum cost cut of s_1 and $s_2, \ldots, s_k$ in polynomial time.

Having justified the time complexity of Step 3, it is clear that the algorithm executes in polynomial time. Moreover, F clearly separates the sources, so the algorithm is correct. Let us now state the approximation guarantee of this algorithm.

Theorem 3.1.2. *Algorithm 3.1.1 achieves an approximation guarantee of 2- 2/k.*

Proof. Let us assume that $B \subseteq E$ is a minimum cost multiway cut of G. As B is a feasible solution to the multiway cut problem, we know that $G \setminus B$ is a graph which places $s_1, \ldots, s_k$ into separate components. Moreover, as the solution is minimum, there exists exactly k components of $G \setminus B$.

Let us denote these components by $C_1, \ldots, C_k$, where $s_i \in C_i$ for $i = 1, ..., k$. More, for $i = 1, .., k$, set $B_i := e(C_i, V \setminus C_i)$, where $e(C_i, V \setminus C_i)$ is the edges of G with exactly one end in C_i.

Observe that each edge of B occurs in exactly two of the B_i sets. Thus we have that $B = \cup_{i=1}^{k} B_i$, and $c(B) = \sum_{i=1}^{k} 2\, c(B_i)$.

It is also clear that each B_i separates the source s_i from the remaining source nodes. Thus, as the cuts in Algorithm 3.1.1, optimally perform this function, we know that $c(F_i) \leq c(B_i)$ for $i = 1, .., k$.

Let us assume without loss of generality that F_k is the most costly of the cuts $F_1, \ldots, F_k$. We observe then that the cost of F_k is at least $\sum_{i=1}^{k} \frac{c(F_i)}{k}$, the average cost of the cuts.

We know that since $F \subseteq \cup_{i=1}^{k-1} F_i$,

$$c(F) \leq \sum_{i=1}^{k} c(F_i) - c(F_k) \leq (1 - \frac{1}{k}) \sum_{i=1}^{k} c(F_i)$$

Now,

$$(1 - \frac{1}{k}) \sum_{i=1}^{k} c(F_i) \leq (1 - \frac{1}{k}) \sum_{i=1}^{k} c(B_i) \leq (1 - \frac{1}{k})\, 2\, c(B)$$

Thus,

$$c(F) \leq (1 - \frac{1}{k}) 2\, c(B)$$

It follows that the algorithm returns a multiway cut with cost at most $(2 - \frac{2}{k})OPT$.

□

3.2 Randomized Region Cut

In this section, we explore the use of linear programming techniques in the context of the multiway cut problem. The algorithm we intoduce in this section uses a linear programming paradigm known as randomized rounding. While the algorithm does not provide a better approximation guarantee than we saw in the previous section, it allows us to introduce techniques that are widely used throughout the text. In particular, we shall later see how to use the same techniques to attain significant improvements over the approximation guarantee of the greedy algorithm from the previous section.

We begin by introducing the necessary framework to structure the multiway cut problem as a integer program. For $i = 1, \ldots, k$, let $\mathcal{P}_i$ be the set of all simple paths from s_i to any source vertex. Moreover, we denote $\mathcal{P}$ as the union of $\mathcal{P}_1, \ldots, \mathcal{P}_k$. If we introduce a variable d_e for each edge $e \in E$, then we have the following integer programming formation of the multiway cut problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c(e) d_e \\ \text{subject to} \quad & \sum_{e \in p_i} d_e \geq 1 \qquad \forall i = 1, \ldots, k, \quad p_i \in \mathcal{P}_i \\ & d_e \in \{0, 1\} \quad \forall e \in E \end{aligned} \tag{3.2.1}$$

The constraints of the integer program ensure that if $\boldsymbol{d} = (d_e)_{e \in E}$ is a valid solution, then each path between source nodes will have at least one edge selected in $\boldsymbol{d}$. Rather, if $F \subseteq E$ is such that $e \in F$ if and only if $d_e = 1$, then F will form a valid multiway cut.

We now consider a relaxation of this integer program. For each $e \in E$, we allow d_e to take values in the unit interval $[0, 1]$. This leaves us with the following linear program:

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c(e) d_e \\ \text{subject to} \quad & \sum_{e \in P_i} d_e \geq 1 \quad \forall i = 1, \ldots, k, \quad p_i \in \mathcal{P}_i \\ & 0 \leq d_e \leq 1 \quad \forall e \in E \end{aligned} \tag{3.2.2}$$

Before we analyze this relaxation, it will be convenient to simplify the constraints of the program. Observe that the cost function $c : E \to \mathbb{Q}_{\geq 0}$ is nonnegative by definition. Combined with the fact that we are interested in optimum solutions to LP (3.2.2), we may forgo the upper bounds placed on the d_e variables. This leaves us with the following linear program in standard form:

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c(e) d_e \\ \text{subject to} \quad & \sum_{e \in P_i} d_e \geq 1 \quad \forall i = 1, \ldots, k, \quad p_i \in \mathcal{P}_i \\ & d_e \geq 0 \quad \forall e \in E \end{aligned} \tag{3.2.3}$$

We now focus our attention on this simplified linear program. From the outset, it is clear that it may have an exponential number of constraints. Rather, if the graph G has a source s_i for which the size of $\mathcal{P}_i$ is exponential in the size of the graph, then our linear program will be exponential in size. As this is the case, it seems that we cannot hope to solve the program in polynomial time via standard LP solvers.

Fortunately, there exists an approach that allows us to circumvent this issue. Firstly, we recognize that our linear program has a *polynomial time separation oracle* (see Subsection 1.3.1 for a general definition). Rather, given a potential solution $\boldsymbol{d}$ to LP (3.2.3), we can compute in polynomial time whether there exists an inequality of LP (3.2.3) which $\boldsymbol{d}$ violates. Moreover, if such an inequality exists, then the oracle will provide an inequality which is violated. If no inequality is violated, then the oracle will indicate that the solution is valid.

Before we implement this polynomial time separation oracle, we introduce a means of measuring distance in the graph G. Let $\boldsymbol{d} = (d_e)_{e \in E}$ be an arbitrary edge assignment, where $d_e \in \mathbb{Q}_{\geq 0}$ for all $e \in E$ ($\boldsymbol{d}$ may or may not be a valid solution to LP (3.2.3)). If p is a path in G, then we refer to the *length* of p *with respect to* $\boldsymbol{d}$, as the quantity, $\sum_{e \in p} d_e$. Moreover, if u and v are two vertices in G, then the *distance* between u and v *with respect to* $\boldsymbol{d}$, is the length of the shortest path between them. We denote this quantity $dist_{G,\boldsymbol{d}}(u, v)$, and simply $dist_{\boldsymbol{d}}(u, v)$ when the context is clear. Computing a shortest path between any two vertices (with respect to $\boldsymbol{d}$) can be done in polynomial time using Dijkstra's algorithm for instance. We now introduce the polynomial time separation oracle:

Algorithm 3.2.1 Separation Oracle for LP (3.2.3)

Let $G = (V, E)$ be an undirected graph.
Let $c : E \to \mathbb{Q}_{\geq 0}$, and $\{s_1, ..., s_k\} \subseteq V$.
Let $\boldsymbol{d} = (d_e)_{e \in E}$ be a potential solution to LP (3.2.3).

```
for i=1, ...,k do
    for j=i+1, ...,k do
        Compute a shortest path p*_{i,j} between s_i and s_j.
        if Σ_{e∈p*_{i,j}} d_e < 1 then
            The solution d is marked as invalid.
            The inequality involving path p*_{i,j} is returned.
The solution d is marked as valid.
```

Proposition 3.2.1. *Algorithm 3.2.1 is a polynomial time separation oracle which can be used to solve LP* (3.2.3) *in polynomial time.*

Proof. We first observe that Algorithm 3.2.1 executes in polynomial time. It remains to show that it satisfies the properties of a separation oracle.

Let us consider the edge assignment $\boldsymbol{d}$, which is passed to Algorithm 3.2.1. We first assume that the solution is infeasible. Rather, there exists $1 \leq i' < j' \leq k$, such that p' lies between $s_{i'}$ and $s_{j'}$, yet p' has length less than 1. That is,

$$\sum_{e \in p'} d_e < 1.$$

In this case, consider the step of the algorithm when it computes a shortest path $p^*_{i',j'}$ between $s_{i'}$ and $s_{j'}$. As $p^*_{i',j'}$ is a shortest path, its length cannot exceed the length of p'. In particular,

$$\sum_{e \in p^*_{i',j'}} d_e \leq \sum_{e \in p'} d_e < 1.$$

The algorithm will therefore mark $\boldsymbol{d}$ as infeasible, and return the inequality pertaining to $p^*_{i',j'}$ as violated.

It remains to prove that if the algorithm marks a solution as feasible, then it will satisfy LP (3.2.3). Let us suppose that $\boldsymbol{d}$ is marked as feasible. We observe that in this case, the algorithm will verify that for each $1 \leq i < j \leq k$,

$$1 \leq \sum_{e \in p^*_{i,j}} d_e,$$

where $p^*_{i,j}$ is the shortest path from s_i to s_j that the algorithm computes. On the other hand, if $p \in \mathcal{P}$ is a simple path from s_i to s_j, then we know that,

$$1 \leq \sum_{e \in p^*_{i,j}} d_e \leq \sum_{e \in p} d_e,$$

as $p^*_{i,j}$ is a shortest path from s_i to s_j. It follows that $\boldsymbol{d}$ does in fact satisfy all the inequalities of LP (3.2.3), so the algorithm made the correct assignment. We have therefore shown that Algorithm 3.2.1 is in fact a polynomial time separation oracle.

Using this algorithm as a subroutine, we may appeal to Theorem 1.3.5 from Subsection 1.3.1 to provide us with a polynomial time algorithm for solving LP (3.2.3).

□

In light of this proposition, let us suppose that we compute an optimum solution to LP (3.2.3), which we label $\boldsymbol{d}$. Solutions of this form have a number of desirable properties. In particular, we observe that $\boldsymbol{d}$ satisfies the *triangle inequality* for any selction of edges. Rather, if u, v and w are vertices of G, then we

have that,

$$d_{u,v} \leq d_{u,w} + d_{w,v},$$

where $\{u, v\}, \{u, w\}$ and $\{w, v\}$ are edges of G. We now prove this property.

Proposition 3.2.2. *If* $\boldsymbol{d}$ *is an optimum solution to LP* (3.2.3)*, then it satisfies the triangle inquality for any selection of edges of* G*.*

Proof. We first observe that $d_e \geq 0$ for each edge $e \in E$, as $\boldsymbol{d}$ is a solution to LP (3.2.3).

It remains to check that $\boldsymbol{d}$ satisfies the triangle inequality on E. Suppose that this is not the case. That is, there exist vertices u, v and w with edges between them, such that,

$$d_{u,v} > d_{u,w} + d_{w,v} \tag{3.2.4}$$

Moreover, observe that since $\boldsymbol{d}$ is an optimum solution to LP (3.2.3), $d_{u,v}$ cannot be decreased without violating a constraint of the linear program.

Thus, there exists a pair of sources, say s_1, s_2, with a simple path p between them, such that,

$$\sum_{e \in p} d_e = 1,$$

where the edge $\{u, v\}$ is guaranteed to lie in p.

Observe that if we remove $\{u, v\}$ from p, and replace it with $\{u, w\}$ followed by $\{w, v\}$, then we are left with a new path p' between s_1 and s_2. We also build a path p'', whose edges are contained in p', and which is guaranteed to be simple.

As consequence of Equation 3.2.4, we know that the length of p' is strictly less than the length of p. Moreover, as the edges of p'' are contained in p', we know that the length of p'' is also less than the length of p.

However, since p has length one, p'' must have length less than one. As p'' is a simple path between source nodes s_1 and s_2, this means that $\boldsymbol{d}$ is not a valid solution to 3.2.3. This is a contradiction, so the violation in Equation 3.2.4 cannot occur. It follows that $\boldsymbol{d}$ satisfies the triangle inquality everywhere, and so the claim holds.

□

Let us once again consider the distance function $dist_{\boldsymbol{d}}(\cdot, \cdot)$, when $\boldsymbol{d}$ is a solution to LP (3.2.3). It is clear that $dist_{\boldsymbol{d}}(\cdot, \cdot)$ forms a *metric* on the vertices of G. On the other hand, if we also assume that $\boldsymbol{d}$ is an *optimum* solution to LP (3.2.3), then we can think of $dist_{\boldsymbol{d}}(\cdot, \cdot)$ as extending $\boldsymbol{d}$ from edges to paths.

Proposition 3.2.3. *Let $\boldsymbol{d}$ be a solution to LP* (3.2.3)*. The function $dist_{\boldsymbol{d}} : V^2 \to \mathbb{Q}_{\geq 0}$, forms a metric on the vertices of G. Moreover, if $\boldsymbol{d}$ is an optimum solution to LP* (3.2.3)*, then*

$$d_{u,v} = dist_{\boldsymbol{d}}(u, v),$$

for each each $\{u, v\} \in E$.

We now return to the original problem of desiging an approximation algorithm for finding multiway cuts. Let us once again assume that $\boldsymbol{d}$ is an optimum solution to LP (3.2.3). As observed in the above proposition, we can use $\boldsymbol{d}$ to form a metric $dist_{\boldsymbol{d}}(\cdot, \cdot)$ on the vertices on our graph. Once we have this metric, we can pick a

radius $0 \leq r < \frac{1}{2}$, and grow regions of the given radius about each source node, $s_1, \ldots, s_k$.

Namely, for $i = 1, \ldots, k$, let $B_r^{\boldsymbol{d}}(s_i)$ be the vertices of G with distance from s_i less than or equal to r (with respect to $dist_{\boldsymbol{d}}(\cdot, \cdot)$). We denote these sets by $B_r(s_i)$ when the context is clear. Observe that since $r < \frac{1}{2}$, we know that the regions will be disjoint. This is easily seen, for if $1 \leq i < j \leq k$, then suppose $B_r(s_i)$ and $B_r(s_j)$ share a vertex $v \in V$. In this case, $dist_{\boldsymbol{d}}(s_i, v) < \frac{1}{2}$ and $dist_{\boldsymbol{d}}(s_j, v) < \frac{1}{2}$, so by the triangle inequality,

$$dist_{\boldsymbol{d}}(s_i, s_j) < 1.$$

As $\boldsymbol{d}$ was assumed to be a valid solution to 3.2.3, this is a contradiction. It follows that $B_r(s_i)$ and $B_r(s_j)$ are disjoint for all $1 \leq i < j \leq k$. That being said, there may exist vertices that lie outside of the regions $B_r(s_1), \ldots, B_r(s_k)$. In other words, $V \setminus \cup_{i=1}^{k} B_r(s_i)$ may be nonempty.

Once we form these disjoint regions, we collect the edges which lie between them. We begin by selecting the edge set $F_1 := \delta(B_r(s_1))$; rather, the edges of G with exactly one end in $B_r(s_1)$. We then iterate this process, where edges $F_i := \delta(B_r(s_i)) \setminus \cup_{j=1}^{i-1} F_j$ are selected for $2 \leq i \leq k$. After this process, the most costly of these edges set, say $F_{i'}$, is computed. The edge set F is then returned, where $F := \cup_{i=1} F_i \setminus F_{i'}$.

Algorithm 3.2.2 Region Multiway Cut Algorithm

Let $G = (V, E)$ be an undirected graph.
Suppose $c : E \to \mathbb{Q}_{\geq 0}$, and $\{s_1, ..., s_k\} \subseteq V$.
Let $0 \leq r < \frac{1}{2}$.
1: Initialize $F, F_1, \ldots, F_k \leftarrow \emptyset$.
2: Find an optimum solution $\boldsymbol{d}$ of LP (3.2.3).
3: **for** $i = 1, \ldots, k$ **do**
4: Set $F_i \leftarrow \delta(B_r(s_i)) \setminus F$.
5: Let $F \leftarrow F \cup F_i$.
6: Compute $1 \leq i' \leq k$, such that $F_{i'}$ is the most costly of $F_1, \ldots, F_k$.
7: Set $F \leftarrow F \setminus F_{i'}$.
8: Return F

We observe that since we can solve LP (3.2.3) in polynomial time, the algorithm itself will run in polynomial time. Moreover, the algorithm will always return a multiway cut.

Proposition 3.2.4. *For any value of* $0 \leq r < \frac{1}{2}$*, Algorithm 3.2.2 will run return a valid multiway cut.*

Proof. Let consider the edge set F returned by the algorithm. We may assume that if $F_{i'}$ is the most costly of $F_1, \ldots, F_k$, then $i' = k$. Thus, $F = \cup_{i=1}^{k-1} F_i$.

We now consider the edge set F^*, defined to be the union of

$$\delta(B_r(s_1)), \ldots, \delta(B_r(s_{k-1})).$$

It is clear from construction that since $F_k \subseteq \delta(B_r(s_k))$, we know that,

$$F^* \subseteq F,$$

as $F = \cup_{i=1}^{k} \delta(B_r(s_i)) \setminus F_k$. Thus, if we can show that F^* is a multiway cut, then F will also be one.

Observe that for each $i = 1, \ldots, k$, the region $B_r(s_i)$ is disjoint from the other regions (see the discussion above Algorithm 3.2.2). In particular, s_i is the only source node present in $B_r(s_i)$. This means that the edge set $\delta(B_r(s_i))$ will *separate* s_i from the other source nodes. Rather, each path $p_i \in \mathcal{P}$ will share an edge with $\delta(B_r(s_i))$. Thus, each path in $\mathcal{P}$ must share an edge with one of $\delta(B_r(s_1)), \ldots, \delta(B_r(s_{k-1}))$, as any path in $\mathcal{P}_k$ must also be in one of $\mathcal{P}_1, \ldots, \mathcal{P}_{k-1}$. It follows that F^* is a multiway cut, thus completing the proof.

