Vehicle Routing: A Survey of Approximation Algorithm Techniques

by

Devanshu Pandey

An essay
presented to the University of Waterloo
in fulfillment of the
essay requirement for the degree of
Master of Mathematics
in
Combinatorics and Optimization

Waterloo, Ontario, Canada, 2013

Abstract

Routing problems are some of the oldest and most extensively studied combinatorial optimization problems. Many of these routing problems are hard and hence are unlikely to admit polynomial time solutions (barring resolution of the massive P? = NP question). This has led to an extensive study of their approximability. The two main types of algorithmic techniques in the literature use structural ideas based on decomposition of minimum spanning trees and set-covering based LP formulations. In this essay, we survey the use of these techniques while focusing on routing problems modeled as graph-covering problems. k-tree cover and Rooted Tree cover (introduced by Arkin *et al.* and Even *et al.* respectively) are discussed in greater detail. We later consider a new generalization of the two problems (coined Budgeted Tree Cover) and give a natural 5-approximation algorithm for the problem. As the algorithm given is a direct generalization of older techniques, we believe that it should be easy to achieve a better approximation guarantee. We discuss a paper by Khani and Salavatipour to describe some new ideas that could lead to a better algorithm for Budgeted Tree Cover.

Acknowledgements

I came far enough to even write these words because of the guidance and support of a lot of different people. I would like to take this opportunity to try and express my gratitude towards everyone who made this possible.

I would like to first thank my supervisor, Dr. Jochen Könemann. During my first term at Waterloo, I told Jochen I felt “stretched” due to the workload. He replied, “Good! Stretching helps you grow”. I believe that best describes my supervisor. The only things that outshine his invaluable technical expertise and guidance are his passion for the subject and an awe-inspiring work ethic. I thank him for being a constant source of instruction and motivation.

Next, I thank Dr. Konstantinos (Costis) Georgiou. Costis has been both a second supervisor, and (if I may be so bold) a colleague. He has taught me how valuable it is to maintain the balance between intuition and detail. I would also like to thank him for being the reader for all my write-ups and suggesting several improvements for each.

To my Ma and Papa, I owe my very existence. Since there are way too many things to mention here, I simply say *thank you*. I can’t say enough to thank my grandparents. Though they are no longer with me, their memories are a never-ending source of love and support. I thank my aunts Nandini and Ragini and my cousins Aakarsh and Natasha for teaching me never to take life too seriously.

Things would have been very dreary without the company of friends. Thank you to Nabiha Asghar, Danielle Drainville (my officemate), David Qian, Rebecca Tessier, Leanne Stuive, Kenneth Webster and Brandon Weir for all the fun times. I would like to distinguish Nishad Kothari for all the interesting discussions at dinner (both mathematical and otherwise) and for helping me out through quite a few tight spots. A shout out to my dearest friend and fellow geek from back home: Apurv Ray.

A special thank you to Dr. Atri Rudra for giving me a fantastic introduction to the world of algorithms and theoretical computer science. I’d like to conclude by thanking all of my professors at Waterloo for four terms of mathematical adventure and enrichment.

Dedication

I dedicate this essay to the memory of my first teacher and my grandfather, Ramchandra Pandey. I owe him my belief in myself.

Table of Contents

List of Figures

Chapter 1

Introduction

Since it's proposal by Dantzig and Ramser in 1959 [7], the Vehicle Routing Problem (VRP) and it's variants have been widely studied. At a high level, one can look at the problem as a generalization of the famous Traveling Salesman Problem (TSP). An instance of the vehicle routing problem is defined by a set of locations to be serviced by one or more vehicles that are all located a starting depot. The goal is to find the best (for example: cheapest or shortest) possible way such that each location is visited exactly once by one of the vehicles. It is not hard to see that vehicle routing problems have a variety of direct applications in the industry. One very obvious application is designing delivery routes for logistics giants like FedEx or UPS where timely and cost-efficient delivery of goods is paramount. Another not so obvious but well studied application is that of assigning cranes to containers at a port loading dock. The developments on vehicle routing have occured on two main fronts: *Approximation algorithms* (See Section 1.5) which provide a guarantee on the cost of your solution regardless of the instance size and various *heuristics* from the field of Operations Research where the performance often depends on the size of the given instance. This essay aims to provide a detailed survey of widely used techniques for designing approximation algorithms for VRP. Treating these real world problems as optimization problems calls for a mathematical model encompassing all the intricacies of a given problem.

1.1 A Mathematical Model for Routing

When routing a set of vehicles in order to perform a certain task, we measure their performance based on a predetermined parameter. This parameter could be the amount of time taken to reach their destination or the total cost of gas used by all vehicles combined. Regardless of the measure, we can always look at the problem via the lens of optimization. An *optimization problem* is a set of instances and an associated set of constraints that determine a set of feasible solutions for each instance. Thus for any input, one gives a concise description of an acceptable (feasible) solution is. The performance of a solution is measured by an *objective function* defined on the set of solutions that maps each solution to the reals.Thus one needs to find a solution that optimizes this objective function. Routing problems are usually minimization problems where the objective is to find a feasible solution while minimizing the cost over all feasible solutions.

A very simple measure of cost is the total distance travelled by all vehicles combined. Another closely related notion is the total time taken. The objective can be made slightly more complex by trying to minimize the maximum time taken by any of the vehicles. The real world instances are even more complex since we have to take into account the individual carrying capacities and speeds of vehicles. There might also be time-windows associated that dictate exactly when the deliveries are to be made or constraints limiting how far a vehicle is allowed to travel based on it's fuel capacity. A very clean way of incorportating most of these details is to use Graph Theory to model the problems.

1.2 Graph Theory Essentials

All of the algorithms in this essay (and most in literature) model routing problems using graphs. This is because the central idea behind the definition of a graph is that of points and connectivity. A graph G is given by a pair (V, E) of sets. V is the set of *vertices* of the graph and $E \subseteq \{u, v : \forall u, v \in V\}$ is called the set of *edges* that connects certain vertices in G. For a graph H we will use $V(H)$ to denote the vertex set of H and $E(H)$ to denote the edge set. For an edge $e = \{u, v\}$, we call u and v the *end-points* of e and we say that u and v are *adjacent*.

If all possible edges in a graph are present, then the graph is said to be *complete*. For a vertex v, $N(v) = \{u \in V : \{u, v\} \in E\}$ is called the neighbourhood of v. The set of edges connecting v to it's neighbours is denoted by $\delta(v)$. For any subset U of the vertices, we can generalize the notion of a neighbourhood by defining U's neighbourhood $N(U)$ be the union of neighbourhoods of each of the vertices in U i.e.$N(U) = \cup_{u \in U} N(u)$. An edge e is said to be *incident* to a vertex v if e belongs to $\delta(v)$. Note that $\delta(v)$ is also called the *cut* of v. The cut $\delta(U)$ of a subset U of vertices can be defined as the set of edges with exactly one end-point in U. An incidence matrix is a simple way of representing a graph for computational purposes (See Section 1.2.1 for an application). So given $G = (V, E)$ where $V = \{1, 2, \cdots, n\}$ and $E = \{1, 2, \cdots, m\}$, the incidence matrix of G is a matrix $A \in \{0, 1\}^{m \times n}$ such that $A[i, j] = 1$, if edge i is incident to vertex j and 0 otherwise.

Two vertices u and v in V are said to be *connected* if there is a sequence of edges (a *path*) in the graph such that one can follow this sequence to get from u to v. If every pair of vertices in a graph is connected, then the graph is called a *connected* graph. A *cycle* is a path that starts and ends at the same vertex. A cycle is called *Hamiltonian* if it visits all the vertices in the corresponding graph. A graph is called a *tree* if it is connected and has no cycles. A *spanning* tree is a tree that contains all the vertices of a graph. A *forest*

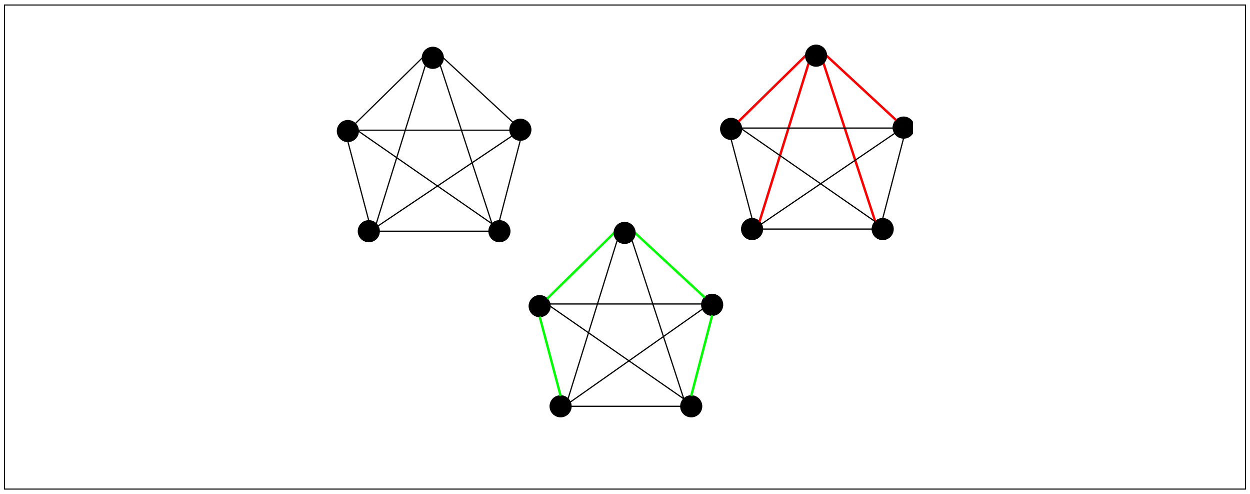

Figure 1.1: The complete graph on 5 vertices. Let all the edges on the outer face have length one and all the inner edges have weight two. The red edges form a spanning tree (weight 6) while the green ones form a minimum spanning tree (weight 4)

is defined as a collection of one or more trees. Thus a forest can be thought of as a set of trees or a possibly disconnected graph where each connected component is a tree.

A *weighted* graph $G = (V, E)$ is one where we associate weights with the edges of the graph using a function $w : E \to \mathbb{R}$. One can then define a *minimum* spanning tree as the spanning tree with the least weight over all possible spanning trees of G. Throughout this essay, we will use *weight* or *length* of a set of edges to refer to the sum of all the edges in the set. Note that using weighted graphs to model routing problems is natural since the edge lengths then can be used to represent the distance between the two end-points (read locations). It makes sense to assume that traveling from point A to point B directly does not take more time than going from A to another point C and then to point B. Also, in a real world situation we can always go from any location to any other. The previous two observations bring us to the idea of a *metric completion*. So given a graph $G = (V, E)$ with edge lengths $l : E \to \mathbb{R}$, we can add in any missing edges to create a complete graph by simply adding the edge and setting the length of the new edge to be the shortest path distance (for instance) between the two end-points in G. Thus when we say we are in a metric, the underlying graph can be assumed to be complete. So given a complete graph $G = (V, E)$ we say that l is a metric if it satisfies the following conditions:

- $l(u, v) = 0$ if and only if $u = v$, $l(u, v) > 0$ otherwise,
- $l(u, v) = l(v, u)$ (i.e. the function is symmetric), and
- For any $u, v, w \in V$, $l(u, v) + l(v, w) \geq l(u, w)$ (l satisfies the triangle inequality).

Note that all of these conditions are satisfied when we use the shortest path metric.

A very important infinite family of graphs is that of bipartite graphs. A graph $G = (T \cup R, E)$ is said to be bipartite where T and R are sets such that no edge in E has both end-points in either T or R. Bipartite graphs lend themselves well to various applications in both routing and assignment problems which are two important classes of combinatorial problems (See Schrijver's [32] for more information on these classes of problems). Assignment problems involve two or more kinds of objects where we want to "match" one type of object to another. This brings us to matchings in graphs. Matching theory by itself is a vast field of study but we will only touch upon areas that are directly useful in the algorithms discussed later in this write-up.

1.2.1 Matchings

Consider a graph $G = (V, E)$ and a function $l : E \to N$ that assigns weights to the edges of the graph. A subset M of the edges is then called a *matching* if no vertex is incident to more than one edge from M. For a subset U of the vertices, we say that a M is *U-saturating* if every vertex in U is incident with one edge in M. A matching is called *perfect* if it saturates the entire vertex set. The objective of the Minimum-weight Perfect Matching problem is to find a matching $M \subseteq E$ that minimizes $\sum_{e \in M} l(e)$ over all perfect matchings M. For a subset U of the vertices we call the cut $D = \delta(U)$ an *odd-cut* if U has an odd number of vertices. Using a simple parity argument one can see that any perfect matching M has at least one edge from every odd-cut. The problem of finding a min-weight perfect matching can be formulated as an *integer program* (IP). As integer linear programs are fundamental to representing optimization problems, a very short discussion on integer programs is included in the Appendix (See Appendix A).