□

Now that we have an algorithm which is guaranteed to return a valid multiway cut, we would like to alter it in such a way that this multiway cut has cost close to optimum. A natural approach to this problem is to optimize the radius chosen based on the inputs given. While this will indeed be possible, it turns out that it is easier to analyze a randomized version of Algorithm 3.2.2, in which the radius is chosen uniformly at random from the interval $[0, \frac{1}{2})$.

More specifically, we first fix the inputs G, c and $s_1, \ldots, s_k$. We then sample a number ρ uniformly at random from $[0, \frac{1}{2})$. Using the source nodes together with G and c, we run Algorithm 3.2.2 with ρ as the radius. The result will be a random edge set F, which is guaranteed to be a multiway cut. Let us formally outline this procedure:

Algorithm 3.2.3 Randomized Multiway Cut Algorithm

Let $G = (V, E)$ be an undirected graph.
Suppose $c : E \to \mathbb{Q}_{\geq 0}$, and $\{s_1, ..., s_k\} \subseteq V$.
1: Sample $0 \leq \rho < 1/2$, uniformly at random
2: Run Algorithm 3.2.2 with the above inputs.
3: Let F be the output of this subroutine.
4: Return F.

We remark that we can model the randomness present in the above algorithm by considering an abstract probability space, $(\Omega, \mathcal{B}, \mathbb{P})$, for which ρ is a random variable; rather, $\rho : \Omega \to \mathbb{R}$. In this sense, the law of ρ, denoted $\mathcal{L}(\rho)$, is distributed as a normalized Lebesgue measure on $[0, \frac{1}{2})$. This means that for each interval $[a, b] \subseteq [0, \frac{1}{2})$, we have that,

$$\mathbb{P}(\rho \in [a, b]) = 2(b - a).$$

We may also view F as a random element from Ω to the power set of E (the collection of all edge subsets of E). When we sample $\omega \in \Omega$, $F(\omega)$ can be thought of as the edges selected by the algorithm, when $\rho(\omega)$ is passed as the radius.

Theorem 3.2.5. *Algorithm 3.2.3 has an expected approximation guarantee of* $2 - \frac{2}{k}$.

In order to prove this theorem, we first prove a useful lemma. Let us suppose $B_\rho(s_1), \ldots, B_\rho(s_k)$ are the random regions formed in Algorithm 3.2.3. Rather, if $1 \leq i \leq k$, then we have that $B_\rho(s_i)(\omega) := B_{\rho(\omega)}(s_i)$ for $\omega \in \Omega$.

Lemma 3.2.6. *We observe that if* $\{u, v\}$ *is an edge of* G*, then*

$$\mathbb{P}(\{u, v\} \in \cup_{i=1}^{k} \delta(B_\rho(s_i))) \leq 2\, d_{u,v}.$$

Proof. Consider the edge $e = \{u, v\}$ of G. Given a source node s_j for $1 \leq j \leq k$, we say that s_j *separates* e, provided exactly one end of e is in $B_\rho(s_j)$. As ρ is generated randomly, we denote S_j as the event in which this occurs. It is clear that

$$\mathbb{P}(e \in \cup_{i=1}^{k} \delta(B_\rho(s_i))) = \mathbb{P}(\cup_{i=1}^{k} S_i)$$

by definition. On the other hand, let us now consider when the event S_j occurs. We may first assume that s_j is closer to u than v; rather, $dist_{\boldsymbol{d}}(s_j, u) \leq dist_{\boldsymbol{d}}(s_j, v)$. Observe that S_j occurs if and only if ρ falls in the interval,

$$[dist_{\boldsymbol{d}}(s_j, u), dist_{\boldsymbol{d}}(s_j, v)).$$

In order for this to happen, it is necessary that $dist_{\boldsymbol{d}}(s_j, u) < 1/2$, as $\rho < 1/2$. We refer to the source node s_j as having the *potential to separate e through u*, provided it has this property. Notice that this is an entirely deterministic classification; it depends only on the edge assignment $\boldsymbol{d}$ and the structure of G.

Similarly, if s_j is closer to v than u, then S_j occurs if and only if

$$\rho \in [dist_{\boldsymbol{d}}(s_j, v), dist_{\boldsymbol{d}}(s_j, u)).$$

Moreover, it remains necessary for $dist_{\boldsymbol{d}}(s_j, v) < 1/2$. In this case, we refer to source node s_j as having the *potential to separate e through v*. Once again, this classification is deterministic.

We observe that there is at most one source node which has the potential to separate e through u. To see this, observe that were it not the case, then there would exist two source nodes with distance between them strictly less than one.

As $\boldsymbol{d}$ is a feasible solution to LP (3.2.3), this cannot occur. Similarly, there is at most one source node with the potential to separate e through v.

In light of this observation, we may assume that s_1 is the only source node with $dist_{\boldsymbol{d}}(s_1, u) < 1/2$, and s_2 is the only source node with $dist_{\boldsymbol{d}}(s_2, v) < 1/2$. We first recognize that,

$$1 \leq dist_{\boldsymbol{d}}(s_1, s_2) \tag{3.2.5}$$

$$= dist_{\boldsymbol{d}}(s_1, v) + dist_{\boldsymbol{d}}(v, s_2) \tag{3.2.6}$$

by Proposition 3.2.3 and the feasibility of $\boldsymbol{d}$. Let us now assume that $dist_{\boldsymbol{d}}(s_1, v) \leq 1/2$. As consequence of Equation 3.2.5, this means that,

$$dist_{\boldsymbol{d}}(s_2, v) \geq \frac{1}{2}.$$

However, we assumed that s_2 had potential to separate e through v, so this cannot occur. It follows that,

$$dist_{\boldsymbol{d}}(s_1, u) < \frac{1}{2} < dist_{\boldsymbol{d}}(s_1, v)$$

Using a symmetrical argument, we also arrive at the conclusion,

$$dist_{\boldsymbol{d}}(s_2, v) < \frac{1}{2} < dist_{\boldsymbol{d}}(s_2, u).$$

Let us now consider the probabilities of S_1 and S_2 occuring. As a result of the above equations,

$$\begin{aligned}\mathbb{P}(S_1) &= \mathbb{P}(\rho \in [dist_{\boldsymbol{d}}(s_1, u), \frac{1}{2})) \\ &= 2(\frac{1}{2} - dist_{\boldsymbol{d}}(s_1, u)),\end{aligned}$$

as ρ is drawn uniformly at random from the interval $[0, \frac{1}{2})$. Similarly,

$$\begin{aligned}\mathbb{P}(S_2) &= \mathbb{P}(\rho \in [dist_{\boldsymbol{d}}(s_2, v), \frac{1}{2})) \\ &= 2(\frac{1}{2} - dist_{\boldsymbol{d}}(s_2, v)).\end{aligned}$$

Finally,

$$\begin{aligned}\mathbb{P}(S_1 \cap S_2) &= \mathbb{P}(\rho \in [dist_{\boldsymbol{d}}(s_2, u), \frac{1}{2})) \\ &= 2(\frac{1}{2} - dist_{\boldsymbol{d}}(s_2, v),\end{aligned}$$

provided we assume that $dist_{\boldsymbol{d}}(s_1, u) \leq dist_{\boldsymbol{d}}(s_2, v)$. It follows that,

$$\begin{aligned}\mathbb{P}(S_1 \cup S_2) &= \mathbb{P}(S_1) + \mathbb{P}(S_2) - \mathbb{P}(S_1 \cap S_2) \\ &= 2\,(\frac{1}{2} - dist_{\boldsymbol{d}}(s_1, u)) \\ &\leq 2\,(dist_{\boldsymbol{d}}(s_1, v) - dist_{\boldsymbol{d}}(s_1, u)) \\ &= 2\,d_{u,v}.\end{aligned}$$

where the last line follows from the optimality of $\boldsymbol{d}$ and Proposition 3.2.3. As s_1 and s_2 were assumed to be the only source nodes with the potential to separate e, we have that

$$\begin{aligned}\mathbb{P}(\cup_{i=1}^{k} S_i) &= \mathbb{P}(S_1 \cup S_2) \\ &= 2\,d_{u,v}.\end{aligned}$$

The only remaining case to consider is when there exists exactly one source node with the potential to separate e. If we assume s_j is such a vertex, then it is

clear that,

$$\begin{aligned}\mathbb{P}(S_j) &\leq 2\,(dist_{\boldsymbol{d}}(s_j, v) - dist_{\boldsymbol{d}}(s_j, u)) \\ &= 2\,d_{u,v},\end{aligned}$$

provided s_j is closer to u than v. In any case, we arrive at the conclusion,

$$\begin{aligned}\mathbb{P}(\cup_{i=1}^{k} S_i) &= \mathbb{P}(S_j) \\ &\leq 2\,d_{u,v},\end{aligned}$$

as before. This concludes the proof of the lemma.

□

With this lemma, we are now able prove the statement of Theorem 3.2.5.

Proof. Ultimately, we would like to show that,

$$\mathbb{E}\,F \leq 2\,(1 - \frac{1}{k})\;OPT_f,$$

where OPT_f is the cost of an optimum solution to LP (3.2.3), and F is the random multicut returned by the algorithm. It will be useful to first examine the random edge set F^*, defined below:

For each $i = 1, \ldots, k$, we may consider the random edge subset, $F_i^* := \delta(B_\rho(s_i))$, together with the random variable, $c(F_i^*)$. Let us denote $F^* := \cup_{i=1}^{k} F_i^*$. Our first goal will be to show that,

$$\mathbb{E}\,c(F^*) \leq 2\,OPT_f.$$

For each edge $e \in E$, let us denote A_e as the event in when $e \in F^*$. Moreover, we denote $\mathbf{1}_{A_e}$ as the indicator variable for this event.

It is clear that,

$$c(F^*) = \sum_{e \in E} c(e)\, \mathbf{1}_{A_e}.$$

Moreover, by linearity of expectation,

$$\begin{aligned} \mathbb{E}\, c(F^*) &= \sum_{e \in E} c(e)\, \mathbb{P}(A_e) \\ &\leq \sum_{e \in E} 2\, c(e)\, d_e, \end{aligned}$$

by Lemma 3.2.6. On the other hand, we know that,

$$OPT_f = \sum_{e \in E} c(e)\, d_e,$$

as $\boldsymbol{d}$ was assumed to be an optimum solution to LP (3.2.3). Thus,

$$\mathbb{E}\, c(F^*) \leq 2\, OPT_f.$$

Let us now consider the random multicut F once again. We recall that since the randomized algorithm invokes Algorithm 3.2.2, it forms the random edge subsets $F_1, \ldots, F_k$ as it executes. As a result, $c(F_i)$ is a random variable describing the cost of F_i for $i = 1, \ldots, k$. We can also denote τ as the index of the most costly of these edge subsets. Rather,

$$\tau := \min\{1 \leq i' \leq k : c(F_{i'}) = \max_{1 \leq i \leq k} c(F_i)\}.$$

Clearly, τ is a random variable with range $[k]$. Moreover, $c(F_\tau)$ is a random variable describing the cost of F_τ.

It is clear that for $i = 1, \ldots, k$, $F_i \subseteq F_i^*$. This implies that $F \subseteq F^*$, and so $c(F) \leq c(F^*)$. In particular, this means that,

$$\mathbb{E}\, c(F) \leq \mathbb{E}\, c(F^*) \leq 2\, OPT_f,$$

so already the multicut F achieves an expected approximation guarantee of 2. We shall improve this factor slightly by comparing $\mathbb{E}\, c(F)$ and $\mathbb{E}\, c(F^*)$ more closely. In particular, we claim that,

$$\mathbb{E}\, c(F) \leq \frac{k-1}{k}\, \mathbb{E}\, c(F^*).$$

In order to see this, first observe that,

$$c(F^*) = \sum_{i=1}^{k} c(F_i) - c(F_\tau),$$

as $F_1, \ldots, F_k$ were designed to be disjoint in Algorithm 3.2.2. On the other hand, we know that

$$c(F_\tau) \geq \sum_{i=1}^{k} \frac{c(F_i)}{k},$$

as F_τ is the most costly of $F_1, \ldots, F_k$. This implies that,

$$\mathbb{E}\, c(F_\tau) \geq \sum_{i=1}^{k} \mathbb{E}\, \frac{c(F_i)}{k},$$

after taking expectations. Combining these equations, we see that,

$$\begin{aligned}\mathbb{E}\, c(F^*) &= \sum_{i=1}^{k} \mathbb{E}\, c(F_i) - \mathbb{E}\, c(F_\tau) \\ &\leq \frac{k-1}{k} \sum_{i=1}^{k} \mathbb{E}\, c(F_i) \\ &= \frac{k-1}{k}\, \mathbb{E}\, c(F^*),\end{aligned}$$

where the last line follows since $F^* = \sqcup_{i=1}^{k} F_i$. To conclude, we have that,

$$\begin{aligned}\mathbb{E}\, c(F^*) &\leq \frac{k-1}{k}\, \mathbb{E}\, c(F^*) \\ &\leq 2\,(1 - \frac{1}{k})\, OPT_f,\end{aligned}$$

thus proving the initial claim.

□

While the randomized algorithm achieves an expected performance guarantee that is reasonably low, it still has a few shortcomings. Firstly, we are only aware of the expectation of the random variable $c(F)$. We have no information about how this random variable is distributed, and whether it is tightly concentrated about its expectation. As a result, $c(F)$ may vary greatly from its mean depending on the value ρ takes on. Moreover, the algorithm involves sampling a *real* number from the interval $[0, \frac{1}{2})$. As we formalize algorithms using Turing machines which work exclusively on rational numbers, this sampling process is outside of the capabilities of our computational model. This can be solved by sampling a rational number ρ^* which is *approximately* uniformly random on $[0, \frac{1}{2})$, but this

is computationally expensive, and involves comparing the cumulative distribution functions of ρ and ρ^*.

To get around these issues, we instead consider a process known as the *derandomization* of Algorithm 3.2.3. Broadly, this process modifying the algorithm in such a way that it no longer relies on randomness, without increasing its approximation guarantee. There are a number of widely used strategies for achieving this goal. We focus specifically on analyzing the various values that $c(F)$ may take on, as ρ varies throughout the interval $[0, \frac{1}{2})$. We shall see that there are in fact a finite number of values $r_1, \ldots, r_l$ in $[0, \frac{1}{2})$ for which the value of $c(F)$ changes. Moreover, we are able to compute these values in polynomial time, thus giving us a natural way to optimize the cost of F. Let us now outline how this is done:

Let e be an edge of G. For each $j = 1, \ldots, k$, let I_e^j be the half open interval on $[0, \frac{1}{2})$, such that for each $r \in I_e^j$, $e \in \delta(B_r(s_j))$. Moreover, set $I_e := \cup_{j=1}^k I_e^j$ to be the union of these intervals.

We define a function $\phi : [0, \frac{1}{2}) \to \mathbb{R}$, such that

$$\phi := \sum_{e \in E} c(e)\, \chi_{I_e},$$

where χ_{I_e} is the indicator function for the interval I_e.

If we once again consider the random variable ρ defined on the probability space $(\Omega, \mathcal{B}, \mathbb{P})$, then suppose F^* is the random edge set formed by taking the union of $\delta(B_\rho(s_1)), \ldots, \delta(B_\rho(s_k))$ (see the proof of Theorem 3.2.3). Observe that

for all $\omega \in \Omega$,

$$\phi(\rho(r)) = c(F^*)(\omega),$$

where $c(F^*)$ is the random variable describing the cost of F^*. As a result of this equation, we have that

$$\mathbb{E}\, c(F^*) = 2 \int_0^{\frac{1}{2}} \phi(r)\, dr.$$

In particular, this implies that

$$\inf_{0 \le r < 1/2} \phi(r) \le \mathbb{E}\, c(F^*) \tag{3.2.7}$$

$$\le 2\, OPT_f. \tag{3.2.8}$$

The second line follows as a result of the analysis of $c(F^*)$ in the proof of Theorem 3.2.5. We also observe that we can find a value $0 \le r_{min} < 1/2$ which attains this infimum. This is done by evaluating ϕ on the endpoints of each interval I_e^j, for $e \in E$ and $j = 1, \ldots, k$. Clearly, this may be done in $O(k\,|E|)$ many steps.

We now pass the radius r_{min} to Algorithm 3.2.2, yielding the determinisitc regions $B_{r_{min}}(s_1), \ldots, B_{r_{min}}(s_k)$, and the *disjoint* edge sets $F_1^{min}(s_1), \ldots, F_k^{min}(s_k)$. If we assume that F_k^{min} is the most costly of these edge sets, then Algorithm 3.2.2 returns the multicut $F^{min} = \cup_{i=1}^{k-1} F_i^{min}$. We claim that F^{min} has cost at most $2(1 - \frac{1}{k})OPT_f$.

To see this, first observe that,

$$\begin{aligned}\phi(r_{min}) &= \sum_{e \in E} c(e)\, \chi_{I_e}(r_{min}) \\ &= c(\cup_{i=1}^{k} B_{r_{min}}(s_i)) \\ &= \sum_{i=1}^{k} c(F_i^{min})),\end{aligned}$$

as $\sqcup_{i=1}^{k} F_i^{min} = \cup_{i=1}^{k} B_{r_{min}}(s_i)$ (where the left-hand side involves a disjoint union).

Moreover, $c(F_k^{min}) \geq \sum_{i=1}^{k} \frac{c(F_i^{min})}{k}$, as F_k^{min} is the most costly edge subset. Thus,

$$\begin{aligned}c(F^{min}) &= \sum_{i=1}^{k} c(F_i^{min}) - c(F_k^{min}) \\ &\leq \frac{k-1}{k} \sum_{i=1}^{k} c(F_i^{min}) \\ &= \frac{k-1}{k} \phi(r_{min}) \\ &\leq 2\,(1 - \frac{1}{k})\, OPT_f,\end{aligned}$$

where the last inequality follows from Equation 3.2.7. We now state an explicit implementation of this deterministic algorithm

Algorithm 3.2.4 Derandomization of the Region Cut Algorithm

Let $G = (V, E)$ be an undirected graph.
Suppose $c : E \to \mathbb{Q}_{\geq 0}$, and $\{s_1, \ldots, s_k\} \subseteq V$.