Given the graph $G = (V, E)$, let $\mathcal{C}$ be the set of all odd-cuts where the associated vertex set is not a singleton. The problem of finding a min-weight perfect matching in G can then be formulated as the following ILP:

$$\begin{aligned} \min \quad & \sum_{e \in E} l_e x_e & & (P) \\ \text{s.t.} \quad & x(\delta(v)) = 1 & \text{for all } v \in V & \\ & x(D) \geq 1 & \text{for all cuts } D \in \mathcal{C} & \\ & x_e \in \{0, 1\} & \text{for all } e \in E & \end{aligned}$$

Let (P) denote the above program. Here x is a characteristic vector of a matching i.e. $x_e = 1$ if edge e is included in the matching and 0 otherwise. The first constraint ensures that every vertex has exactly one incident edge in any solution for (P). The second constraint deals with odd-cuts with corresponding to subsets in $\mathcal{C}$ and ensures that any feasible solution to (P) has at least one edge from every odd-cut. As (P) is an integer program x is only allowed to take on integer values there by enforcing that an edge e is either in the solution or not (so x_e cannot take on fractional values). (Such a program is also called a $0,1$-integer program). Since a feasible solution to (P) satisfies these contraints, one can check that each feasible solution corresponds to a perfect matching in G. Edmonds' fundamental theorem showed that the optimal value of the linear programming relaxation (See Appendix A) of (P) is the weight of a minimum-weight perfect matching [10]. Note that we use the linear programming relaxation of (P) since solving $0,1$-integer programs is NP-Hard [24] (See Section 1.5 for a definition of NP-Hardness).

The problem of finding minimum-weight matching in a bipartite graph can also be solved optimally by solving a linear program. Consider a bipartite $G = (V, E)$ with edge lengths $l : E \to \mathbb{R}$ and incidence matrix A.

$$\begin{aligned} \min \quad & \sum_{e\in E} l_e x_e & (P^{'}) \\ \text{s.t.} \quad & A \cdot x = \mathbf{1} \text{for all } v \in V & \\ & x_e \in \{0,1\} \text{for all } e \in E & \end{aligned}$$

Here $\mathbf{1}$ is an $|V|$-dimensional vector of all 1's. A matrix $M \in \mathbb{Z}^{n\times m}$ is called totally unimodular if every square sub-matrix of M has determinant -1, 0 or 1. Note that the incidence matrix of any bipartite graph is totally unimodular [9]. Thus the incidence matrix A in $P^{'}$ is totally unimodular and every extreme point of the polyhedron $\{Ax = \mathbf{1}\}$ is integral [6]. One can prove that A^T is also totally unimodular and this implies that the integrality holds for the corresponding dual polyhedron as well. Birkhoff showed that the optimal solution to $(P^{'})$ corresponds to a minimum-weight perfect matching in G [6] and the stronger statement that the optimal value of the $(P^{'})$ gives us the weight of a minimum-weight perfect matching. An important theorem that characterizes the existence of a matching in a bipartite graph is Hall's theorem. We state it here without proof:

Theorem 1.2.1. *Let U, W be a bipartition of a bipartite graph $G = (V, E)$ then a U-saturating matching exists if and only if $\forall X \subseteq U$, $|N(X)| \geq |X|$.*

This concludes our of graph theory basics. For a vehicle routing problem modeled as a graph, assigning routes for vehicles corresponds to "covering" a graph with a particular class of graphs. This will tie in to the three main problems that form the meat of this essay. Now that we have the model in place, we can move to the central topic of this essay – Routing problems.

1.3 The Traveling Salesman Problem

A discussion on Vehicle Routing Problem (VRP) would be incomplete without talking about the more basic (though in no way any easier) Traveling Salesman Problem (TSP).

It is one of the oldest and extensively studied problems in combinatorial optimization. The exact origins of the problem seem to be unclear (See Schrijver's study [32] for an exposition on the origins of TSP). For a comprehensive discussion of the problem in it's full glory I refer the reader to William Cook's work [5]. An instance of TSP consists of an undirected, edge-weighted, connected graph and any Hamiltonian cycle in the graph is a feasible solution (unless stated otherwise, assume that we are working with connected graphs). The objective is then to find a Hamiltonian cycle of minimum edge weight. We can think of the vertices of the graph as cities and the weight of an edge as the cost of traveling between the cities corresponding to the end-points of the edge. The name of the problem stems from the idea that we can think of a salesman trying to organize a tour over all the cities while minimizing the total cost incurred.

We will see later (See Section 1.5) why the one of the most studied variants is when the edge lengths of the graph form a metric. For any given $G = (V, E)$ we can construct the shortest-path metric completion $G^* = (V, E^*)$ of G by adding any missing edges and assigning each new edge a weight equivalent to the length of the shortest path in G between the end-points. Thus we can always assume that the underlying graph in any TSP instance is always a complete graph. Now for a complete graph on n vertices, there are $n!/2$ Hamiltonian cycles (and hence, $n!/2$ feasible solutions!). This suggests that one may not be able to solve the problem efficiently simply by enumerating the feasible solutions. If this was not complicated enough, one can add different constraints and modify the problem to incorporate highly complex real world scenarios. This brings us squarely to vehicle routing problems.

1.4 The Vehicle Routing Problem

Getting from TSP to VRP is quite easy. All you need to do is replace the salesman with one or more of your favorite vehicles. Thus an instance of VRP consists of an undirected, edge-weighted graph and a fleet of vehicles. A feasible solution to the problem corresponds to a set of cycles (as routes for the vehicles) such that the union of the vertices contained in the cycles is the whole vertex set. Formally, we have a complete graph $G = (V, E)$ with edge lengths given by $l : E \to \mathbb{N}$ that form a metric and a fleet of k vehicles. A feasible solution is a set $\{C_1, C_2, \cdots, C_k\}$ of cycles of G such that $\cup_{1 \le i \le k} V(C_i) = V$. Based on the variant under consideration, objective functions differ. One possible objective could be to minimize the total length of all tours i.e. to minimize $\sum_i \sum_{e \in C_i} l(e)$. Another widely considered objective is to minimize the length of the longest cycle in the solution (also called the *makespan* of the solution) i.e. minimize $max_i \sum_{e \in C_i} l(e)$.