1: Set $F^{min} \leftarrow E$.
2: **for** $e \in E$ and $j = 1, \ldots, k$ **do**
3: Initialize r' and r'' as the endpoints of I_e^j.
4: Run Algorithm 3.2.2 with inputs r' and r''.
5: Let F' and F'' be the cuts returned.
6: Set F^{min} to be the cheaper of the cuts F' and F''.
7: Return the multiway cut F^{min}.

Using the above discussion, we arrive at the following theorem:

Theorem 3.2.7. *Algorithm 3.2.4 runs in polynomial time, and achieves an approximation guarantee of* $2 - \frac{2}{k}$.

If we restrict our attention to IP (3.2.1) when $k \geq 2$ is fixed, then Theorem 3.2.7 places an upper bound of $2 - \frac{2}{k}$ on the integrality gap of this restricted integer program. This is because it compares the cost of its multicut F^{min}, with the value of an optimum solution to LP (3.2.3). Rather, if I' corresponds to the *arbitrary* problem instance of G, c and $s_1, \ldots, s_k$ (here k is fixed), then the analysis of Theorem 3.2.7 yields the comparison,

$$OPT(I') \leq c(F^{min}) \leq 2\,(1 - \frac{1}{k})\,OPT_f(I'),$$

where $OPT(I')$ is the cost of an optimum solution to IP (3.2.1), and $OPT_f(I')$ is the cost of an optimum solution to LP (3.2.3), both involving instance I'.

On the other hand, the integrality gap of IP (3.2.1) is defined to be,

$$\sup_I \frac{OPT(I)}{OPT_f(I)},$$

where the supremum is taken over all problem instances of multiway cut, in which k source nodes are to be separated. The combination of these inequalities yields the upper bound on the integality gap. We now observe that this bound is in fact tight, as evident from the following example:

Example 3.2.8. *Let $H = (V, E)$ be an undirected graph with source nodes $s_1, \ldots, s_k$ and regular nodes $v_1, \ldots, v_k$. More, let us denote its capacity function by $c : E \to \mathbb{Q}_{\geq 0}$. We now construct the edges of H. For $i = 1, \ldots, k$,*

- *Add edge $\{s_i, v_i\}$ to H.*
- *If $i < k$, add edge $\{v_i, v_{i+1}\}$ to H.*

Moreover, all the edges of H are defined to have cost equal to one; rather, $c(e) = 1$ for all $e \in E$. We denote I as this particular problem instance, where I includes G,c and $s_1, \ldots, s_k$.

Let us first denote $F^* := \{\{s_i, v_i\} : i = 1, \ldots, k-1\}$. We observe that F^* is an optimum multiway cut. Rather,

$$OPT(I) = c(F^*) = k - 1.$$

We also denote $\boldsymbol{d} = (d_e)_{e \in E}$ as a fractional multiway cut, where $d_{s_i, v_i} := \frac{1}{2}$ for $i = 1, \ldots, k$, and zero everywhere else. It is clear that $\boldsymbol{d}$ is an optimum solution to LP (3.2.3). Rather,

$$OPT_f(I) = \sum_{e \in E} c(e)\, d_e = \frac{k}{2}.$$

On the other hand, we have that,

$$\frac{OPT(I)}{OPT_f(I)} = 2 - \frac{2}{k}.$$

Thus, the integrality gap of instance I is precisely $2 - \frac{2}{k}$.

With this class of examples, together with Theorem 3.2.7, we observe the following theorem and corollary.

Theorem 3.2.9. *If $k \geq 2$ is fixed for IP* (3.2.1)*, then its integrality gap is exactly* $2(1 - \frac{1}{k})$.

Corollary 3.2.10. *If $k \geq 2$ is allowed to vary in IP* (3.2.1)*, then its integrality gap is exactly* 2.

3.3 A Second IP Formulation of Multiway Cut

Let us suppose that $\mathcal{A}$ is an arbitrary algorithm for approximating optimum multiway cuts. Moreover, suppose that we are able to prove that there is some $\alpha \geq 1$, such that that for any instance I' of the problem, $\mathcal{A}$ returns a solution with cost at most $\alpha\, OPT_f(I')$. That is, if the instance I' contains a graph $G = (V, E)$, together with a cost function $c : E \to \mathbb{Q}_{\geq 0}$, and $k \geq 2$ distinct source nodes, then $\mathcal{A}$ returns a multiway cut $F \subseteq E$, for which,

$$c(F) \leq \alpha \sum_{e \in E} c(e)\, d_e = \alpha\, OPT_f(I'),$$

where $\boldsymbol{d} = (d_e)_{e \in E}$ is an optimum solution to LP (3.2.3). In this case, we may conclude that,

$$OPT(I') \leq c(F) \leq \alpha\, OPT_f(I'),$$

where $OPT(I')$ is the cost of an optimum solution to IP (3.2.1). On the other hand, the integrality gap of IP (3.2.1) is defined as,

$$\sup_I \frac{OPT(I)}{OPT_f(I)},$$

where the supremum is taken over all instances of the multiway cut problem. As we saw in the previous section, this integrality gap is precisely two, so it follows that $\alpha \geq 2$.

Thus, the integrality gap of IP (3.2.3) limits how well a large class of algorithms can approximate optimum multiway cuts. In particular, algorithms which apply linear programming techniques to round solutions of LP (3.2.1) to solutions

of IP (3.2.3) typically fall into this category. As this is the case, we can't expect there to exist a smarter rounding algorithm than we saw in the previous section.

If we would like to improve upon the approximation guarantees that we've seen, then one approach is to introduce a new integer programming formulation of the multiway cut problem. Ideally, this alternative formulation will have a smaller integrality gap than seen in IP (3.2.1). If this is indeed the case, then (randomized) rounding algorithms will not suffer the same restrictions as outlined above. The primary goal of this section is to explore this approach.

Suppose we are given a connected graph $G = (V, E)$, a cost function $c : E \to \mathbb{Q}_{\geq 0}$, and source nodes $s_1, \ldots, s_k$. Moreover, assume we are given a *minimal* multiway cut F of G. That is, a multiway cut which doesn't contain any multiway cut of lesser cost. Once we remove F from G, we know that $G \setminus F$ has at least k distinct components - one for each source node. In turns out that the minimality of F ensures that $G \setminus F$ has exactly k components. To see this holds, suppose we assume otherwise. In this setting, there are components $U_1, \ldots, U_k$ in $G \setminus F$ for which $s_i \in V_i$, $i = 1, \ldots, k$. Moreover, we have that $U := G \setminus \cup_{i=1}^{k} U_i$ is nonempty and contains no source node. As G is connected, we know that there is a component, say U_i, such that $e_G(U, U_i) \neq \emptyset$. Observe that if e is any edge between U and U_i, then e is a member of F. Moreover, if $F' := F \setminus e$, then F' separates all the source nodes, is contained in F, and yet has cost strictly less than F. As F was assumed to be minimal, this is a contradiction.

This means that given any minimal multiway cut F, we know that $G \setminus F$ has exactly k components. We can therefore associate to F a partition of the

vertices into k components. Conversely, given any partition of the vertices into k components, each containing exactly one source node, we can recover a multiway cut for G. This is easily done by collecting all edges in G that lie between distinct components. It is clear that this cut will indeed be minimal.

We now introduce an integer program which formulates this interpretation of the multiway cut problem. Suppose we are given a solution in the form of a partition $U_1, \ldots, U_k$ of V, where $s_i \in U_i$ for $i = 1, \ldots, k$. For each vertex $v \in G$, we introduce the variables x_i^v for $i = 1, \ldots, k$. We ensure that x_i^v is one, provided v_i is placed into U_i, and zero otherwise. Moreover, for each edge $e \in E$, we define the variables y_i^e for $i = 1, \ldots, k$, where y_i^e is one if $e \in \delta(U_i)$, and zero otherwise. If the multiway cut $F := \cup_{i=1}^k \delta(U_i)$, then the cost of F, denoted $c(F)$, may be related to these variables as follows:

$$c(F) = \frac{1}{2} \sum_{e \in E} c(e) \sum_{i=1}^{k} y_i^e.$$

We introduce the factor of $\frac{1}{2}$ in front of this sum, as each edge $e \in F$ occurs in exactly two $\delta(U_i)$ edge subsets.

Our goal now is to encode the restrictions of these variables as an integer program. First, observe that each variable is restricted to the range $\{0, 1\}$. Moreover, we enforce that $x_i^{s_i} = 1$, as every solution must have $s_i \in U_i$, for each $i = 1, \ldots, k$. We also observe that $\sum_{i=1}^k x_i^v = 1$ for each $v \in G$, as $U_1, \ldots, U_k$ form a partition of the vertices of G.

The final requirement of the integer program is that for each edge $e \in E$, $y_i^e \geq x_i^u - x_i^v$ and $y_i^e \geq x_i^v - x_i^u$, where $e = \{u, v\}$, and $i = 1, \ldots, k$. These inequalities ensure that $y_i^e = 1$, if and only if $e \in \delta(U_i)$.

Summarizing, we have the following integer program:

$$\begin{aligned}
\text{minimize} \quad & \frac{1}{2} \sum_{e \in E} c(e) \sum_{i=1}^{k} y_i^e & & \\
\text{subject to} \quad & \sum_{i=1}^{k} x_i^v = 1 & & \forall v \in V, \\
& x_{s_i}^i = 1 & & \forall i = 1, \ldots, k, \\
& y_i^e \geq x_i^u - x_i^v, & & \\
& y_i^e \geq x_i^v - x_i^u & & \forall i = 1, \ldots, k, \text{ and } e = \{u, v\} \in E \ , \\
& y_i^e, x_i^v \in \{0, 1\} & & \forall i = 1, \ldots, k,\ e \in E \text{ and } v \in V.
\end{aligned} \tag{3.3.1}$$

We now wish to consider a relaxation of this integer program. For each variable in IP (3.3.1), let us extend its domain to the interval $[0, 1]$. Since the objective function is to be minimized, and the cost function c is nonnegative, we can in fact relax the upper bound on these variables. This yields the following

linear program:

$$\begin{aligned}
\text{minimize} \quad & \frac{1}{2}\sum_{e\in E} c(e) \sum_{i=1}^{k} y_i^e \\
\text{subject to} \quad & \sum_{i=1}^{k} x_i^v = 1 && \forall v \in V, \\
& x_{s_i}^i = 1 && \forall i = 1, \ldots, k, \\
& y_i^e \geq x_i^u - x_i^v, \\
& y_i^e \geq x_i^v - x_i^u && \forall i = 1, \ldots, k, \text{ and } e = \{u, v\} \in E\ , \\
& y_i^e, x_i^v \geq 0 && \forall i = 1, \ldots, k,\ e \in E \text{ and } v \in V.
\end{aligned} \tag{3.3.2}$$

Observe that given an optimum solution to LP (3.3.2), we have that $y_i^e = |x_i^v - x_i^u|$ for each $e = \{u, v\} \in E$, and $i = 1, \ldots, k$. Thus, we can restrict our attention to solutions to LP (3.3.2), for which this property is satisfied. This leaves us with a simplified optimization problem, in which we eliminate the edge variables:

$$\begin{aligned}
\text{minimize} \quad & \frac{1}{2}\sum_{e=\{u,v\}\in E} c(e) \sum_{i=1}^{k} |x_i^u - x_i^v| \\
\text{subject to} \quad & \sum_{i=1}^{k} x_i^v = 1 \quad \forall v \in V, \\
& x_{s_i}^i = 1 \quad \forall i = 1, \ldots, k, \\
& x_i^v \geq 0 \quad \forall i = 1, \ldots, k, \quad e \in E \text{ and } v \in V.
\end{aligned} \tag{3.3.3}$$

We can simplify this formulation further, by introducting a vector $\boldsymbol{x}^v \in \mathbb{R}^k$ for each $v \in V$, where $\boldsymbol{x}^v := (x_1^v, \ldots, x_k^v)$. For each $i = 1, \ldots, k$, we also introduce a vector $\boldsymbol{e}^i \in \mathbb{R}^k$, which is zero everywhere, except for its i^{th} component, where it is

one. If Δ_k is the *convex hull* of $\boldsymbol{e}^1, \ldots, \boldsymbol{e}^k$, then it is clear that,

$$\Delta_k = \{\boldsymbol{x} \in \mathbb{R}^k : \sum_{i=1}^{k} x_i = 1\}.$$

Moreover, we have the following optimization problem, which we refer to as a *vector optimization problem* (VOP):

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} \sum_{e=\{u,v\} \in E} c(e) \|\boldsymbol{x}^u - \boldsymbol{x}^v\|_1 \\ \text{subject to} \quad & \boldsymbol{x}^v \in \Delta_k \quad \forall v \in V, \\ & \boldsymbol{x}^{s_i} = \boldsymbol{e}^i \quad \forall i = 1, \ldots, k. \end{aligned} \tag{3.3.4}$$

A feasible solution to this optimization problem is an assignment to each vector $\boldsymbol{x}^v$ for $v \in V$, such that $(\boldsymbol{x}^v)_{v \in V}$ satisfies the constraints of the problem. While VOP (3.3.4) is technically not a linear program, we have shown that it can be reduced to LP (3.3.2). In particular, this means that an optimum solution to this problem can be found in polynomial time. Our goal now will be to show how we can use this solution to find a find an approximately optimum solution to IP (3.3.1).

Let us suppose that $(\boldsymbol{x}^v)_{v \in V}$ is an optimum solution to VOP (3.3.4). We may think of this solution as an embedding of the vertices of G into the k-simplex, Δ_k. That is, vertex v is mapped to the vector $\boldsymbol{x}^v$ of Δ_k for each $v \in V$. We may think of Δ_k as equipped with the ℓ_1 norm. In particular, for $\boldsymbol{x} \in \Delta_k$, $B_r(\boldsymbol{x}) := \{\boldsymbol{x'} \in \Delta_k : \|\boldsymbol{x} - \boldsymbol{x'}\|_1 \leq r\}$ for $r \geq 0$.

Now, for each $i = 1, \ldots, k$, we know that s_i is mapped to the vector $\boldsymbol{e}^i$ in Δ_k. The goal of our algorithm will be to choose a radius $r \geq 0$, and sequentially grow

a region $B_r(\boldsymbol{e}^i)$ about each vertex $\boldsymbol{e}^i$ for $i = 1, \ldots, k$. The vertices of V whose embedding falls into $B_r(\boldsymbol{e}^i)$, will then be included into set U_i for $i = 1, \ldots, k$. Any vertices which fail to be partitioned can then be arbitarily assigned to $U_1, \ldots, U_k$.

In order to ensure that each vertex is assigned to exactly one partite set, we shall give precedence to source vertices whose regions are processed earliest. Thus, in addition to providing a radius r, we must also provide a permutation $\pi : [k] \to [k]$, which specifies when each source node should be processed. We outline the above algorithm in detail below:

Algorithm 3.3.1 Multiway Partition Algorithm

Let $G = (V, E)$ be a connected and undirected graph.
Suppose $c : E \to \mathbb{Q}_{\geq 0}$, and $\{s_1, ..., s_k\} \subseteq V$.
Let $0 < r < 2$.
Let π be a permutation of $[k]$.
1: Initialize $F \leftarrow \emptyset$.
2: Initialize $L \leftarrow \emptyset$.
3: Solve VOP (3.3.4), and obtain an optimum embedding, where $\boldsymbol{x}^v \in \Delta_k$ for each $v \in V$.
4: **for** $i = 1, \ldots k-1$ **do**
5: Let $U_{\pi(i)} \leftarrow \{v \in V\colon \boldsymbol{x}^v \in B_r(\boldsymbol{e}^{\pi(i)})$ and $v \notin L\}$.
6: Let $L \leftarrow L \cup U_{\pi(i)}$.
7: Let $U_{\pi(k)} \leftarrow V \setminus L$.
8: Let $F \leftarrow \cup_{i=1}^{k} \delta(U_i)$.
9: Return F.

We remark that the value of the radius is bounded above to ensure that the source nodes remain in disjoint sets. It follows that for any choice of $0 < r < 2$ and permutation π, the algorithm returns a valid multiway cut.

Let us now outline a randomized algorithm which attains a better approximation guarantee than seen in the previous sections. We first choose ρ uniformly

at random from the interval $(0, 2)$, and then we independently choose σ from all the permutations of $[k]$, uniformly at random. Using these values as inputs, we run Algorithm 3.3.1 and then return the resulting output, thus completing the computation. We observe the following theorem regarding this randomized algorithm:

Theorem 3.3.1. *The randomized version of Algorithm 3.3.1 runs in polynomial time and achieves an expected approximation guarantee of* $\frac{3}{2}$.

In order to prove this theorem, we first consider an essential lemma stated below:

Lemma 3.3.2. *If* $e = \{u, v\}$ *is an edge of* G, *then* $\mathbb{P}(e \in F) \leq \frac{3}{4}\|\boldsymbol{x}^u - \boldsymbol{x}^v\|_1$, *where* $\boldsymbol{x}^u$ *and* $\boldsymbol{x}^v$ *are the embeddings of* u *and* v *used in the randomized version of Algorithm 3.3.1.*

Let us assume for now that this lemma is true. We shall first see how we can use it to prove Theorem 3.3.1.

Proof. Let $G = (V, E)$ be a connected graph, $c : E \to \mathbb{Q}_{\geq 0}$ a cost function, and $\mathcal{S}$ a collection of $k \geq 1$ source nodes of G. We denote this problem instance of multiway cut by $I := (G, c, \mathcal{S})$. Moreover, we may denote the multiway cut returned by executing Algorithm 3.3.1 with these inputs by F. If $c(F)$ is the cost of this cut, then $c(F)$ is a random variable. Our goal is to show that,

$$\mathbb{E}\, c(F) \leq \frac{3}{2}\, OPT(I),$$

where $OPT(I)$ is the cost of an optimum multiway cut for the problem instance I.