VRP was introduced by Dantzig and Ramser in 1959 [7] as the Truck Dispatching problem with the objective of minimizing the total length of all the tours. The real-world analogy can be described by considering a set of cities and a fleet of vehicles. Each city is to be serviced by one vehicle. Though TSP has several applications, vehicle routing problem was introduced as a wider, more encompassing basis for several variants that can be constructed by adding the corresponding constraints. Some of these variants are covered in the following part of this essay. However, the field is extensive and for a more comprehensive discussion I would like to refer the reader to Toth and Vigo's work [34].

1.4.1 Capacitated Vehicle Routing

One natural constraint to take into consideration is that fact that vehicles have finite capacities. One of the first papers to investigate multiple variants of vehicle routing and traveling salesman problems was by Haimovich and Rinnooy-Kan [23]. They discussed a variant of vehicle routing called Capacitated Vehicle Routing Problem (CVRP). CVRP is a natural variant where the depot has an infinite supply of some resource, the vehicles can carry a fixed amount of the resource at a time. If we require each vehicle to make exactly one trip then the problem can be formulated as follows: Consider a graph $G = (V, E)$ and lengths $l : E \to \mathbb{R}_+$ that form a metric. Say you have a fleet of k vehicles each with capacity D. A feasible solution to CVRP is a set of cycles $\{C_1, C_2, \cdots, C_k\}$ such that $l(C_i) \leq D$ for all i.

A further intricacy can be introduced into the problem by observing that a vehicle may carry multiple items (not necessarily identical) and that each city might have a different demand associated with each item. In this case we modify the above definition by associating a demand function $d : V \to \mathbb{R}^m_+$ where m is the number of types of items and the modified capacity constraint $\sum_{v \in C_i} d(v) \leq D$ for all i where the capacity D is a vector in $\mathbb{R}^m$.

Another interesting idea is that of *split deliveries*. For a real world example one may think of trucks delivering dirt and cement to construction sites. Here the constraint is to meet the demand of each location regardless of the number of vehicles that contribute to meeting this demand. Let the contribution of a vehicle j at location v be $a_{i,j} \in \mathbb{R}^m$. A feasible solution then is a set of cycles $\{C_1, C_2, \cdots, C_k\}$ that satisfies the additional constraint $\sum_{1 \leq j \leq m} a_{i,j} = d(v)$ so that all demands are satisfied.

Previously we discussed how TSP and VRP are closely related. CVRP reduces to TSP in the absence of capacity constraints and when we have a single vehicle. Haimovich and Rinnooy-Kan *et al.* [23] showed that the best known approximation algorithm for CVRP achieves a guarantee of $\rho + 1$, where ρ is the best known approximation for TSP. When the objective is to minimize the makespan of the tours and all vehicles are identical, Frederickson *et al.* showed that a simple tour-splitting heuristic (See Section 2.1) acheives a 3-approximation [13]. When the vehicles have different speeds (called Heterogenous CVRP) Gortz *et al.* give a constant factor approximation based on the tour-splitting heuristic of [13] and a 2-approximation for scheduling on unrelated machines by Shmoys and Tardos [33].

1.4.2 Vehicle Routing with Time Windows

Observe that there can be natural situations where a delivery made to a location has to be within a certain time window (for example receiving your gifts before you leave for the holidays). This introduces the variant of vehicle routing referred to as Vehicle Routing with Time Windows (VRP-TW). In this case we associate a time with each edge that represents the amount of time needed to travel between the end-points and with vertex an interval. Then the objective is to nd a minimum cost set of routes for the vehicles so that we visit every location while ensuring that we visit the location within the assigned time windows. The problem was first considered by Desrochers *et al.* in 1988 [8].

1.4.3 Routing problems as Covering problems

Each of the vehicle routing variants discussed above involve finding a set of cycles such that every vertex in the graph is in one of the cycles. Consider an infinite class of graphs $\mathcal{H}$ (for example: trees, cycles, paths). A $\mathcal{H}$ *cover* of a graph $G = (V, E)$ is a set $\{H_i\}_i$ such that $\cup_i V(H_i) = V$. We then say that $\{H_i\}_i$ *covers* G.

The *makespan* of a cover is defined to be the total edge-length of the longest graph in the cover i.e. $max_{H\in\mathcal{H}} l(H)$ where l is the length function. Each graph in the cover can be thought of as a route for a vehicle. We must note that this change in terminology is highly effective in capturing the versatility of routing problems since it allows feasible solutions which are not restricted to being sets of cycles unlike the previously discussed problems.

An important problem considered in this essay (See Chapters 2 and 4) is that of tree-covers. Given a graph $G = (V, E)$, a *tree cover* is a set $\{T_i\}_i$ of trees such that $\{T_i\}_i$ covers G. We first mention the k-TREE COVER problem.

Definition 1.4.1. The k-TREE COVER (KTC) problem consists of a graph $G = (V, E)$, a metric $l : E \to \mathbb{N}$ and a given parameter $k \in \mathbb{N}$. A feasible solution to the problem is a set of at most k trees $\mathcal{T} = \{T_1, T_2, \cdots, T_k\}$ such that $\cup_i V(T_i) = V$. The objective of the problem is to find a tree cover of size at most while minimizing the makespan.

The problem was introduced by Arkin, Hassin and Levin and they provide a 4-approximation for the problem [1]. They study a variety of vehicle routing problems. In each case, a problem instance consists of a graph $G = (V, E)$ and edge lengths $l : E \to \mathcal{R}_+$. The length of a subgraph H of G is denoted by $l(H)(= \sum_{e\in E(H)} l(e))$. One variant is the MINIMUM TREE COVER problem. Here a problem instance satisfies the additional constraints that the edge lengths follow the triangle inequality and includes an addition parameter $\lambda > 0$ in the input. A feasible solution is a set of trees $\mathcal{T} = \{T_i\}_i$ such that $\cup_i V(T_i) = V$ and $l(T_i) \leq \lambda$ for every tree T_i. The objective is to minimize the cardinality of the cover $\mathcal{T}$. Thus the MINIMUM TREE COVER problem is the dual problem to KTC. Arkin *et al.* provide a 3-approximation for the problem. Khani and Salavatipour improve this to 2.5 in [25].