For each edge $e = \{u, v\} \in E$, we introduce an indicator random variable Z_e, which is nonzero if and only if $e \in F$. This yields the following relation:

$$c(F) = \sum_{e \in E} c(e)\, Z_e$$

Taking expectations on both sides, we see that:

$$\begin{aligned}
\mathbb{E}\, c(F) &= \sum_{e \in E} c(e) \mathbb{E} Z_e \\
&= \sum_{e \in E} c(e) \mathbb{P}(e \in F) \\
&\leq \frac{3}{4} \sum_{e=\{u,v\} \in E} c(e) \|\boldsymbol{x}^u - \boldsymbol{x}^v\|_1,
\end{aligned}$$

after applying Lemma 3.3.2. Now, $(\boldsymbol{x}^v)_{v \in V}$ is assumed to be an optimum solution to VOP 3.3.4. Thus, if $OPT_f(I)$ is the cost of an optimum solution to LP (3.3.2), we know that,

$$OPT_f = \frac{1}{2} \sum_{e=\{u,v\} \in E} c(e) \|\boldsymbol{x}^u - \boldsymbol{x}^v\|.$$

Moreover, as LP (3.3.2) is the linear relaxation of IP (3.3.1), we have that $OPT_f(I) \leq OPT(I)$. It follows that

$$\mathbb{E}\, c(F) \leq \frac{3}{2}\, OPT(I),$$

thus proving the claim.

□

Our goal now is to prove Lemma 3.3.2. In order to do so, we first prove a few results regarding the ℓ_1-norm on Δ_k.

Lemma 3.3.3. *If $\boldsymbol{x}, \boldsymbol{y} \in \Delta_k$, then for $j = 1, \ldots, k$, $|x_j - y_j| \leq \frac{1}{2}\|\boldsymbol{x} - \boldsymbol{y}\|_1$.*

Proof. Assume without loss of generality that $x_j \geq y_j$. It follows that,

$$|x_j - y_j| = x_j - y_j = (1 - \sum_{i \neq j} x_i) - (1 - \sum_{i \neq j} y_i) = \sum_{i \neq j} (x_i - y_i).$$

Now,

$$\sum_{i \neq j} (x_i - y_i) \leq \sum_{i \neq j} |x_i - y_i|.$$

Thus,

$$2|x_j - y_j| \leq \sum_{i \neq j} (x_i - y_i) + |x_j - y_j| = \|\boldsymbol{x} - \boldsymbol{y}\|_1.$$

The result therefore follows. □

Lemma 3.3.4. *If $\boldsymbol{x} \in \Delta_k$ and $r \geq 0$, then we have that for $1 \leq j \leq k$, $\boldsymbol{x} \in B_r(\boldsymbol{e}^j)$ if and only if $1 - x_j \leq \frac{r}{2}$.*

Proof. We first observe that,

$$\|\boldsymbol{x} - \boldsymbol{e}^j\|_1 = 1 - x^j + \sum_{i \neq j} x_i.$$

Now, $1 - x_j = \sum_{i \neq j} x_i$. Thus,

$$2\,(1 - x_j) = \|\boldsymbol{x} - e_j\|_1.$$

It follows that,

$$\boldsymbol{x} \in B_r(\boldsymbol{e}^j), \text{ if and only if } 2\,(1 - x_j) \leq r.$$

□

Proof. Let us now return to the statement of Lemma 3.3.2. We recall that σ is a uniformly random permutation of $[k]$, and ρ is a uniformly random number from $(0, 2)$. By definition, σ and ρ are drawn independently. Given an edge $e = \{u, v\} \in E$, we shall show that $\mathbb{P}(e \in F) \leq \frac{3}{4} \|\boldsymbol{x}^u - \boldsymbol{x}^v\|_1$.

We first introduce a number of *events* involving the fixed edge e. For $1 \leq i \leq k$, we say that source node s_i *affects* e, provided $\sigma^{-1}(i)$ is the smallest index in $[k]$, for which at least one of u, v is in $B_\rho(\boldsymbol{e}^i)$. We denote A_i to be the event in which this occurs.

Similarly, we say that s_i *separates* e, provided exactly one of u, v is in $B_\rho(\boldsymbol{e}^i)$. We denote this event by S_i. It is clear that the event S_i is independent of the random permutation σ, as it only depends on the value of ρ.

We observe that $e \in F$, if and only if there is some $1 \leq i \leq k$, for which s_i affects *and* separates e. Thus, applying the union bound, we have that,

$$\begin{aligned}\mathbb{P}(e \in F) &= \mathbb{P}(\cup_{i=1}^{k} A_i \cap S_i) \\ &\leq \sum_{i=1}^{k} \mathbb{P}(A_i \cap S_i).\end{aligned}$$

We now estimate the value of $\mathbb{P}(A_i \cap S_i)$ for $i = 1, \ldots, k$. Let us fix source node s_i. Moreover, assume that $\|\boldsymbol{x}^u - \boldsymbol{e}^i\|_1 \leq \|\boldsymbol{x}^v - \boldsymbol{e}^i\|_1$. That is, $\boldsymbol{e}^i$ is closer to $\boldsymbol{x}^u$ than $\boldsymbol{x}^v$ in Δ_k. This implies that, $x_i^u \geq x_i^v$.

By applying Lemma 3.3.4, we see that S_i occurs if and only if $1 - x_i^u \leq \frac{\rho}{2} < 1 - x_i^v$. Thus, $\mathbb{P}(S_i) = |x_i^u - x_i^v|$.

Immediately, we may conclude that,

$$\begin{aligned}\mathbb{P}(e \in F) &\leq \sum_{i=1}^{k} |x_i^u - x_i^v| \\ &= \|\boldsymbol{x}^u - \boldsymbol{x}^v\|_1.\end{aligned}$$

In order to improve this upper bound, we need to incorporate the likelihood of a source vertex affecting the edge e.

Toward this goal, for each $i = 1, \ldots, k$, let us first consider the smaller of the distances $\|\boldsymbol{e}^i - \boldsymbol{x}^u\|_1$ and $\|\boldsymbol{e}^i - \boldsymbol{x}^v\|_1$. We refer to this value as the *distance* between s_i and e. Observe that there is an index $1 \leq l \leq k$, for which the distance between s_l and e is minimal. That is, e_l is the closest point to one of x_u and x_v, among all source node candidates.

We observe that for any $1 \leq i \leq k$, if $\sigma^{-1}(l) < \sigma^{-1}(i)$, then s_i cannot separate e. Thus, if $i \neq l$, we have that, $\mathbb{P}(A_i \cap S_i | \, \sigma^{-1}(l) < \sigma^{-1}(i)) = 0$. Applying conditional probabilities, this implies that,

$$\mathbb{P}(A_i \cap S_i) = \mathbb{P}(A_i \cap S_i | \, \sigma^{-1}(i) < \sigma^{-1}(l)) \, \mathbb{P}(\sigma^{-1}(i) < \sigma^{-1}(l)).$$

Now, we know that since σ is a uniformly random permutation,

$$\mathbb{P}(\sigma^{-1}(i) < \sigma^{-1}(l)) = \frac{1}{2}.$$

Moreover,

$$\mathbb{P}(A_i \cap S_i | \, \sigma^{-1}(i) < \sigma^{-1}(l)) \leq \mathbb{P}(S_i | \, \sigma^{-1}(i) < \sigma^{-1}(l)).$$

Of course, S_i is independent of σ, and so in particular, it is independent of the event, $\sigma^{-1}(i) < \sigma^{-1}(l)$. Thus, $\mathbb{P}(S_i | \, \sigma^{-1}(i) < \sigma^{-1}(l))) = \mathbb{P}(S_i) = |x_i^u - x_i^v|$. It follows that $\mathbb{P}(A_i \cap S_i) \leq \frac{1}{2}|x_i^u - x_i^v|$, provided $i \neq l$. Thus,

$$\begin{aligned}
\mathbb{P}(e \in F) &\leq \sum_{i=1}^{k} \mathbb{P}(S_i \cap A_i) \\
&= \mathbb{P}(S_l \cap A_l) + \sum_{i \neq l} \mathbb{P}(S_i \cap A_i) \\
&\leq |x_l^u - x_v^l| + \frac{1}{2} \sum_{i \neq l} |x_i^u - x_i^v| \\
&= \frac{1}{2}|x_l^u - x_v^l| + \frac{1}{2}\|\boldsymbol{x}^u - \boldsymbol{x}^v\|_1.
\end{aligned}$$

Now, by Lemma 3.3.3,

$$|x_l^u - x_v^l| \leq \frac{1}{2}\|\boldsymbol{x}^u - \boldsymbol{x}^v\|_1.$$

Thus, we have that,

$$\mathbb{P}(e \in F) \leq \frac{3}{4}\|\boldsymbol{x}^u - \boldsymbol{x}^v\|_1,$$

concluding the proof of the initial claim.

□

As consequence of the proof of this Lemma, we have shown that the randomized version of Algorithm 3.3.1 has an expected approximation guarantee of $\frac{3}{2}$. That being said, we can reduce the number of random bits needed to execute this algorithm by a significant degree. To see this, consider permutations π_1 and π_2, where $\pi_1(i) := i$ and $\pi_2(i) := k - (i-1)$ for $i = 1, \ldots, k$. Our alternative algorithm selects a random permutation σ that is π_1 or π_2 with equal probability. It then

independently chooses a radius $0 < \rho < 2$ uniformly at random. Using these parameters as inputs, it then executes Algorithm 3.3.1. Let us formally outline this procedure:

Algorithm 3.3.2 Efficient Randomized Multiway Partition Algorithm

Let $G = (V, E)$ be an undirected graph.
Suppose $c : E \to \mathbb{Q}_{\geq 0}$, and $\{s_1, ..., s_k\} \subseteq V$.
1: Generate $0 < \rho < 2$ uniformly at random.
2: Generate σ at random, with equally probable outcomes π_1 and π_2.
3: Execute Algorithm 3.3.1 with the above inputs.
4: Let F be the multiway cut returned from this algorithm.
5: Return F.

Theorem 3.3.5. *Algorithm 3.3.2 achieves an expected approximation guarantee of* $\frac{3}{2}$.

Proof. We observe that, for fixed $1 \leq i < j \leq k$, the events $\sigma^{-1}(i) < \sigma^{-1}(j)$ and $\sigma^{-1}(j) < \sigma^{-1}(i)$ occur with equal probability.

This is the main property required of the random permutation in the proof of Lemma 3.3.2. As the generation of σ is the only way in which the algorithm changes, we may apply the same analysis as previously seen.

□

It turns out that restricting the assignments of σ allows us to derandomize Algorithm 3.3.2. At a high level, this procedure first involves fixing one of the two assignments that σ may take on. Once this is done, an optimum choice of ρ is found, using a procedure similar to that of Algorithm 3.2.4 from the previous section. At this point, the multiway cuts F_1 and F_2 are stored, each respectively corresponding to the permutations π_1 and π_2. The cut of smaller cost is then

returned as the output of this procedure. This derandomization procedure achieves an approximate guarantee no worse than Algorithm 3.3.2, thus yielding a deterministic algorithm for computing a multiway cut of the desired optimality. We leave the details of this argument to the reader.

Theorem 3.3.6. *There exists a derandomization of Algorithm 3.3.2, which achieves a performance guarantee of* $\frac{3}{2}$.

CHAPTER 4
The Multicut Problem

This chapter focuses specifically on the multicut problem, a further generalization of the multiway cut problem. The first section provides a detailed explanation of the problem, and relates it back to the cut problems previously studied. It also considers a restriction of the problem to connected noncyclic graphs (trees). An approximation preserving reduction is then presented from the vertex cover problem to the multicut problem on trees of height one. This establishes the **NP**-hardness of the multicut problem on trees, thus motivating the development of approximation algorithms in the subsequent sections.

The second and third sections of the chapter focus exclusively on the multicut problem on trees. An IP formulation of this problem is presented, together with a simplified LP relaxation of this program. The dual of this LP is then taken, and defined to be the multi-commodity flow problem on trees. As in the case of a single-source pair for network flow, this multi-commodity flow problem has an integral version, in which the flows are restricted to integers. A primal-dual algorithm is then presented, which simultaneously builds both a multicut as well as an integral multi-commodity flow. In the former case, an approximation guarantee of 2 is witnessed, whereas in the later case, an approximation of $1/2$ is seen. The material from these sections is heavily based on the material from the book "Approximation Algorithms" [Vv11].

The final section of this chapter involves a study of the multicut problem on general graphs. An IP formulation of this problem is again presented, generalizing the formulation of the previous section. Unlike before, this integer program has exponentially many constraints, and so its LP relaxation requires a polynomial time separation oracle to remain solvable. Such an oracle is presented, and used to solve the LP relaxation in polynomial time. Two rounding algorithms are then analyzed, where the first achieves an expected approximation guarantee of k, where $k \geq 1$ is the number of source-sink pairs to be separated. I independently designed this algorithm as a extension of the techniques used in the multiway cut problem. This algorithm is then improved to deterministically achieve an approximation guarantee of $4 \ln(k + 1)$. The section ends with a brief discussion of the integrality gap of the multicut problem on general graphs. The majority of the material from this section is based on the books "Approximation Algorithms" [Vv11] and "the Design of Approximation Algorithms" [WS11].

4.1 Introduction to Multicut

Suppose we are given undirected graph $G = (V, E)$, together with and a cost function $c : E \to \mathbb{Q}_{\geq 0}$. Moreover, suppose $\{(s_i, t_i)\}_{1 \leq i \leq k}$ is a collection of $k \geq 1$ distinct vertex pairs of G. For each $1 \leq i \leq k$, we refer to (s_i, t_i) as a *source-sink pair*, where s_i is the *source* and t_i is the *sink*. Given an edge subset, $F \subseteq E$, we refer to F as a *multicut*, provided each source-sink pair is disconnected in $G \setminus F$. We typically denote $c(F)$ as the cost of F, where $c(F) := \sum_{e \in F} c(e)$. The goal of the problem is to compute multicuts of small cost.

Problem 4.1.1 (Multicut on General Graphs)**.** *Let $G = (V, E)$ be an undirected graph, and $c : E \to \mathbb{Q}_{\geq 0}$ a cost function. Moreover, suppose $\{(s_i, t_i)\}_{1 \leq i \leq k}$ is a collection of $k \geq 1$ distinct source-sink pairs, The goal of the multicut problem is to choose a multicut F of G, whose cost is minimal; rather, $c(F)$ is as small as possible.*

We observe that if $k = 2$, then the multicut problem is equivalent to the minimum cut problem on undirected graphs, and thus can be solved in polynomial time (see Section 2.3). If we instead look at the problem any fixed $k \geq 3$, then it is **NP**-hard. This can easily be seen, as the multicut problem is a generalization of the multiway cut problem. In particular, if we are given $l \geq 1$ source nodes $s_1, \ldots, s_l$ of G, then finding a minimum multiway cut for these nodes is equivalent to finding a multicut for the $\binom{l}{2}$ source-sink pairs $\{(s_i, s_j)\}_{1 \leq i < j \leq l}$ (see Section 3.1)

We shall first consider a special case of the multicut problem, where we restrict our attention to trees instead of general graphs. It turns out that even if we limit ourselves further to trees of height one, the problem remains **NP**-hard.

Recall that given a graph $G = (V, E)$ with vertex weight $w : V \to \mathbb{Q}_{\geq 0}$, the goal of the *vertex cover problem* is to find a minimum weight subset of vertices, $U \subseteq V$, for which each edge of E is incident to a vertex of U. If U satisfies these covering properties, then it is referred to as a *vertex cover for* G. In this case, we denote the *weight* of U by $w(U)$, where $w(U) := \sum_{v \in U} w(v)$. If we restrict our attention to graphs with unit weight vertices, then this is known as the *cardinality vertex cover problem*. It is well known that both variants are **NP**-hard. There is a natural correspondence between the multicut problem on trees of height one, and the vertex cover problem.

Proposition 4.1.1. *There exists an approximation preserving reduction from the vertex cover problem, to the multicut problem on trees of height one (see Definition 1.2.5 from Subsection 1.2.1).*

Moreover, if we restrict ourselves to the cardinality vertex cover problem, then the reduction is to the multicut problem on trees of height one and unit weight edges.

Proof. For convenience, let us denote Π_1 as the vertex cover problem, and Π_2 as the multicut problem on trees of height one.

If $G = (V, E)$ is a graph with a weight function $w : V \to \mathbb{Q}_{\geq 0}$, then let us assume that G has n vertices and m edges. We note that (G, w) is an instance of the vertex cover problem. Our first goal is to design a polynomial time algorithm $\mathcal{A}_1$, which inputs the instance (G, w), and returns an instance of the multicut problem on trees of height one.

Let T be a tree which is isomorphic to $K_{1,n}$, a star on $n+1$ vertices. Furthermore, let us assume that the leaves of T are in fact the vertices of G, and the non-leaf of T is labelled by ν. For each edge $\{u, v\} \in E$, we add the source-sink pair (u, v) to the collection $\mathcal{S}$, leaving us with a total of m pairs.

It remains to construct a cost function c for the edges of T. If $v \in V$ and ν is the root of T, then $c(e) := w(v)$, where $e := \{u, \nu\}$. We are now left with $(T, \mathcal{S}, c)$, an instance to the multicut problem on trees of height one. It is clear that the algorithm $\mathcal{A}_1$ may be designed to construct $(T, \mathcal{S}, c)$ from (G, w) in polynomial time. Thus, we have that $\mathcal{A}_1(G, w) := (T, \mathcal{S}, c)$.

In order to complete the reduction, we must design an algorithm $\mathcal{A}_2$, whose inputs include (G, w), as well as a multicut F of T. This algorithm must output a vertex cover of U of G.

Let us suppose that $F \subseteq E$ is a multicut of T. For each edge $(u, r) \in F$, we include u in our set U. We observe that since each source-sink pair is separated by F, we know that each edge of G will be incident to a node in U. It follows that U is a vertex cover of G. It is clear that the construction of U can be completed by $\mathcal{A}_2$ in time polynomial in the size of F, so we set $\mathcal{A}_2(G, w, F) := U$.

It remains to verify that this reduction is in fact approximation preserving. In other words, we must show that,

$$OPT_{\Pi_2}(\mathcal{A}_1(G, w)) \leq OPT_{\Pi_1}(G, w),$$

and that

$$w(\mathcal{A}_2(G, w, F)) \leq c(F).$$

It is clear that the second equation holds, as the weight of U is equal to the cost of F, and $\mathcal{A}_2(G, w, F) = U$.

In order to prove the first inequality, we observe that given *any* vertex cover U' of G, we can construct a multicut F' of T, such that,

$$c(F') \leq w(U'),$$

by reversing the steps of the algorithm $\mathcal{A}_2$. It follows that,

$$OPT_{\Pi_2}(T, \mathcal{S}, c) \leq w(U')$$

for all vertex covers U' of G. Thus,

$$OPT_{\Pi_2}(T, \mathcal{S}, c) \leq OPT_{\Pi_1}(G, w),$$

so the first equation holds since $\mathcal{A}_1(G, w)) = (T, \mathcal{S}, c)$. The reduction is therefore approximation preserving, thus proving the initial claim. The same reduction can be used when looking at the cardinality vertex cover problem, and the multicut problem on unit weight trees of height one.

□

We therefore have the following corollary regarding the multicut problem:

Corollary 4.1.2. *The multicut problem on trees of height one and unit weight weights is* ***NP****-hard.*

Proof. It is known that the cardinality vertex cover problem is **NP**-hard. As a consequence of the approximation preserving reduction, this implies that the

multicut problem on trees of height one and unit weight edges is also **NP**-hard, by Proposition 1.2.6 from Subsection 1.2.1.

□

4.2 Multi-Commodity Flow on Trees

We now return to our original goal of designing an approximation algorithm for the multicut problem on trees. Let $T = (V, E)$ be a tree with $k \geq 1$ source-sink pairs $\mathcal{S} = \{(s_i, t_i)\}_{i=1}^{k}$, and a cost function $c : E \to \mathbb{Q}_{\geq 0}$. For each $1 \leq i \leq k$, let p_i be the unique path from s_i to t_i in T.

Let us introduce an integer program that incorporates the restrictions of the problem. For each edge e of the tree T, we associate an indicator variable d_e, which is nonzero, if and only if the edge e is taken in the multicut $F \subseteq E$. In order to enforce that the source-sink pairs should be disconnected in $T \setminus F$, we impose the condition,

$$\sum_{e \in p_i} d_e \geq 1$$

for each $i = 1, \ldots, k$.

We are now left with choosing the appropriate objective function of the integer program. It is clear that if F is the multicut chosen, then we have that $c(F) = \sum_{e \in E} c(e) d_e$. To summarize, we observe the following integer program:

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c(e)\, d_e \\ \text{subject to} \quad & \sum_{e \in p_i} d_e \geq 1 \qquad \forall i = 1, \ldots, k, \\ & d_e \in \{0, 1\} \quad \forall e \in E \end{aligned} \tag{4.2.1}$$

Each solution $\boldsymbol{d} = (d_e)_{e \in E}$ is of course associated with a multicut F of G. We will sometimes refer to $\boldsymbol{d}$ as an *integral multicut*. This will help us distinguish

between solutions to IP (4.2.1), as well as the integer program's linear relaxation, which we now define:

For each edge variable d_e, we relax its value to the unit interval $[0, 1]$. As the cost function c is positive, we may further forgo the upper bound on these variables. This yields the following linear relaxation of IP (4.2.1):

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c(e)\, d_e \\ \text{subject to} \quad & \sum_{e \in p_i} d_e \geq 1 \quad \forall i = 1, \ldots, k, \\ & d_e \geq 0 \quad \forall e \in E \end{aligned} \tag{4.2.2}$$

We refer to a solution $\boldsymbol{d} = (d_e)_{e \in E}$ of LP (4.2.2) as a *fractional multicut*. In this sense, the variable d_e specifies the amount that e is included in the solution. As any integral multicut is a fractional multicut by definition, an optimum solution to LP (4.2.2) places a lower bound on an optimum solution to IP (4.2.1). Moreover, we observe that we can solve LP (4.2.2) in polynomial time.

Let us now consider the dual of LP (4.2.2). We shall interpret its significance once the entire program is stated. For each $i = 1, \ldots, k$, we introduce a variable f_i, associated to path p_i.

Each edge yields a constraint to the dual program. In particular, if $e \in E$, then we have that, $\sum_{j : e \in p_j} f_j \leq c(e)$. These are the only constraints present in the dual program.

Finally, the objective function of the dual program is $\sum_{i=1}^{k} f_i$. The problem is known as the *multi-commodity flow problem on trees*. We summarize this program below:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{k} f_i \\ \text{subject to} \quad & \sum_{j:e\in p_j} f_j \leq c(e) \quad \forall e \in E, \\ & f_i \geq 0, \qquad \forall i = 1, \ldots, k \end{aligned} \tag{4.2.3}$$

We interpret a solution to the dual program as specifying a *multi-commodity flow* through T. For each $1 \leq i \leq k$, we route a separate commodity between s_i and t_i, using the path p_i. The variable f_i quantifies the amount of the flow being routed through this path.

The cost function associates a capacity to each edge $e \in E$. The capacity of a given edge restricts the total amount of flow that may pass through it. If the total flow through an edge is equal to its capacity, then we say that the edge is *saturated*. Similarly, if the total flow through an edge exceeds its capacity, then the solution is infeasible, and we say that the edge is *oversaturated*. The objective is then to maximize the total amount of flow routed between the source-sink pairs, without *oversaturating* any edges. As this linear program has a polynomial number of constraints, we can find an optimum solution to this problem in polynomial time.

We also introduce an integral version of this program, in which the variables are restricted to nonnegative integers. This problem is known as the *integral multi-commodity flow problem on trees*.

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{k} f_i \\ \text{subject to} \quad & \sum_{i:e\in p_i} f_i \leq c(e) \\ & f_i \in \mathbb{Z}_{\geq 0}, \quad i = 1, \ldots, k \end{aligned} \tag{4.2.4}$$

We refer to a solution to this problem as a *integral multi-commodity flow*. As any integral multi-commodity flow forms a multi-commodity flow, the maximum value of a solution to LP (4.2.3) places an upper bound on the maximum value of a solution to IP (4.2.4).

We observe that this problem in fact has a polynomial time algorithm, provided we restrict the capacities of the edges. This algorithm is based on dynamic programming techniques, however we do not explore the details of this procedure.

Proposition 4.2.1. *If $T = (V, E)$ is a tree, and $c : E \to \mathbb{Z}_{\geq 0}$ is identically one, then an integral multi-commodity flow through T may be found in polynomial time.*

On the other hand, if we place no restrictions on the capacities of the edges, then this problem is **NP**-hard for trees of height greater than three.

4.3 Primal-Dual Multicut Algorithms for Trees

In this section, we focus on designing algorithms for the multicut problem as well as the integral multi-commodity flow problem. In both cases, we restrict our attention to trees which have nonnegative integer capacities. Rather, if T is a tree, then the cost function c is a map from E into $\mathbb{Z}_{\geq 0}$. As we saw that both of these problems are **NP**-hard for trees with unrestricted capacites, we specifically focus on designing approximation algorithms for these problems.

The approximation algorithm that we consider simutaneously builds an integral multi-commodity flow, as well as a multicut of T. In this sense, we obtain a means of approximating optimum solutions to both the *integral multi-commodity flow problem* as well as the *multicut* problem.

We first introduce some terminology which will be vital to this algorithm. Given a tree $T = (V, E)$, we arbitrarily choose a root ν from the vertex set V. If v is any vertex of T, we refer to its *depth* as the number of edges in the unique path in T from ν to v. If u and v are two vertices of T, we refer to the *lowest common ancestor*, denoted $lca(u, v)$, as the vertex of least depth on the unique path between u and v.

Our algorithm is based on the *primal-dual method*. It begins with a trivial solution $(f_i)_{i=1}^k$ to the dual program, LP (4.2.4), as well as an infeasible solution, $F := \emptyset$, to the multicut problem. Among the source-sink pairs that remain connected in $T \setminus F$, a pair (s_i, t_i) will be chosen whose $lca(s_i, t_i)$ is of maximum depth. The algorithm then routes as much flow as possible between s_i and t_i; later updating the variable f_i to reflect this change. When this routing causes an edge

to become saturated, we then add it to F. This procedure terminates when no source-sink pair remains connected in $T \setminus F$. At this point, we scan over the edges of F in the order opposite to which they were added, removing edges which are not essential to the multicut. The output of the algorithm includes both the multicut F, as well as the integer multi-commodity flow $(f_i)_{i=1}^k$.

Before we formally outline this algorithm, we first specify the subroutine used to route flow between source nodes. Given a multi-commodity flow $(f_i)_{i=1}^k$ through T and a source-sink pair (s_j, t_j) for $1 \leq j \leq k$, we *greedily route flow* between s_j and t_j as follows:

Algorithm 4.3.1 Greedy Routing Algorithm

Suppose $T = (V, E)$ is an undirected tree.
Suppose $c : E \to \mathbb{Z}_{\geq 0}$, and $\{(s_i, t_i)\}_{i=1}^k$ are $k \geq 1$ distinct vertex pairs.
Suppose $(f_i)_{i=1}^k$ is a multi-commodity flow through T, with index $j \in [k]$ specified.

1: Compute the path p_j between source-sink pair (s_j, t_j).
2: Initialize $\varepsilon \leftarrow 0$
3: **for** each edge $e \in p_j$ **do**
4: $\quad \varepsilon \leftarrow \min\{\varepsilon, c(e) - \sum_{i:e \in p_i} f_i\}$
5: Let $f_j \leftarrow f_j + \varepsilon$.
6: Return $(f_i)_{i=1}^k$.

With this subroutine, we now state our main algorithm:

Algorithm 4.3.2 Primal Dual Tree Multicut Algorithm

Suppose $T = (V, E)$ is an undirected tree.
Suppose $c : E \to \mathbb{Z}_{\geq 0}$, and $\{(s_i, t_i)\}_{i=1}^{k}$ are $k \geq 1$ distinct vertex pairs.
1: Initialize $F = \emptyset$, and $f_i \leftarrow 0$ for $i = 1, \ldots, k$.
2: **for** $v \in V$; processed in decreasing order of depth **do**
3: **for** $j = 1, \ldots, k$ **do**
4: **if** $lca(s_i, t_i) = v$ **then**
5: Compute the path p_j, and note its currently saturated edges.
6: Run Algorithm 4.3.1, to route flow between s_j and t_j.
7: Mark the edges of p_j, which have now become saturated.
8: In any order, add these newly marked edges to F.
9: Set $l \leftarrow |F|$.
10: Label the edges of F by $e_1, \ldots, e_l$; the order in which they were added.
11: **for** $i = 0, \ldots, k-1$ **do**
12: **if** $F \setminus e_{k-i}$ is a multicut **then** Set $F \leftarrow F \setminus e_{k-i}$.
13: Return F and $(f_i)_{i=1}^{k}$.

For a fixed vertex v of T, we refer to Steps 2 to 8, as the time in which v is *processed* by the algorithm. These are the steps executed by the outermost loop, when v is held as its parameter.

It is clear that the above algorithm will run in polynomial time. Moreover, the multi-commodity flow $(f_i)_{i=1}^{k}$ is guaranteed to be feasible, as the algorithm never oversaturates an edge. To see that the edge set F is in fact a multicut, we first observe that $(f_i)_{i=1}^{k}$ is maximal. In other words, we cannot increase an f_i variable by an integral amount, without oversaturating an edge. This implies that for each $i = 1, \ldots, k$, there exists a saturated edge present in the path p_i between s_i and t_i. By Step 10 of the algorithm, all of these saturated edges will have been added to F. Thus, at this point, F is a multicut. Moreover, F remains a multicut through the final steps of the algorithm, as only redundant edges are removed.

Now that we have verified Algorithm 4.3.2 is both efficient and correct, we shall prove a guarantee on the optimality of its ouputs. In order to prove this theorem, we first observe an essential lemma:

Lemma 4.3.1. *Let v be a vertex of T. If there is some $1 \leq i \leq k$, such that $lca(s_i, t_i) = v$, then once the algorithm processes v, an edge from p_i will have been added to F.*

Moreover, suppose that we also know that nonzero flow is routed from s_i to t_i. In this case, if e is an edge of p_i and F, then it must have been added when v was processed or in subsequent iterations of the outermost loop.

Proof. We begin by proving the first part of the lemma. Let us consider Step 2 of the algorithm, in which we begin to process v. If an edge from p_i has already been added to F, then the statment holds. Otherwise, no edge of p_i has been added to F at this point. In particular, this implies that none of the edges of p_i have been saturated. It follows that we can greedily route nonzero flow through p_i. Once this is done, at least one edge of p_i must now be saturated. This edge will then be added to F.

For the second part of the lemma, let us assume that an edge e of p_i is added to F before v has been processed. Since this edge was added to F, we know that it must have been saturated at some point. When the algorithm eventually processes v, it will fail to route nonzero flow through p_i, as any amount would oversaturate e. This yields a contraction, so we know that e cannot have been added before v was processed.

□

Proposition 4.3.2. *Let $(f_i)_{i=1}^k$ and F denote the outputs returned by Algorithm 4.3.2 as above. For each $i = 1, \ldots, k$, if $f_i > 0$, then at most two edges of F belong to p_i.*

Proof. Let us fix a source sink pair (s_i, t_i). We shall prove the statement of the theorem for this vertex pair. For simplicity, we may assume that $i = 1$.

We denote v to be $lca(s_1, t_1)$. Moreover, we denote $s_1 - v$ and $t_1 - v$ to be the paths from s_1 to v and t_1 to v, respectively. Our goal is to show that F takes at most one edge from each of $s_1 - v$ and $t_1 - v$. We shall prove this statement for $s_1 - v$. The argument will extend to $t_1 - v$ by symmetry.

Let us suppose that this statment does not hold. Rather, there exists edges e and e' in F, each of which belongs to $s_1 - v$. We observe that since both occur on the same path to v, they cannot be of the same depth from ν. We may assume that e occurs at a lower depth than e'.

Consider Step 12 of Algorithm 4.3.2, when the membership of e is reviewed. Since the algorithm does not remove e from F, there must be some source-sink pair (s_j, t_j), such that e is the only member of p_j in F at this time.

Let us suppose that $u = lca(s_j, t_j)$. We observe that since e' does not lie in p_j, u must have greater depth than v. In particular, this means that u is processed before v. Moreover, by Lemma 4.3.1, when u is processed, an edge e'' is added to F.

On the other hand, we know that $f_1 > 0$. By Lemma 4.3.1, this implies that the earliest time at which the edge e could have been added to F was when v was processed. As v was processed after u, this means that e'' was added to F before

e. Thus, when e was reviewed by the algorithm, e'' was also present in F. As e was supposed to be the only edge of p_j present at this time, this yields a contradiction. The result thus holds.

□

We are now ready to state and prove the approximation guarantees of our algorithm. As the algorithm simutaneously builds solutions to two optimization problems, we include both guarantees in the following theorem:

Theorem 4.3.3. *Algorithm 4.3.2 achieves an approximation of* 2 *for the multicut problem, and* 1/2 *for the multi-commodity flow problem.*

Proof. Let $\boldsymbol{f} = (f_i)_{i=1}^k$ and F be the outputs of the algorithm. We introduce the variables, $\boldsymbol{d} = (d_e)_{e \in E}$, where $d_e = 1$ if and only if $e \in F$. Our goal will be to show that approximate complementary slackness conditions hold for $\boldsymbol{f}$ and $\boldsymbol{d}$ (see the discussion before Proposition 1.3.9 in Section 1.3).

First, consider the solution $\boldsymbol{d}$. If $e \in E$ has $d_e = 1$, then by definition, it is included in F. On the other hand, if e is placed in F, then Algorithm 4.3.2 must have saturated this edge. In other words, $\sum_{i:e \in p_1} f_i = c(e)$. This tells us that the primal complementary slackness conditions are satisfied.

Now consider the solution $\boldsymbol{f}$. For each $i = 1, \ldots, k$, we know that if $f_i > 0$, then at most two edges of p_i are in F, by Proposition 4.3.2. Moreover, F is a valid multicut, so at least one edge of p_i is in F. In terms of the solution $\boldsymbol{d}$, this means that,

$$1 \leq \sum_{e \in p_i} d_e \leq 2$$

for each $i = 1, \ldots, k$. This implies that the dual complementary slackness conditions are also satisfied. The approximation guarantees therefore follow by Proposition 1.3.9 from Section 1.3.

□

4.4 Multicut on General Graphs

In this section, we once again consider the multicut problem. However, unlike before, we do not restrict our attention to dealing specifically with trees. Instead, we design an algorithm for generating multicuts that can be applied to any graph. As finding optimum multicuts is **NP**-hard, we instead build an algorithm with a good approximation guarantee. This algorithm uses the same techniques as seen in Section 3.2. As this is the case, many of the initial claims are slight generalizations of propositions which were seen earlier, and so can be skimmed over quickly. That being said, the later analysis is more sophisticated, and yields an algorithm with a better approximation guarantee than we'd achieve from exactly replicating the work done earlier.

Let $G = (V, E)$ be an undirected graph, together with a cost function $c : E \to \mathbb{Q}_{\geq 0}$. Moreover, let $\{(s_i, t_i)\}_{1 \leq i \leq k}$ be $k \geq 1$ distinct source-sink pairs of G. We wish to find a multicut $F \subseteq E$ of close minimal cost, with respect to cost function c.