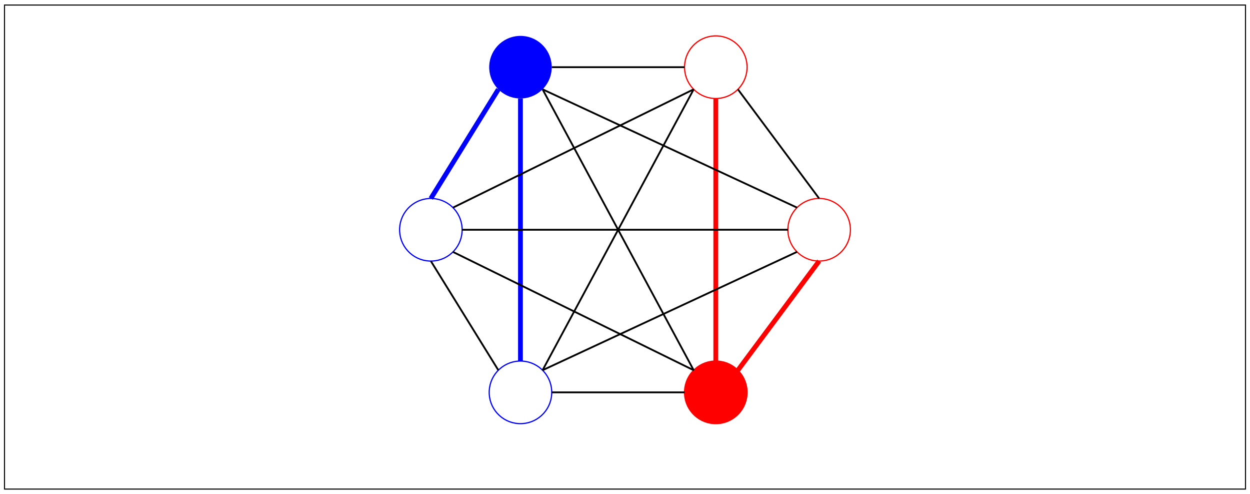

Figure 1.2: A Rooted Tree Cover for the complete graph on 6 vertices. The blue and red vertex constitute the set of roots and the corresponding coloured edges form the trees. Thus we have a cover of size 2.

ROOTED TREE COVER (RTC) is a variant of the k-TREE COVER problem introduced by Even *et al.* is [11].

Definition 1.4.2. Given a complete graph $G = (V, E)$, a metric $l : E \to \mathbb{N}$ and a set $R \subset V$, we call a set of tree, vertex pairs $\{(T_i, r_i)\}_i$ an R-rooted tree cover if the trees cover G and every tree T_i is rooted at a vertex in R. As in k-Tree Cover, the objective is to find a rooted tree cover for G while minimizing the makespan.

Note that for both KTC and RTC, the trees in the solution are edge-disjoint but not vertex-disjoint (See Lemma 3.1.1). Additionally, for two trees T_i and T_j in a solution for RTC, the root of T_i may be contained in T_j but they may not have the same root. Even *et al.* provide 4-approximation algorithms for both KTC and RTC. See Sections 3.1 and 3.2 for a detailed discussion of the two problems.

We also introduce a generalized tree cover problem called BUDGETED TREE COVER (BTC). Here the problem instance is a graph with metric edge-lengths and weights on the vertices. The goal is to find a tree cover where the trees are each rooted a vertex such that the total weight of roots does not exceed a given budget.

Definition 1.4.3. Consider a complete graph $G = (V, E)$ with a edge lengths $l : E \to \mathbb{N}$ (where l is a metric), weights $w : V \to \mathbb{N}$ and a budget $K \in \mathbb{N}$. A feasible solution to the problem is a set of tree, root pairs (called a cover) $T = \{(T_i, r_i)\}_i$ such that $\sum_{r_i \in R} w(r_i) \leq K$. In the Budgeted Tree Cover (BTC) problem, we want to find a feasible solution to the problem while minimizing the makespan.

A closely related problem is k-MST. The setting remains the same as KTC so we have a graph $G = (V, E)$ and metric edge lengths $l : E \to \mathbb{N}$. The objective is to find a subset of at least k vertices of a G whose Minimum Spanning Tree has least weight among all subsets of at least k vertices. The state of the art for k-MST is a 2-approximation by Garg [16]. It is primal-dual algorithm based on the work of Goemans and Williamson [17] that looks at k-MST from a set-covering perspective. More on this in Chapter 2. Note that a *Minimum set cover* instance consists of a groundset S, a family of subsets of S and a cost function $c : 2^S \to \mathbb{R}_+$. A feasible solution is a set of subsets from the given family so that every element of S belongs to some set in our solution. The objective is to find a minimum cost feasible solution.

1.5 Hardness of Routing Problems

Given a problem π and an algorithm A that solves the problem, we call A *efficient* if the time taken by A to produce a solution to any instance of π is bounded by a polynomial in the size of π. One notion of *size* is the length of the string that encodes π. In an ideal world, we would have an efficient algorithm for any problem that we come across. However, that is not the case. Though it is in no way trivial to prove this, it is not too hard to intuit. In Section 1.3 we stated that a feasible solution to TSP is any Hamiltonian cycle in the given graph. We also pointed out that for a given graph on n vertices there can be up to $n!/2$ Hamiltonian cycles. Thus if we checked even a fraction of the feasible solutions to minimize the cost, the time taken would be exponential (and hence not efficient) in n. Clearly, this would not make for a very fast algorithm. This brings us to a discussion of *computational complexity* of problems.

A *decision problem* is any *yes-or-no* question on a (possibly infinite) set of inputs (also called *instances*). For example, given an integer x, the question “Is x odd?” is a decision problem. We call an instance a *yes-instance* if the answer for it’s given decision problem is “yes” – in our example, all the integers that are not divisible by two are yes-instances. One of the most important and most natural questions to ask is whether or not one can answer a decision problem efficiently (which is another decision problem). The complexity class P is the set of all decision problems for which an efficient algorithm exists.

P’s counterpart (in no way a complement) is the set NP which the set of decision problems for which a solution can be *verified* in polynomial time. A verifier is an algorithm that takes a candidate solution and a problem instance as input and outputs a yes if the solution is feasible for the problem. Note that the running time of the verifier has to be polynomial in the size of the instance and not the certificate. For example given a graph and a cycle, we can decide whether the cycle is Hamiltonian in time polynomial in the size of the graph. We just traverse the cycle and see if every vertex is contained in the cycle. However, there is no known polynomial time algorithm that can decide whether or not a given graph contains a Hamiltonian cycle. Thus Hamiltonian cycle (HC) belongs to NP.