It will be useful to establish an integer program which encodes this problem. For each $i = 1, \ldots, k$, we denote $\mathcal{P}_i$ as the set of simple paths from s_i to t_i in G. Moreover, we denote $\mathcal{P}$ as the union of $\mathcal{P}_1, \ldots, \mathcal{P}_k$. For each edge $e \in E$, we introduce a $\{0, 1\}$ variable d_e, which is nonzero if and only if e is placed in the set F. In order to ensure that F is a valid multicut, we must have $\sum_{e \in p} d_e \geq 1$, for all $p \in \mathcal{P}$. Clearly, the objective function $\sum_{e \in E} c(e) d_e$ encodes the proper cost of F. To summarize, we have the following integer program:

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c(e)\, d_e \\ \text{subject to} \quad & \sum_{e \in p} d_e \geq 1 \qquad \forall p \in \mathcal{P} \\ & d_e \in \{0,1\} \quad \forall e \in E \end{aligned} \tag{4.4.1}$$

This formulation explicitly generalizes IP (4.2.1), which was introduced in the previous section, as well as IP (3.2.1) from Section 3.2. It will once again be useful to obtain a linear relaxation of this integer program. For each $e \in E$, we relax variable d_e to the unit interval $[0, 1]$. As the cost function $c : E \to \mathbb{Z}_{\geq 0}$ is nonnegative, we can forgo these upper bounds, yielding the linear program below. We refer to this program as the *fractional multicut problem*:

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c(e)\, d_e \\ \text{subject to} \quad & \sum_{e \in p} d_e \geq 1 \quad \forall p \in \mathcal{P} \\ & d_e \geq 0 \quad \forall e \in E \end{aligned} \tag{4.4.2}$$

Given a solution $\boldsymbol{d} = (d_e)_{e \in E}$ to this problem, we refer to it as a *fractional multicut*. The quantity d_e may be thought of as representing the fractional extent to which edge e is included in the multicut. While this linear program may have exponentially many constraints, we shall see that it can still be solved in polynomial time.

Before we prove this claim, we recall a means of measuring paths in the graph G, which was introduced in Section 3.2. Let $\boldsymbol{d} = (d_e)_{e \in E}$ be an arbitrary edge assignment, where $d_e \in \mathbb{Q}_{\geq 0}$ for all $e \in E$ ($\boldsymbol{d}$ may or may not be a valid solution to

LP (4.4.2)). If p is a path in G, then we refer to the *length* of p *with respect to* $\boldsymbol{d}$ as the quantity $\sum_{e \in p} d_e$.

Proposition 4.4.1. *An optimum solution to LP* (4.4.2) *may be found in polynomial time in the size of* $G = (V, E)$ *and* $c : E \to \mathbb{Q}_{\geq 0}$.

Proof. We observe that a polynomial time separation oracle exists for this linear program. To see this, let us suppose that $\boldsymbol{d} = (d_e)_{e \in E}$ is a potential solution to LP (4.4.2). For each $i = 1, \ldots, k$, let us compute a shortest path p_i from s_i to t_i, with respect to edge weight $\boldsymbol{d}$ (we measure the length of a path as the sum of its $\boldsymbol{d}$ edge weights). If there exists some $1 \leq i' \leq k$, such that,

$$\sum_{e \in p_{i'}} d_e < 1$$

then we mark $\boldsymbol{d}$ as infeasible, and return this violated constraint.

On the other hand, if for each $i = 1, \ldots, k$, we have that,

$$1 \leq \sum_{e \in p_i} d_e,$$

then we know that $\boldsymbol{d}$ is a feasible solution to the linear program.

To see this, observe that if $p \in \mathcal{P}$ is a path from s_{i^*} to t_{i^*} for some $1 \leq i^* \leq k$, then we know that,

$$1 \leq \sum_{e \in p_{i^*}} d_e \leq \sum_{e \in p} p$$

as p_{i^*} is a shortest path from s_{i^*} to t_{i^*} with respect to edge weight $\boldsymbol{d}$.

In this scenario, our separation oracle simply validates the solution $\boldsymbol{d}$ as feasible. Clearly, it is possible to implement this algorithm in polynomial time.

Using this oracle, we may appeal to Theorem 1.3.5 from Subsection 1.3.1 to provide us with a polynomial time algorithm for LP (4.4.2).

□

In the preceding proof, it was convenient to use the potential solution $\boldsymbol{d}$ of LP (4.4.2) to measure path lengths in G. We can extend this notion to vertices of G as well. Rather, if u and v are two vertices of G, then the *distance* between u and v *with respect to* $\boldsymbol{d}$, is the length of the shortest path between them. We denote this quantity by $dist_{G,\boldsymbol{d}}(u, v)$, and simply $dist_{\boldsymbol{d}}(u, v)$ when the context is clear.

If $\boldsymbol{d}$ is an optimum solution to LP (4.4.2), then it satisfies the *triangle inequality* for any selection of edges. Rather, if u, v and w are vertices of G, then we have that,

$$d_{u,v} \leq d_{u,w} + d_{w,v},$$

where $\{u, v\}, \{u, w\}$ and $\{w, v\}$ are edges of G. Before we prove this claim, we remark that it is a generalization of Proposition 3.2.3 from Section 3.2. Moreover, the proof of the below proposition is effectively the same as the proof of Proposition 3.2.3, but we include it again for convenience.

Proposition 4.4.2. *If $\boldsymbol{d}$ is an optimum solution to LP* (4.4.2)*, then it satisfies the triangle inquality for any selection of edges of G.*

Proof. We first observe that $d_e \geq 0$ for each edge $e \in E$, as $\boldsymbol{d}$ is a valid solution to LP (4.4.2).

It remains to check that $\boldsymbol{d}$ satisfies the triangle inequality on E. Suppose that this is not the case. That is, there exist vertices u, v and w with edges between

them, such that,

$$d_{u,v} > d_{u,w} + d_{w,v} \tag{4.4.3}$$

Moreover, observe that since $\boldsymbol{d}$ is an optimum solution to LP (4.4.2), $d_{u,v}$ cannot be decreased without violating a constraint of the linear program.

Thus, there is some source-sink pair, say (s_i, t_i) for $1 \leq i \leq k$, with a simple path p between them, such that

$$\sum_{e \in p} d_e = 1,$$

where the edge $\{u, v\}$ is guaranteed to lie in p.

Observe that if we remove $\{u, v\}$ from p, and replace it with $\{u, w\}$ followed by $\{w, v\}$, then we are left with a new path p' between s_i and t_i. We may also build a path p'', whose edges are contained in p', and which is guaranteed to be simple.

As a consequence of Equation 4.4.3, we know that the length of p' is strictly less than the length of p. Moreover, as the edges of p'' are contained in p', we know that the length of p'' is also less than the length of p.

However, since p has length one, p'' must have length less than one. As p'' is a simple path between nodes s_i and t_i, this means that $\boldsymbol{d}$ is not a valid solution to LP (4.4.2). This is a contradiction, so the violation in Equation 4.4.3 cannot occur. It follows that $\boldsymbol{d}$ satisfies the triangle inquality everywhere, and so the claim holds.

□

If $\boldsymbol{d}$ is an optimum solution to LP (4.4.2), then we can view $dist_{\boldsymbol{d}}(\cdot, \cdot)$ as an extension of $\boldsymbol{d}$.

Proposition 4.4.3. *Let $\boldsymbol{d}$ be a nonnegative edge assignment of G. The function $dist_{\boldsymbol{d}} : V^2 \to \mathbb{Q}_{\geq 0}$, forms a metric on the vertices of G. Moreover, if $\boldsymbol{d}$ is an optimum solution to LP* (4.4.2)*, then,*

$$d_{u,v} = dist_{\boldsymbol{d}}(u, v)$$

for each each $\{u, v\} \in E$.

Given an optimum solution $\boldsymbol{d}$ to LP (4.4.2), we may devise a *region growing process* for computing multicuts. We begin by describing a procedure that is a natural generalization of the techniques used in Algorithm 3.2.3 of Section 3.2. In particular, a *radius* $0 \leq \rho < 1$ is chosen uniformly at random, which we then use to build the *regions*, $B^{\boldsymbol{d}}_{\rho}(s_1), \ldots B^{\boldsymbol{d}}_{\rho}(s_k)$. Unlike the regions built in Algorithm 3.2.3, there is no guarantee that these balls will be disjoint. This is because the multicut problem does not impose restrictions on how close source nodes can be.

Once these regions are built, we collect the edges which are separated by each source node. Rather, for $i = 1, \ldots, k$, we set $F_i := \delta(B^{\boldsymbol{d}}_{\rho}(s_i))$. If $e = \{u, v\}$ is an edge of G, let us assume that s_i is closer to u than v. In this case, e will be placed in F_i if and only if $\rho \in [dist_{\boldsymbol{d}}(s_i, u), dist_{\boldsymbol{d}}(s_i, v))$. As ρ is chosen uniformly at random from the interval $[0, 1)$, it follows that for $i = 1, \ldots, k$,

$$\mathbb{P}(e \in F_i) \leq dist_{\boldsymbol{d}}(s_i, v) - dist_{\boldsymbol{d}}(s_i, u) \tag{4.4.4}$$

$$= d_e, \tag{4.4.5}$$

where the second equality follows from Proposition 4.4.3.

On the other hand, the edge set $F := \cup_{i=1}^{k} F_i$ is clearly a multicut, as it separates all the source-sink pairs. Rather, for $i = 1, \ldots, k$, $t_i \notin B_{\rho}^{\boldsymbol{d}}(s_i)$ as $dist_{\boldsymbol{d}}(s_i, t_i) \geq 1$, and $r < 1$. Using the above equations,

$$\mathbb{P}(e \in F) \leq k\, d_e$$

by the the union bound. Thus, if we consider the random variable $c(F)$, we have that,

$$\begin{aligned}
\mathbb{E}\, c(F) &= \sum_{e \in E} c(e)\, \mathbb{P}(e \in F) \\
&\leq k \sum_{e \in E} c(e)\, d_e \\
&= k\, OPT_f \\
&\leq k\, OPT,
\end{aligned}$$

where OPT_f is the value of an optimum fractional multicut, and OPT is the value of an optimum integral multicut.

We remark that as in the case of Algorithm 3.2.3 of Section 3.2, there exists a derandomization of this algorithm (see Algorithm 3.2.2). By the above discussion, we arrive at the following theorem:

Theorem 4.4.4. *There exists an algorithm for Problem 4.1.1 which achieves an approximation guarantee of k.*

While the above algorithm is reasonably effective for small values of k, it yields very poor approximation guarantees if k is proportional to the size of G. As we are often in situations were we would like to separate large numbers of

source-sink pairs, this algorithm alone will not suffice. To rectify this issue, we introduce another algorithm which is based upon similar techniques, yet achieves a much better approximation guarantee.

Our first goal will be to modify the above algorithm, such that it is guaranteed to build disjoint regions. This will be accomplished by building our regions sequentially, processing the source nodes in the order in which they are indexed. When a region is built, we remove its vertices from the graph G. This ensures that when the next source node is processed, its region will be disjoint from its predecessors. Let us now formally describe this process.

It will be convenient to refer to G as G_1, and the edge assignment $\boldsymbol{d}$ by $\boldsymbol{d}^1$. We next choose k radii $r_1, \ldots, r_k \in [0, \frac{1}{2})$, and then build the region, $U_1 := B^{\boldsymbol{d}^1}_{r_1}(s_1)$ (we shall see later why it is necessary to restrict the range of the radii). In order to build the next region, we first remove the vertices of U_1 from G_1, leaving us with subgraph G_2. This will of course remove a number of edges from G_1, so we denote $\boldsymbol{d}^2$ as $\boldsymbol{d}^1$ restricted to the edges of G_2. If source node s_2 is not present in G_2, then we set $U_2 := \emptyset$. Otherwise, we set $U_2 := B^{\boldsymbol{d}^2}_{r_2}(s_2)$, and in each case remove U_2 from G_2. In general, if $1 \leq i \leq k$, then suppose we currently have graph G_i with edge weight $\boldsymbol{d}^i$. If source node s_i is not present in G_i, then we set $U_i := \emptyset$. If the source node is present, then we set $U_i := B^{\boldsymbol{d}^i}_{r_i}(s_i)$. If $i+1 > k$, then the process terminates, and we are left with regions $U_1, \ldots, U_k$. Otherwise, the process continues, and we form the graph $G_{i+1} := G_i \setminus U_i$. The edge weight $\boldsymbol{d}^i$ is then restricted to the edges of G_{i+1}, yielding the new edge weight $\boldsymbol{d}^{i+1}$.

Once the regions $U_1, \ldots, U_k$ have been formed, we collect the edges that lie between them. Namely, for $i = 1, \ldots, k$, we set $F_i := \delta_{G_i}(U_i)$. In other words, F_i contains the edges of G_i with exactly one vertex in U_i. We then collect the edge set $F := \cup_{i=1}^{k} F_i$, and return F as the output of the algorithm.

Algorithm 4.4.1 Disjoint Region Growing Algorithm

Let $G = (V, E)$ be an undirected graph.
Let $c : E \to \mathbb{Q}_{\geq 0}$ be a cost function.
Let $\{(s_i, t_i)\}_{i=1}^{k}$ be a collection of $k \geq 1$ source-sink pairs of G.
Let $\boldsymbol{r} = (r_1, \ldots, r_k) \in [0, \frac{1}{2})^k$.
Compute an optimum solution $\boldsymbol{d}$ to LP (4.4.2).
Initialize $G_1 \leftarrow G$, and $\boldsymbol{d}^1 \leftarrow \boldsymbol{d}$.
for $i = 1, \ldots, k$ **do**
    Initialize $U_i \subseteq G_i$.
    if $s_i \in G_i$ **then**
        Set $U_i \leftarrow B_{r_i}^{\boldsymbol{d}^i}(s_i)$.
    else
        Set $U_i \leftarrow \emptyset$.
    Initialize $F_i \leftarrow \delta_{G_i}(U_i)$.
    if $i < k$ **then**
        Initialize graph G_{i+1}, and edge assignment $\boldsymbol{d}^{i+1}$.
        Set $G_{i+1} \leftarrow G_i \setminus U_i$.
        Set $\boldsymbol{d}^{i+1}$ as $\boldsymbol{d}^i$ restricted to $e(G_{i+1})$.
Initialize $F := \cup_{i=1}^{k} F_i$.
Return F.

While the algorithm stores the collection of graphs $(G_i)_{i=1}^{k}$ and their respective edge assignments $(\boldsymbol{d}^i)_{i=1}^{k}$, this is simply to clarify how the algorithm executes. It is not necessary in practise. In any case, since we can solve LP (4.4.2) efficiently, it is clear that the algorithm will execute in polynomial time.

Proposition 4.4.5. *Algorithm 4.4.1 returns a valid multicut in polynomial time, provided it is passed radii from the interval* $[0, \frac{1}{2})$.

Proof. As in the Algorithm 4.4.1, let us denote the regions formed by $U_1, \ldots, U_k$. Moreover, let us consider source-sink pair (s, t). Suppose that there exists some $1 \leq i \leq k$, for which both s and t reside in R_i. In this case, we know that by definition, $s, t \in B_{r_i}^{\boldsymbol{d}^i}(s_i)$. This means that if we consider the distance function $dist_{\boldsymbol{d}^i}(\cdot, \cdot)$,

$$dist_{\boldsymbol{d}^i}(s_i, s) \leq r_i < \frac{1}{2}.$$

Similarly,

$$dist_{\boldsymbol{d}^i}(s_i, t) \leq r_i < \frac{1}{2}.$$

On the other hand, $dist_{\boldsymbol{d}^i}(\cdot, \cdot)$ forms a metric on ther vertices of G_i, as it is derived from the edge assignment $\boldsymbol{d}^i$ (see Proposition 4.4.3). In particular, this implies that.

$$dist_{\boldsymbol{d}^i}(s, t) < 1,$$

by the triangle inequality.

If we now consider the original edge assignment $\boldsymbol{d}$ of G, then since $G_i \subseteq G$, we know that,

$$dist_{\boldsymbol{d}}(s, t) \leq dist_{\boldsymbol{d}^i}(s, t),$$

as $\boldsymbol{d}^i$ retains the same edge values as $\boldsymbol{d}$. As a consequence, it follows that,

$$dist_{\boldsymbol{d}}(s, t) < 1.$$

Since $\boldsymbol{d}$ was assumed to be a valid solution to LP (4.4.2), this is a contradiction. It follows that for each source-sink pair (s, t) and $1 \leq i \leq k$, at most one of s and t is contained in U_i. In other words, there is no region which contains both a

source and a sink. This implies that the edge set F will correctly separate all the relevant vertex pairs.

□

Now that we have proven the correctness and efficiency of Algorithm 4.4.1, we shall derive a variation of it that achieves a desirable performance guarantee. In order to do this, it is convenient to introduce a *weight function* w which assigns a value to each region computed in the above algorithm. Let us once again assume that $\boldsymbol{d}$ is an optimum solution to LP (4.4.2). We denote β as the cost of this solution. Rather,

$$\beta = \sum_{e \in E} c(e)\, d_e.$$

We then assign $w(s_i) := \beta/k$ for $i = 1, \ldots, k$. These are the only vertices of G which are given nonzero weight. In order to extend w to regions, we first add a weight to each edge $e \in E$, where $w(e)$ is set to $c_e\, d_e$. The weight of a region is roughly defined as the weight of its source node, together with the weight of all edges with at least one end in the region. We refine this definition slightly, where given region U_i, edges with exactly one end in U_i are only given a proportion of their total weight. Rather,

$$w(U_i) := w(s_i) + \sum_{\substack{e=\{u,v\} \in G_i: \\ u,v \in U_i}} c(e)\, d_e + \sum_{\substack{e=\{u,v\} \in G_i: \\ u \in U_i, v \notin U_i}} c(e)\, (r_i - dist_{\boldsymbol{d}^i}(s_i, u)) \qquad (4.4.6)$$

for $i = 1, \ldots, k$, where $G_i = (V_i, E_i)$ is the graph constructed after $i - 1$ iterations of the "for loop" of Algorithm 4.4.1, and $\boldsymbol{d}^i$ is the restriction of $\boldsymbol{d}$ to E_i. If $U_i = \emptyset$,

then we set this value to zero. We refer to $w(U_i)$ as the *weight* of region U_i. Similarly, if $F_i := \delta(U_i)$ as in Algorithm 4.4.1, then we set

$$c(F_i) := \sum_{e \in F_i} c(e).$$

If $F_i = \emptyset$, then this value is set to zero. We refer to $c(F_i)$ as the *cost* of F_i, for $i = 1, \ldots, k$.

Let us suppose that we are allowed the vary the radii of Algorithm 4.4.1, whereas the input graph, cost function and source-sink pairs must remain fixed. In this case, the region U_i is built from G_i, and thus depends on the radii $r_1, \ldots, r_i$, where $1 \leq i \leq k$. As a consequence, we replace U_i with $U_i^{\boldsymbol{r}}$ to indicate that U_i depends on $\boldsymbol{r}$. Similarly, F_i is replaced with $F_i^{\boldsymbol{r}}$ for $i = 1, \ldots, k$.

Our goal will be to show that if $\varepsilon := 2 \ln(k+1)$, then there exists $\boldsymbol{r}^* \in [0, \frac{1}{2})^k$, such that,

$$c(F_i^{\boldsymbol{r}^*}) \leq \varepsilon \, w(U_i^{\boldsymbol{r}^*})$$

for each $i = 1, \ldots, k$. Let us assume for now that such a vector $\boldsymbol{r}^*$ exists. In this case, let $F^{\boldsymbol{r}^*}$ be the multicut returned by executing Algorithm 4.4.1 with $\boldsymbol{r}^*$ as input. We observe that the cuts $F_1^{\boldsymbol{r}^*}, \ldots, F_k^{\boldsymbol{r}^*}$ will be disjoint, as the regions themselves are disjoint by construction. In particular,

$$\begin{aligned} c(F^{\boldsymbol{r}^*}) &= \sum_{i=1}^{k} c(F_i^{\boldsymbol{r}^*}) \\ &\leq \sum_{i=1}^{k} \varepsilon \, w(U_i^{\boldsymbol{r}^*}), \end{aligned}$$

where the second line follows by the assumption involving $\boldsymbol{r}^*$ in the above inequality. On the other hand, if we set $\beta_i^{\boldsymbol{r}} := \sum_{\substack{e \in E_i: \\ e \cap U_i^{\boldsymbol{r}^*} \neq \emptyset}} c(e)\, d_e$, then

$$w(U_i^{\boldsymbol{r}^*}) - w(s_i) \leq \beta_i^{\boldsymbol{r}},$$

as a consequence of Equation 4.4.6, for $i = 1, \ldots, k$. Thus, we have that,

$$c(F^{\boldsymbol{r}^*}) \leq \varepsilon \sum_{i=1}^{k} (\beta_i^{\boldsymbol{r}} + w(s_i)) = \varepsilon\, (\beta + \sum_{i=1}^{k} \beta_i^{\boldsymbol{r}}), \tag{4.4.7}$$

as each source node takes on weight β/k.

Let us now suppose that $e = \{u, v\}$ is an edge of G. Since $U_1^{\boldsymbol{r}^*}, \ldots, U_k^{\boldsymbol{r}^*}$ are disjoint, the term $c(e)\, d_e$ will occur in at most one of $\beta_1^{\boldsymbol{r}}, \ldots, \beta_k^{\boldsymbol{r}}$. It follows that,

$$\sum_{i=1}^{k} \beta_i^{\boldsymbol{r}} = \sum_{i=1}^{k} \sum_{\substack{e \in E_i: \\ e \cap U_i^{\boldsymbol{r}^*} \neq \emptyset}} c(e)\, d_e \leq \sum_{e \in E} c(e)\, d_e = \beta.$$

Combining this observation with Equation 4.4.7, we have that,

$$c(F^{\boldsymbol{r}^*}) \leq 2\, \varepsilon\, \beta = 4 \ln(k+1)\, OPT_f,$$

where OPT_f is equal to the cost of the optimum solution $\boldsymbol{d}$ of LP (4.4.2). We summarize these observations in the following proposition:

Proposition 4.4.6. *If $\boldsymbol{r}^* \in [0, \frac{1}{2})^k$ is such that,*

$$c(F_i^{\boldsymbol{r}^*}) \leq 2\, \ln(k+1)\, w(U_i^{\boldsymbol{r}^*})$$

for $i = 1, \ldots, k$, then the multicut $F^{\boldsymbol{r}^*}$ will have cost,

$$c(F^{\boldsymbol{r}^*}) \leq 4\, \ln(k+1)\, OPT_f,$$

where OPT_f is the value of an optimum solution to LP (4.4.2).

In light of this proposition, Algorithm 4.4.1 will yield an approximation guarantee of $4\ln(k+1)$, provided we devise a means to compute $\boldsymbol{r}^*$. It turns out that it is possible to find a collection of radii which satisfy these properties in polynomial time. Our next goal will be to prove their existence, and outline how this computation can be done.

Proposition 4.4.7. *We may compute a collection of radii, $\boldsymbol{r}^* \in [0, \frac{1}{2})^k$, satisfying the inequality,*

$$c(F_i^{\boldsymbol{r}^*}) \leq \varepsilon w(U_i^{\boldsymbol{r}^*})$$

for $i = 1, \ldots, k$. Moreover, this computation can be done in polynomial time.

Before we prove this proposition, it will convenient to state a simple lemma from calculus. We omit the proof as it is elementary.

Lemma 4.4.8 (Generalized Fundamental Theorem of Calculus)**.** *Let $a < b$ be real numbers and $h, g : [a, b] \to \mathbb{R}$. Assume that h is integrable, and g is continuous. Moreover, assume that there are $x_1 < \ldots < x_l$ in $[a, b]$, where for $x \in (a, b)$, g is differentiable at x and*

$$g'(x) = h(x),$$

provided $x \neq x_i$, for $1 \leq i \leq l$. In this case,

$$\int_a^b h(x)\,dx = g(b) - g(a).$$

We are now ready to prove Proposition 4.4.7

Proof. We outline a way to efficiently compute $r_1^*, \ldots, r_k^*$, which satisfy the inequality of Proposition 4.4.7. This computation is done dynamically as Algorithm 4.4.1 is executed, so we reuse much of the terminology developed earlier.

Let us begin by describing how to find the radius r_1^*. We recall that $G_1 = (V_1, E_1)$ denotes the graph G, before the first region about s_1 is grown. Moreover, the edge weight $\boldsymbol{d}^1$ denotes the edge weight $\boldsymbol{d}$, before any edges of G are removed.

For each edge $e = \{u, v\} \in E_1$, we define the interval $I_e^1 := [dist_{\boldsymbol{d}^1}(s_1, u), dist_{\boldsymbol{d}^1}(s_1, v))$, where $dist_{\boldsymbol{d}^1}(s_1, u) \leq dist_{\boldsymbol{d}^1}(s_1, v)$. We denote the indicator function of the interval I_e^1 by $\chi_{I_e^1}$. Let us suppose $\widetilde{c}_1 : [0, 1/2) \to \mathbb{R}$, for which

$$\widetilde{c}_1 := \sum_{e \in E_1} c(e)\, \chi_{I_e^1}.$$

If $\boldsymbol{r} \in [0, 1/2)^k$, then suppose Algorithm 4.4.1 returns edge set $F_1^{\boldsymbol{r}}$ when passed radii $\boldsymbol{r}$. In this case, we have that,

$$c(F_1^{\boldsymbol{r}}) = \sum_{e \in E} c(e)\, \chi_{I_e^1}(r_1),$$

where $c(F_1^{\boldsymbol{r}})$ denotes the cost of $F_1^{\boldsymbol{r}}$. Thus, we have that,

$$\widetilde{c}_1(r_1) = c(F_1^{\boldsymbol{r}}).$$

for all $\boldsymbol{r} = (r_1, \ldots, r_k) \in [0, 1/2)^k$. In other words, $\widetilde{c}_1(r_1)$ describes the cost of the edge set $F_1^{\boldsymbol{r}}$. We shall now build a similar function to describe the weight of the region $U_1^{\boldsymbol{r}}$.

For each edge $e = \{u, v\} \in E_1$, let us define $\phi_1^e : [0, 1/2) \to \mathbb{R}$, where for $r_1 \in [0, 1/2)$,

$$\phi_1^e(r_1) = \begin{cases} 0 & \text{if } r_1 < dist_{_1}(s_1, u) \\ r_1 - dist_{\boldsymbol{d}^1}(s_1, u) & \text{if } dist_{\boldsymbol{d}^1}(s_1, u) \leq r_1 < dist_{\boldsymbol{d}^1}(s_1, v) \\ d_{u,v} & \text{if } dist_{\boldsymbol{d}^1}(s_1, v) \leq r_1, \end{cases}$$

provided we assume $dist_{\boldsymbol{d}^1}(s_1, u) \leq dist_{\boldsymbol{d}^1}(s_1, v)$. Moreover, we define $\widetilde{w}_1 : [0, 1/2) \to \mathbb{R}$, where

$$\widetilde{w}_1 := w(s_1) + \sum_{e \in E_1} c(e)\, \phi_e^1.$$

Recall that if $\boldsymbol{r} = (r_1, \ldots, r_k) \in [0, 1/2)^k$, then the weight of $U_1^{\boldsymbol{r}}$, denoted $w(U_1^{\boldsymbol{r}})$, is defined as

$$w(U_1^{\boldsymbol{r}}) := w(s_1) + \sum_{\substack{e=\{u,v\} \in G_1: \\ u,v \in U_1^{\boldsymbol{r}}}} c(e)\, d_e + \sum_{\substack{e=\{u,v\} \in G_1: \\ u \in U_1^{\boldsymbol{r}}, v \notin U_1^{\boldsymbol{r}}}} c(e)\, (r_1 - dist_{\boldsymbol{d}^1}(s_1, u)),$$

where the region $U_1^{\boldsymbol{r}}$ is returned by executing Algorithm 4.4.1 with radii $\boldsymbol{r}$. Clearly, we have that,

$$\widetilde{w}_1(r_1) := w(U_1^{\boldsymbol{r}}),$$

for all $\boldsymbol{r} = (r_1, \ldots, r_k) \in [0, 1/2)^k$.

We observe that w_1 and c_1 have a number of desirable properties.

- $\widetilde{w}_1$ is continuous on $[0, 1/2)$.
- For each $r_1 \in (0, 1/2)$, if $\widetilde{w}_1$ is differentiable at r_1, then

$$\widetilde{w}_1'(r_1) = \widetilde{c}_1(r_1).$$

- $\widetilde{w}_1$ is differentiable at all but finitely many points of $[0, 1/2)$.
- $\widetilde{c}_1$ is integrable.

We now define functions $h_1 : [0, 1/2) \to \mathbb{R}$ and $g_1 : [0, 1/2) \to \mathbb{R}$, where $h_1(r_1) := \widetilde{c}_1(r)/\widetilde{w}_1(r_1)$, and $g_1(r_1) := \ln w_1(r_1)$ for each $r_1 \in [0, 1/2)$.

We observe that both g_1 and h_1 are well defined, as $\widetilde{w}_1 \geq wt(s_1) = \beta/k > 0$, where $\beta := \sum_{e \in E} c(e)\, d_e$. Moreover, we observe the following facts regarding these functions, as a consequence of the properties of $\widetilde{w}_1$ and $\widetilde{c}_1$:

- g_1 is continuous on $[0, 1/2)$
- For all but finitely many points of $(0, 1/2)$, g_1 is differentiable, and

$$g_1'(r_1) = h_1(r_1)$$

  where $r_1 \in (0, 1/2)$.
- h_1 is integrable

As a consequence of Lemma 4.4.8 and the above properties, we have that,

$$\begin{aligned}\int_0^{1/2} \frac{\widetilde{c}_1(r_1)}{\widetilde{w}_1(r_1)}\, dr_1 &= \int_0^{1/2} h_1(r_1)\, dr_1 \\ &= g_1(1/2) - g_1(0) \\ &= \ln(\widetilde{w}_1(1/2)) - \ln(\widetilde{w}_1(0))\end{aligned}$$

On the other hand, we know that $w_1(0) = \beta/k$, and $w_1(1/2) \leq \beta + \beta/k$, as the weight of a region based about s_1 cannot exceed β. It follows that,

$$\int_0^{1/2} \frac{\widetilde{c}_1(r_1)}{\widetilde{w}_1(r_1)} dr_1 \leq \ln(k+1).$$

In particular, if $M := \inf_{0 \leq r_1' < 1/2} \frac{\widetilde{c}_1(r_1')}{\widetilde{w}_1(r_1')}$, then

$$\int_0^{1/2} M \, dr_1 \leq \int_0^{1/2} \frac{\widetilde{c}_1(r_1)}{\widetilde{w}_1(r_1)} dr_1 \leq \ln(k+1).$$

Thus,

$$\inf_{0 \leq r_1' < 1/2} \frac{\widetilde{c}_1(r_1')}{\widetilde{w}_1(r_1')} \leq 2 \, \ln(k+1).$$

It is possible to efficiently compute a radius which matches this infimum. Observe that since $\widetilde{c}_1 = \sum_{e \in E} c(e) \, \chi_{I_e^1}$, and $\widetilde{w}_1 = \sum_{e \in E} c(e) \, \phi_1^e$, it is sufficient to evalutate $\widetilde{c}_1 / \widetilde{w}_1$ on the endpoints of I_1^e for each $e \in E$. Moreover, the endpoints of an edge $\{u, v\} \in E_1$ are precisely $dist_{\boldsymbol{d}^1}(s_1, u)$ and $dist_{\boldsymbol{d}^1}(s_1, v)$, so these can be computed in polynomial time.

As a result of these observations, we may compute $0 \leq r_1^* < 1/2$ in polynomial time, such that

$$\widetilde{c}_1(r_1^*) \leq 2 \, \ln(k+1) \, \widetilde{w}_1(r_1^*).$$

Once we compute this radius r_1^*, we may grow the first region $B_{r_1^*}(s_1)$, and then remove it from G_1. This leaves us with a new graph G_2 and a restricted edge assignment $\boldsymbol{d}^2$.

In order to continue this process, we may suppose that $1 \leq i \leq k$, and that we have already found the radii $r_1^*, \ldots, r_{i-1}^*$. We may assume at this point that the graphs $G_1, \ldots, G_i$ and edge assignments $\boldsymbol{d}^1, \ldots, \boldsymbol{d}^i$ have been computed using these radii as parameters to Algorithm 4.4.1.

We now generalize the functions $\widetilde{w}_1$ and $\widetilde{c}_1$, by defining $\widetilde{w}_i, \widetilde{c}_i : [0, 1/2) \to \mathbb{R}$. If $r_i \in [0, 1/2)$, then we may define $\widetilde{w}_i(r_i)$ to be the weight of the region $B_{r_i}(s_i)$, and $\widetilde{w}_i(r_i)$ as the cost of the edge set $\delta(B_{r_i}(s_i))$. Using an identical analysis as before,

we arrive at the conclusion that,

$$\inf_{0 \leq r_i' < 1/2} \frac{\widetilde{c}_i(r_i')}{\widetilde{w}_i(r_1')} \leq 2 \ln(k+1).$$

Moreover, we can once again evaluate $\widetilde{w}_i$ and $\widetilde{c}_i$ at the points $dist_{\boldsymbol{d}^i}(s_i, u)$ and $dist_{\boldsymbol{d}^i}(s_i, v)$ for each $\{u, v\} \in E_i$. This will be sufficient to find a radius r_i^*, witnessing the above infimum. Rather,

$$\widetilde{c}_i(r_i^*) \leq 2 \ln(k+1)\, \widetilde{w}_i(r_i^*),$$

where r_i^* is computed in polynomial time. At this point, the region $B_{r_i^*}(s_i)$ is grown about s_i, and if $i+1 \leq k$, then we update G_{i+1} by removing this region from G_i. The edge assignment $\boldsymbol{d}^i$ is then restricted to the edges of G_{i+1}, yielding the assignment $\boldsymbol{d}^{i+1}$.

Once the growing process finishes, we are left with regions, $B_{r_1^*}(s_1), \ldots, B_{r_k^*}(s_k)$ and edge sets, $\delta(B_{r_1^*}(s_1)), \ldots, \delta(B_{r_k^*}(s_k))$. It is clear that,

$$c(\delta(B_{r_i^*}(s_i))) \leq 2 \ln(k+1) w(B_{r_i^*}(s_i))$$

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

These are precisely the regions and edge sets formed by passing $\boldsymbol{r}^* := (r_1^*, \ldots, r_k^*)$, to Algorithm 4.4.1, so the claim holds.

□

Let us now summarize an algorithm which makes use of our work so far. We first compute a selection of radii which satisfy the above proposition, run

Algorithm 4.4.1 with these radii as inputs, and then return the subsequent multicut.

Algorithm 4.4.2 Efficient Disjoint Region Growing Algorithm

Let $G = (V, E)$ be an undirected graph.
Let $c : E \to \mathbb{Q}_{\geq 0}$ be a cost function.
Let $\{(s_i, t_i)\}_{i=1}^{k}$ be a collection of $k \geq 1$ source-sink pairs of G.
1: Compute $\boldsymbol{r}^* \in [0, \frac{1}{2})^k$, which satisfies Proposition 4.4.6
2: Execute Algorithm 4.4.1 with the above inputs.
3: Store the above output in edge set $F^{\boldsymbol{r}^*}$.
4: Return $F^{\boldsymbol{r}^*}$.

We remark that while the above algorithm computes $\boldsymbol{r}^*$ before executing Algorithm 4.4.1, this is not necessary in practise. A closer examination of the proof of Proposition 4.4.7 shows that the indices of $\boldsymbol{r}^*$ may be computed dynamically as Algorithm 4.4.1 executes. Namely, given $r_1^*, \ldots, r_{i-1}^*$, where $1 \leq i \leq k$, let us suppose we have computed the graph G_i and edge assignment $\boldsymbol{d}^i$, as in Algorithm 4.4.1 (if $i = 1$, then no radii have yet been computed). At this point, we may mimic the procedure outlined in the proof of Proposition 4.4.7 to find a raidus r_i^* for which,

$$c(\delta(B_{r_i^*}(s_i))) \leq 2 \ln(k+1) w(B_{r_i^*}(s_i)),$$

where $B_{r_i^*}(s_i)$ is the region formed using radius r_i^* about source node s_i in G_i. As this equation will clearly hold for each $i = 1, \ldots, k$, the analysis of this dynamic algorithm will be identical to that of Algorithm 4.4.2.