If two problems belong to NP, a natural question to ask is if one of them is more difficult than the other. A problem $A \in \mathsf{NP}$ is NP-complete if it is “at least as hard as” any other problem $B \in \mathsf{NP}$. We say that A is *at least as hard as* B if there is a polynomial time reduction from B to A. A polynomial time reduction from B to A that if we had a deciding algorithm for A then we could use it to solve B. Thus a polynomial time algorithm for any one NP-complete problem would imply a polynomial time decider for every NP-complete problem. We will now try to convey the difference between decision and optimization.

1.5.1 Decision and Optimization

All the problems considered in the above section are decision problems and the routing problems discussed are optimization problems. Consider an minimization problem M. We can construct a related decision problem $M^{'}$ where given an instance of M, we want to decide if it has a solution of cost less than some parameter k. Note that we can then solve $M^{'}$ using an algorithm for M (by the definition of minimization). So we can see that the decision problem is no harder than the corresponding optimization problem. This also brings us to the class fo NP-hard problems. We say that a problem A is NP-hard if there exists an NP-complete problem B such that there polynomial time reduction from B to A. Note that NP-hard problems are not restricted to be decision problems. Thus it is fair to say that an optimization problem is NP-hard.

The satisfiability problem k-SAT asks whether we can satisfy all the constraints (each a disjunction of k variables) in a given collection. This was the first problem that was proven to be NP-complete (by Stephen Cook and Leonid Levin in arguably the most important paper in theoretical computer science) [2]. Thus by definition of NP-completeness if there existed a polynomial time algorithm for k-SAT, we would have $\mathsf{P} = \mathsf{NP}$. This is the P versus NP problem and is the biggest unsolved problem in theoretical computer science.

The next section mentions results from the literature that establish the hardness of some vehicle routing variants discussed above. Also, regardless of whether or not $\mathsf{P} = \mathsf{NP}$ we need a way to route vehicles as efficiently as possible. One way around this is to give up on achieving optimality. Approximation algorithms do just that.

Definition 1.5.1. *Approximation algorithm:* Consider an optimization problem M and let Opt be the value of the optimal solution. If M is a maximization problem then an α-approximation algorithm always returns a solution of value at least Opt/α. If M is a minimization problem, the an α-approximation algorithm always returns a solution of value at least $\alpha\mathsf{Opt}$.

Here α is called the approximation ratio and is always at least 1. Note that the approximation ratio is the worst-case measure of the performance of an approximation algorithm. Obviously, it would be ideal to find the best possible approximation algorithm for any given problem. This would give us a lower bound on the approximation ratio. This brings us to a discussion of the hardness of approximation of optimization problems. In particular, we will focus on the Traveling Salesman and Vehicle Routing problems.

1.5.2 Hardness of Approximation of TSP and VRP

Earlier in this dicussion, we noted that we can think of TSP as a special case of VRP 1.4. The following is known for TSP [35]:

Theorem 1.5.2. *For any polynomial time computable function $\alpha(n)$ (where n is the size of the given TSP instance), TSP cannot be approximated within a factor of $\alpha(n)$, unless* $\mathsf{P} = \mathsf{NP}$.

Such problems are said to be hard to approximate. Note that this strong impossibility result is true when we look at TSP in it's most general form. However, if we restrict ourselves to Metric TSP (when the distances between the points obey the triangle inequality), the problem is no longer hard to approximate. In fact, an approximation factor of 3/2 can be acheived [35]. The hardness of approximating general TSP is the reason that the routing problems considered in this essay are based in a metric.

1.6 Outline

In the next chaper we will discuss certain variants of VRP in greater detail and touch upon the techniques that are used across the literature for solving these variants both combinatorial and LP-based set-cover like techniques. Chapter 3 forms the technical heart of the paper by talking about finding tree covers of graphs (See Section 1.4.3 for the defintion of a tree cover) and the work by Even *et al.* Chapter 4 is about our 5-approximation for a new generalization of tree covering. The last chapter (Chapter 5) attempts to present some ideas for further work based on the 2011 paper by Khani and Salavatipour [25] and a few of our own ideas for further work.

Chapter 2

Approximation Algorithms for Vehicle Routing and certain variants

We now survey the two main types of techniques used for approximating solutions to VRP and it's variants. We bifurcate techniques into two main categories based on the lens one uses to examine the problem. The first category consists of algorithms based on constructing minimum spanning trees and splitting them to create routes for each individual vehicle. The second category is based on set covering techniques and primal dual algorithms.

2.1 Tour-splitting techniques

We start off with describing approximation algorithms for Metric TSP. A problem instance consists of a graph $G = (V, E)$ and edge lengths $l : E \to \mathbb{N}$ that form a metric. The objective is to find a minimum cost Hamiltonian cycle in the graph. If the objective was to find some feasible cover, it could be achieved in polynomial time by constructing a minimum spanning tree. However, this does not give us a tour that visits every vertex exactly once. We then need to convert this tree into a cycle. Since we are in a metric, the algorithm uses the triangle inequality to great effect. Thus given two edges (u, v) and (v, w) we can always introduce a new edge (u, w) as long as $l(u, w) \leq l(u, v) + l(v, w)$ (i.e. the triangle inequality holds). This idea of introducing a new edge is known as *short-cutting* since the new edge introduced is no longer than the shortest path between u and w. Given a minimum spanning tree T of G, double every edge in the tree to get a graph where every vertex has even degree (better known as an *Eulerian* graph). Then, start traversing the tree from an arbitrary vertex and construct a tour by short-cutting every time we revisit have to a vertex. It is not too hard to see that this algorithm gives a 2-approximation for the problem [35]. The most important observation is that every edge in T is doubled and short-cutting can not lead to increase in cost.

2.1.1 Christofides' algorithm

Christofides' algorithm for the Metric Traveling Salesman Problem is a surprisingly easy to understand 3/2-approximation algorithm that is yet to be improved upon. It uses the important fact that in order to construct an Eulerian graph covering G, we only need to take care of vertices with odd degree. Thus Christofides' algorithm constructs a perfect matching on the odd-degree vertices of T and adds it to T. It can be proven that this perfect matching has cost no more than $\mathsf{Opt}/2$. Thus we get a 3/2-approximation algorithm for Metric TSP. This algorithm is also *tight*. This means that there are instances of Metric TSP such that you can do no better than a 3/2-approximation using Christofides' algorithm.