Theorem 4.4.9. *Algorithm 4.4.2 achieves an approximation guarantee of* $4 \ln(k+1)$ *in polynomial time.*

Proof. We first observe that the above algorithm is able to compute $\boldsymbol{r}^*$ efficiently, as a result of Proposition 4.4.7. As Algorithm 4.4.1 is known to execute in polynomial time, it is clear that the above algorithm will as well.

Let us now suppose that $F_1^{\boldsymbol{r}^*}, \ldots, F_k^{\boldsymbol{r}^*}$ and $U_1^{\boldsymbol{r}^*}, \ldots, U_k^{\boldsymbol{r}^*}$ are the cuts and regions formed while executing Algorithm 4.4.1. We observe that by Proposition 4.4.7,

$$c(F_i^{\boldsymbol{r}^*}) \leq 2 \ln(k+1) w(U_i^{\boldsymbol{r}^*}),$$

for $i = 1, \ldots, k$. This implies that if $F^{\boldsymbol{r}^*}$ is the output of the above algorithm, then by Proposition 4.4.6,

$$c(F^{\boldsymbol{r}^*}) \leq 4 \ln(k+1) OPT_f,$$

where OPT_f is the cost of an optimum solution to LP (4.4.2). Thus, the above algorithm achieves an approximation guarantee of $4 \ln(k+1)$, as $OPT_f \leq OPT$, where OPT is the cost of an optimum multicut of G.

□

We remark that Algorithm 4.4.2 can be thought of as the derandomization of a certain randomized algorithm we now describe. This randomized algorithm first samples $\rho_1, \ldots, \rho_k$ from $[0, \frac{1}{2})$ independently and uniformly at random. It then passes $\boldsymbol{\rho} := (\rho_1, \ldots, \rho_k)$ to Algorithm 4.4.1, together with G, c and the source nodes $s_1, \ldots, s_k$. Using these parameters as inputs, Algorithm 4.4.1 returns a random multicut F which is used as the output of the randomized algorithm.

Algorithm 4.4.3 Randomized Disjoint Region Growing Algorithm

Let $G = (V, E)$ be an undirected graph.
Let $c : E \to \mathbb{Q}_{\geq 0}$ be a cost function.
Let $\{(s_i, t_i)\}_{i=1}^{k}$ be a collection of $k \geq 1$ source-sink pairs of G.
1: Sample $\rho_1, \ldots, \rho_k$ independently and uniformly at random from $[0, \frac{1}{2})$.
2: Initialize $\boldsymbol{\rho} \leftarrow (\rho_1, \ldots, \rho_k)$.
3: Execute Algorithm 4.4.1 with the above inputs.
4: Initialize F, and use it to store the above output.
5: Return F.

As the above algorithm invokes Algorithm 4.4.1, we can denote the random regions it builds by $U_1, \ldots, U_k$, and the random cuts by $F_1, \ldots, F_k$. Moreover, the random variables $w(U_1), \ldots, w(U_k)$ and $c(F_1), \ldots, c(F_k)$ describe the weights and costs of their respective regions and cuts. A similar computation involving the integrals seen in the proof of Proposition 4.4.7 allows us to conclude that,

$$\mathbb{E}\, c(F_i) \leq 2\, \ln(k+1)\, \mathbb{E}\, w(U_i)$$

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

We can now mimic the remarks before Proposition 4.4.6, by defining the random variables $\beta_i := \sum_{\substack{e \in E_i: \\ e \cap U_i^{r^*} \neq \emptyset}} c(e)\, d_e$, for $i = 1, \ldots, k$. Clearly, $w(U_i) \leq \beta_i + \beta/k$, after accounting for the potential weight based at source node s_i. Moreover, since the regions $U_1, \ldots, U_k$ are disjoint, we have that

$$\sum_{i=1}^{k} w(U_i) \leq \sum_{i=1}^{k} \beta_i + \beta \leq 2\, \beta,$$

where β is the deterministic quantity, $\sum_{e \in E} c(e)\, d_e$. Combining these equations,

$$\begin{aligned}
\mathbb{E}\, c(F) &= \sum_{i=1}^{k} \mathbb{E}\, c(F_i) \\
&\leq 2\, \ln(k+1) \sum_{i=1}^{k} \mathbb{E}\, w(F_i) \\
&\leq 4\, \ln(k+1)\beta,
\end{aligned}$$

after taking expectations. As β is equal to OPT_f, the optimum value of a fractional multicut of G, we arrive at the following theorem:

Theorem 4.4.10. *Algorithm 4.4.2 achieves an expected approximation guarantee of* $4\, \ln(k+1)$ *in polynomial time.*

We now return to analyzing Algorithm 4.4.1, and its implications towards the integrality gap of IP (4.4.1).

Suppose we fix $k \geq 1$, and restrict our attention to multicut problem instances with exactly k source-sink pairs. In particular, the problem instance I consists of a graph $G = (V, E)$, a cost function $c : E \to \mathbb{R}$, and k source-sink pairs $\{(s_i, t_i)\}_{i=1}^{k}$. Algorithm 4.4.1 then computes a multicut F, for which

$$c(F) \leq 4\, \ln(k+1)\, OPT_f(I),$$

where $OPT_f(I)$ is the value of an optimum solution to LP (4.4.2). As a result of this comparison, if we restrict IP (4.4.1) to instances with exactly k source-sink pairs, then its integrality gap is bounded above by $4\, \ln(k+1)$.

Theorem 4.4.11. *Let $k \geq 1$ be fixed, and suppose we only consider multiway cut problem instances with exactly k source-sink pairs. The integrality gap of the IP* (4.4.1) *restricted to problem instances of this type is at most* $4\ln(k+1)$.

On the other hand, suppose we view the integrality gap as a function of k. That is, for each $k \geq 1$, define

$$\phi(k) := \sup_I \{\frac{OPT(I)}{OPT_f(I)} : I \text{ has } k \text{ source-sink pairs}\}$$

where $\phi : \mathbb{Z}_{\geq 0} \to [1, \infty)$. If we consider the asymptotic behaviour of ϕ as k becomes large, then we can lower bound its growth. It turns out that up to a constant factor, this lower bound matches the approximation guarantee of Algorithm 4.4.1. Thus, we may conclude that the integrality gap as a function of k is $\Theta(\ln(k))$. In order to prove this claim, we first consider a definition for graphs which are regular.

Given a d-regular graph H, let us refer to it as an *expander graph* provided for each non-empty subset $S \subseteq V(H)$, $S \neq V$

$$e(S, \overline{S}) \geq \min\{|S|, |\overline{S}|\}, \tag{4.4.8}$$

where $e(S, \overline{S})$ is the number of edges between S and $\overline{S}$. This definition is often stated in the terminology of *graph conductance*, however Equation 4.4.8 will suffice for our purposes.

It turns out that for each even $d \geq 4$, there exists some $l_d \geq 1$ such that for all $l \geq l_d$, there is a d-regular graph on l vertices which is an expander graph. This can be proven by the probabilistic method, where one shows that a uniformly

random d-regular graph has a positive probability of having the *expansion property* of Equation 4.4.8. We shall not include the proof of this statement here, but the reader may consult the book "Introduction to Random Graphs" by Frieze and Karonski for the details of this argument [FK16].

Let us now fix an even $d \geq 4$ and some $l \geq l_d$. Moreover, assume that $H_l = (V_l, E_l)$ has l vertices, satisfies the expansion property above, and has unit capacity edges. We first observe that if we fix $v \in V_l$ and take some integer $\beta \geq 1$, then there are at most

$$\sum_{i=0}^{\beta-1} d^i < d^\beta, \tag{4.4.9}$$

many vertices in H_l with distance at most $\beta - 1$ from v. In particular, if we set $\beta = \lfloor \log_d(l/2) \rfloor$, then there are at least $l/2$ vertices at distance β or more away from v.

Let us now define $\mathcal{S}_l$ to be the source-sink pairs of H_l, where a distinct pair of vertices $s, t \in V_k$ is contained in $\mathcal{S}_l$, if and only if $dist(s,t) \geq \beta$. As a consequence of the above observations, we may conclude that $|\mathcal{S}_l| \geq l^2/4$, and so $|\mathcal{S}_l| = \Theta(l^2)$.

If we now consider a multicut M of H_l which separates $\mathcal{S}_l$, then we claim that $|M| \geq l/2$. In order to see this, consider the components $U_1, \ldots, U_q$ of $H_l \setminus M$, where $1 \leq q \leq l$. In light of the definition of $\mathcal{S}_l$, we know that for each $i = 1, \ldots, q$, $diam(H_l[C_i]) < \beta$, where $diam(H_k[C_i])$ is the diameter of the the induced subgraph $H_l[C_i]$. Using the same computations as in Equation 4.4.9, we may conclude that $|C_i| < d^\beta \leq l/2$. In particular, this implies that $|C_i| \leq |\overline{C_i}|$, and so $e(C_i, \overline{C_i}) \geq |C_i|$ for each $i = 1, \ldots, l$, by the expansion property of H_l.

As a result of these computations, we know that

$$\sum_{i=1}^{q} e(C_i, \overline{C_i}) \geq \sum_{i=1}^{q} |C_i| = l.$$

On the other hand, each edge of M occurs in at most two components of $H_l \setminus M$, so we know that $|M| \geq l/2$. If $OPT(H_l, \mathcal{S}_l)$ is the value of an optimum multicut of $(H_l, \mathcal{S}_l)$, then this result implies that $OPT(H_l, \mathcal{S}_l) \geq l/2$. Our goal will now be to bound the value of a fractional multicut of H_l. Combined with the above result, this will imply a lower bound of the integrality gap of $(H_l, \mathcal{S}_l)$; namely, the quantity

$$\frac{OPT(H_l, \mathcal{S}_l)}{OPT_f(H_l, \mathcal{S}_l)},$$

where $OPT_f(H_l, \mathcal{S}_l)$ is the value of an optimum solution to LP (4.4.2).

By the duality theorem for linear programs, a maximum multi-commodity flow has value equal to the size of a minimum fractional multicut. We shall therefore work with (fractional) multi-commodity flows, opposed to directly considering fractional multicuts of $(H_l, \mathcal{S}_l)$.

Let us suppose that $(f_p)_{p \in \mathcal{P}}$ is a multi-commodity flow through $(H_l, \mathcal{S}_l)$, where $\mathcal{P}$ is the set of all simple paths between members of $\mathcal{S}_l$. We first observe that by the feasibility of the solution,

$$\sum_{p \in \mathcal{P}: e \in p} f_p \leq 1$$

for each $e \in E$, as the graph H_l has unit capacity edges. By summing over all the edges, this implies that

$$\sum_{e \in E} \sum_{p \in \mathcal{P}: e \in p} f_p \leq |E_l| = \frac{ld}{2},$$

where the last equality follows from the d-regularity of H_l. On the other hand, we know that for each $p \in \mathcal{P}$

$$\sum_{e \in E : e \in p} f_p \geq \beta f_p,$$

as each path p connects a pair of vertices at least distance β away. It follows that

$$\beta \sum_{p \in \mathcal{P}} f_p \leq \sum_{e \in E} \sum_{p \in \mathcal{P} : e \in p} f_p \leq \frac{ld}{2},$$

after exchanging the summation of the above equation. We may therefore conclude that

$$\sum_{p \in \mathcal{P}} f_p \leq \frac{ld}{2\beta}.$$

Thus, each multi-commodity flow has value at most $\frac{ld}{2\beta}$, and so if $OPT_f(H_l, \mathcal{S}_l)$ is the value of an optimum solution to LP (4.4.2), then we know that

$$OPT_f(H_l, \mathcal{S}_l) \leq \frac{ld}{2\beta}.$$

As a result,

$$\frac{OPT(H_l, \mathcal{S}_l)}{OPT_f(H_l, \mathcal{S}_l)} \geq \frac{\beta}{d} = \alpha_d \ln(l),$$

for some constant $\alpha_d > 0$ dependent on d. Moreover, if we set $k_l := |\mathcal{S}_l|$, then we know that $l^2/4 \leq k_l \leq l^2$, and so

$$\frac{OPT(H_l, \mathcal{S}_l)}{OPT_f(H_l, \mathcal{S}_l)} \geq \frac{\alpha_d}{2} \ln k_l.$$

Thus, for every $l \geq l_d$, the existence of $(H_l, \mathcal{S}_l)$ implies that

$$\phi(k_l) \geq \frac{\alpha_d}{2}(k_l),$$

where $\phi(k_l)$ is the integrality gap of multicut instances with exactly k_l source-sink pairs. On the other hand, since $|S_l| \to \infty$ as $l \to \infty$, this means that there are infinitely many $k \geq 1$ such that

$$\phi(k) \geq \frac{\alpha_d}{2}(k).$$

As the integrality gap as a function of k is clearing increasing, this implies the result below.

Theorem 4.4.12. *There exists a constant $\alpha > 0$ and some $k_0 \geq 1$ such that for all $k \geq k_0$,*

$$\alpha \ln(k+1) \leq \phi(k) \leq 4 \ln(k+1).$$

Rather, the integrality gap of the restricted IP (4.4.1) *is between $\alpha \ln(k+1)$ and $4 \ln(k+1)$, when considered as a function of k.*

As a result of this theorem, we know that Algorithm 4.4.1 has a performance guarantee that is best possible among algorithms which employ rounding techniques, up to a constant factor.

CHAPTER 5
Conclusion

After an introduction to some basic concepts from complexity theory and linear programming, the thesis presented a number of approximation algorithms for a variety of network flow and minimum cut type problems. More specifically, the multiway cut problem was examined, and seen to have an algorithm achieving an approximation guarantee of 3/2. When we later considered the multi-commodity flow problem on trees, an approximation guarantee of 1/2 was witnessed, together with a performance guarantee of 2 for the restricted multicut problem. The thesis concluded with an examination of the multicut problem on general graphs, where an approximation guarantee of $4\ \ln(k+1)$ was seen, assuming k source-sink pairs were to be separated. All of these claims are known as positive results, as they establish upper bounds on how well their respective problems can be approximated (lower bounds in the case of maximization problems).

A natural question to wonder is whether there exist algorithms whose performance guarantees are better than previously seen. In the context of the first IP formulation of the multiway cut problem, an integrality gap of $2\,(k-1)/k$ was established, thus limiting the performance guarantees of algorithms which apply primal-dual and randomized rounding techniques. Similarly, the multicut problem on general graphs was seen to have an integrality gap of $\Omega(\log(k))$, when viewed as a function of its number of source-sink pairs. This confirmed that the

approximation algorithm presented is best up to a constant factor, among algorithms which employ linear programming techniques. Both of these claims are known as *hardness of approximation results*, or negative results, as they establish lower bounds on how well their respective (minimization) problems can be approximated, assuming LP techniques are used.

While limiting the effectiveness of LP techniques is useful, it does not discount what combinatorial algorithms may achieve, or LP algorithms whose analysis is nonstandard (no comparison between OPT and OPT_f is made in the analysis). It is therefore often desirable to extend these hardness results in such a way that they account for *all* possible approximation algorithms. The highest possible standard for such a result is to witness an algorithm achieving an approximation guarantee of α, and to prove a hardness result limiting any approximation algorithm from improving on α, assuming $\mathbf{P} \neq \mathbf{NP}$.

The multicut problem on general graphs is known to *not* have an approximation algorithm achieving a constant factor of $\alpha \geq 1$, assuming the Unique Games Conjecture - a stronger assumption than $\mathbf{P} \neq \mathbf{NP}$, but still believable [CKKRS06]. It would of course be desirable to investigate whether this assumption could be dropped without weakening this claim. Another well studied problem is the vertex cover problem, which cannot be approximated by a constant smaller than 1.3606, assuming $\mathbf{P} \neq \mathbf{NP}$ [DS05]. This of course also limits how well the multicut problem can be approximated on trees, in light of the approximation preserving reduction from the vertex cover problem to the multicut problem presented earlier. Improving the gap between these bounds in the case of the multicut problem on

trees is thus a natural question to consider. There are of course many other unresolved questions on optimization problems on graphs. Frequently, these problems have gaps between their best known positive and negative approximation results. Better understanding the state of these results would serve as a natural followup to the work done in this thesis.

REFERENCES

[Vv11] Vazirani, V., *Approximation Algorithms*, Berlin: Springer, 2011.

[WS11] Williamson, D., & Shmoys, D., *The Design of Approximation Algorithms*, Cambridge: Cambridge University Press, 2011.

[AB16] Arora, S., & Barak, B., *Computational Complexity*, New York (NY): Cambridge University Press, 2016.

[KT14] Kleinberg, J., & Tardos, E., *Algorithm Design*, Boston, Mass. [u.a.]: Pearson/Addison-Wesley, 2014.

[FK16] Frieze, A., & Karonski, M., *Introduction to Random Graphs*, Cambridge University Press, 2016.

[Cp97] Crescenzi, P., A Short Guide to Approximation Preserving Reductions, *IEEE Conference on Computational Complexity*, 1997.

[EK72] Edmonds, J., & Karp, R.M., Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems,*Combinatorial Optimization*, 1972.

[CKKRS06] Chawla, S., Krauthgamer, R., Kumar, R., Rabani, Y., & Sivakumar, D. On the hardness of approximating multicut and sparsest-cut, *Comput. Complexity* 15 (2006), no. 2, 94114.

[DS05] Dinur, I., & Safra, S., On the hardness of approximating minimum vertex cover. *Ann. of Math. (2)* 162 (2005), no. 1, 439485.