2.1.2 From tour to tours

The main difference between VRP and TSP is that we now want to construct a set of tours instead of a single tour. In a k-Vehicle Routing Problem (k-VRP) instance, a feasible solution is a set of k cycles $\{C_i\}_{i=1}^k$ such that $\bigcup_i V(C_i) = V$. The objective is to minimize the makespan of the cover. Intuitively, the idea of *tour-splitting* is to start with an Eulerian tour as constructed before and then break into k pieces of equal length. This idea was introduced by Frederickson, Hecht and Kim in [13] and they gave a 3-approximation algorithm for the problem of designing a solution to vehicle routing with k vehicles. This technique of tour-splitting is an elegant and powerful tool for designing solutions to vehicle routing problems. We will see tour-splitting based algorithms by Even *et al.* again in the next chapter.

2.1.3 Tours for non-identical vehicles

Another variation can be introduced to k-VRP by considering the real world idea that not all vehicles are identical. The length of a tour is then measured in terms of time i.e. $\frac{l(C)}{s}$ where s is the speed of the vehicle assigned to the tour described by cycle C. A problem instance of Heterogeneous Vehicle Routing (HetVRP) consists of a graph G with a special depot vertex r whose edge lengths form a metric and a set of k vehicles with different speeds. The objective is to find a set of tours (each containing r) covering G while minimizing the makespan. Thus when assigning a vehicle to a tour in an effort to minimize the makespan, one has to take care that a slow vehicle maybe not need to be assigned to a long route and a faster one to a short route since we could swap this assignment and acheive a smaller makespan.

This introduces a new aspect which is an important combinatorial optimization problem - assignment. An assignment problem instance consists of two types of objects. Each object of one type has a (not-necessarily strict) ordering associated with every object of the other type. The objective is to "match" the objects to each other such that every object is matched to it's most preferred choice.

2.1.4 Vehicle Routing with Capacities

Gortz *et al.* consider HetVRP in [21] and provide a constant factor approximation for the problem. A closely related problem that they also consider is HetCVRP (Heterogeneous Capacitated VRP) when all vehicles have a fixed capacity. The problem and the objective remains the same and they introduce a fixed capacity Q for each vehicle. Haimovich and Rinnooy-Kan showed that a ρ-approximation for HetVRP gives a $\rho + 1$ approximation for HetCVRP [23].

Hence, Gortz *et al.* focus on constructing an approximation for HetVRP. There are three main components of the algorithm. The first is to binary search for the optimal makespan. Assuming that they can find an upper bound M on the optimal makespan, they determine, for each vehicle, the vertices that the vehicle can visit in time M. The aspect of time and matching is taken care of using a result on *scheduling on unrelated parallel machines* by Lenstra *et al.* [29]. (The scheduling problem is that of distributing a set of jobs over a set of machines while minimizing the maximum completion time). The second part of the algorithm involves the construction of a special spanning tree (called a Level-Prim tree) rooted at r where the vertices are arranged in levels based on their distance from r. Finally, the Level-Prim tree is decomposed into a set of subtrees which are used by the first part as jobs to be scheduled on machines. As said before, their algorithm achieves an constant factor approximation.

2.2 Set-Covering based techniques

We discussed in Section 1.4.3 how routing problems could be perceived as covering problems. The most general (and in a way eponymous) example of a covering problem is Minimum-weight Set Cover. A problem instance of Minimum-weight Set Cover consists of a *groundset* $E = \{e_i\}_{i=1}^{n}$, a set $\mathcal{S} = \{S_j\}_{j=1}^{m}$ of subsets of E and a cost function $c : 2^E \to \mathbb{R}$ that assigns costs to each of the subsets in $\mathcal{S}$. A feasible solution is a *cover* $\mathcal{F} \subseteq \mathcal{S}$ such that each element of E is in some element of $\mathcal{F}$. An optimal solution is a feasible solution that minimizes the total cost of the elements in $\mathcal{F}$. The set cover problem is NP-hard and we discuss approximation algorithms for the problem in this section. The goal is to introduce a the *primal-dual* method of designing approximation algorithms.

Now let x_j be an indicator variable such that $x_j = 1$ if the set S_j is in our cover and 0 otherwise. Since we want every element e_i of E to be covered, it follows that we must pick at least one set containing e_i. This gives us the constraint that every feasible solution has to satisfy $\sum_{j:e_i \in S_j} x_j \geq 1$. Based on this we can write down the following linear programming relaxation which models the set-cover problem:

$$\begin{aligned} \min \quad & \sum_{j=1}^{m} x_j c_j & (P) \\ \text{s.t.} \quad & \sum_{j:e_i \in S_j} x_j \geq 1 \text{ for all } i \in \{1, 2, \cdots, n\} \\ & x_j \geq 0 \text{for all } j \in \{1, 2, \cdots, m\} \end{aligned}$$

The dual (D) for the above LP (P) is as follows:

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} y_i & (D) \\ \text{s.t.} \quad & \sum_{i:e_i \in S_j} y_i \leq c_j \text{for all } j \in \{1, 2, \cdots, m\} \\ & x_j \geq 0 \text{ for all } i \in \{1, 2, \cdots, n\} \end{aligned}$$

Here we can think of each y_i as a price that element e_i has to pay in order to be covered. The constraint in (D) dictates that in feasible dual solution, the total price paid by the elements in a given set S_j is no more than the cost of the set as given by c_j. One way to solve the problem would be to solve (P) to get a fractional optimal solution and then "round" the solution to obtain a solution to the set cover problem [36]. Another approach would be to solve the dual and then round the optimal dual solution. Then weak duality (See Appendix A) gives a solution to the set cover problem. Both of these approaches require solving a linear program and give an α-approximation algorithm for the set-covering problem where α is the maximum number of sets in which any element appears. This approximation guarantee can be matched by the primal-dual method which actually does not require us to solve an LP.

2.2.1 The Primal-Dual Method for Set Cover

The central idea of a general primal-dual algorithm is to start with a dual feasible solution which is then modified to increase the value of the dual objective function till we have the maximum possible number of *tight* dual constraints (i.e. constraints that hold with equality). Primal-dual optimization algorithms exist for several linear programming problems, network-flow problems and many problems related to vehicle routing (k-MST, shortest s-t path to mention two).

For the set covering problem, the algorithm starts with setting each dual variable y_i to 0 implicitly. Note that this gives us a feasible dual solution. The algorithm is best visualized as *growing* a region (think of a ball of radius y_i growing as we increment y_i) around each dual variable till the dual constraint corresponding to some set S_j becomes tight. The first time this would happen is when the regions corresponding to the elements in the cheapest set(s) (among all sets in $\mathcal{S}$) touch each other. The set S_j is then added to the cover and the dual variables corresponding to each element in S_j is then *frozen* i.e. for each element e_i in S_j, we are not allowed to increment the corresponding dual variable y_i any more. Note that more than one constraint can become tight at the same time. However every time we add a set to the cover, we are covering at least one new element. Thus this growing step is repeated at most n times where n is the number of elements in E.

To formalize this idea, consider a snapshot of the algorithm and the corresponding solution y to (D). Let $\mathcal{F}'$ be the set $\{S_j : \sum_{i:e_i \in S_j} y_i \leq c_j\}$ of sets such that their corresponding constraints are tight in y. If $\mathcal{F}'$ forms a cover (in other words, a feasible solution for the primal) then we are done. If not, then it means that there exists an element e_i that is not in any of the sets in $\mathcal{F}'$. So we can increase the corresponding dual variable y_i (there by also improving the value of our dual solution) by some positive amount so that the constraint for a cheapest set S_k containing e_i becomes tight. Thus we can add S_k to $\mathcal{F}'$ while maintaining the feasibility of the dual solution. As mentioned in the previous paragraph, this process will be repeated at most n times since we cover at least one new element each time we add a set to our cover. We now discuss primal-dual algorithms for *Prize-collecting Steiner Tree* problem and then tie it in with approximation algorithms for the k-MST problem.

2.2.2 Prize-Collecting Steiner Tree

Recall that a minimum spanning tree of a graph is the cheapest tree that contains all the vertices of the graph. Now consider a graph $G = (V, E)$ with non-negative costs on the edges given by $c : E \to \mathbb{R}_{\geq 0}$. The set of vertices is partitioned into the set T of *terminals* and the non-terminals (also called *Steiner* vertices) $V \setminus T$. A feasible solution (called a *Steiner tree*) is a tree that spans all the terminals. The objective is to find a minimum cost Steiner tree i.e. a tree spanning the terminals with least possible total edge-cost. Each of the terminals can be thought of as a location that we need to visit and each Steiner vertex represents a depot that we can set up for vehicles. So a solution to the *Minimum-cost Steiner Tree* problem gives tells us which depots we need to set up so that we are covering each of the terminals in the cheapest possible way while ensuring that each location is connected to some depot.

The Prize Collecting Steiner Tree (PCST) problem introduces a penalty π_i (the *prize*) on each terminal vertex i and then any Steiner tree gives us a feasible solution to PCST. An optimal solution to PCST is now a tree T that maximizes the weights of the terminals covered by T (in other words minimizes the penalty incurred due to the vertices not in our solution) and achieves the lowest possible edge-cost. Formally the objective is to minimize $\sum_{v \in V(T)} c_e + \sum_{w \in V \setminus V(T)} \pi_w$ over all feasible solutions T to the problem. One can extend the vehicle routing analogy here since the penalties now represent the profit to be made by serving a location. Thus an optimal prize-collecting Steiner tree connects the set of locations such that the profit is maximized while minimizing the total cost of connecting these locations.

Goemans and Williamson gave a general primal-dual algorithm which attains a 2-approximation for PCST [17]. They considered a rooted version of the problem where a root r is prespecified. This is without loss of generality since the algorithm can be run once for each vertex. We present a sketch of the algorithm based on the discussion by Blum *et al.* [4]. Say we are given a graph $G = (V, E)$ with edge costs $c : E \to \mathbb{R}_{\geq 0}$, a root r and penalties $\pi : T \to \mathbb{R}_{\geq 0}$ on the terminals. Then the linear programming relaxation for PCST is as follows:

$$\begin{aligned}
\min \sum_{j=1}^{m} x_e c_e + \sum_{v \neq r} (1 - z_v)\pi_v & & (P - PC)\\
\text{s.t.} \qquad \sum_{e \in \delta(S)} x_e \geq z_v & \text{ for all } v \in S; r \notin S\\
x_e \geq 0 & \qquad \text{for all } e \in E\\
z_v \geq 0 & \qquad \text{for all } v \in V
\end{aligned}$$

The corresponding dual linear program is:

$$\begin{aligned}
\max \qquad \sum_{S : r \notin S} y_S & & (D - PC)\\
\text{s.t.} \quad \sum_{S : e \in \delta(S)} y_S \leq c_e & \qquad e \in E\\
\sum_{S \subset T} y_S \leq \sum_{v \in T} \pi_v & \text{ for all } T \subset V; r \notin T\\
y_S \geq 0 & \text{ for all } S \subset V; r \notin S
\end{aligned}$$

Here y_S is an indicator variable denoting whether or not a subset S of vertices is covered by the solution. The algorithm proceeds in two phases. The first phase is the *growth phase*. Intially all subsets S are set to be *active*. In each step, y_S's are uniformly increased for all active components by some $\epsilon > 0$. If an edge constraint goes tight then add the edge to our forest F. If a set constraint for some set S becomes tight, freeze the corresponding dual variable and set the component to be inactive. When a set S is made inactive, every vertex in S is labeled with all of the elements in S. The second phase is the *pruning* phase where all inessential edges from F are removed while maintaing primal feasibility.

This algorithm was used by Blum *et al.* [4] to give a constant factor algorithm for the k-MST problem. A k-MST instance consists of a graph G with non-negative weights on the edges and a given integer k. A feasible solution is a tree spanning k vertices. The objective is to find a feasible solution minimizing total edge cost. The problem was shown to be NP-hard by Ravi *et al.* [31] and independently by Fischetti *et al.* [12] and also by Zelikovsky and Lozevanu [37]. Blum *et al.* 's algorithm was improved in a series of papers by Garg [15], Arora and Karakostas [3] and Garg [16] to 3, $2+\epsilon$ and finally to 2 respectively. Each of the algorithms were mainly primal-dual algorithms and then the results strengthened via careful implementation.

Chapter 3

Tree Covers of Graphs

Tour-splitting (Section 2.1) describes how one can easily convert a tree into a cycle (in a metric setting) and hence, a tour by short-cutting. This chapter focuses two of the problems from a paper by Even *et al.* [11]. The problems of k-tree cover and Rooted Tree Cover are examined in full detail and then the ideas are extended to solve Budgeted Tree Cover.

3.1 Algorithm for k-Tree Cover

Let us review the k-Tree Cover problem. A problem instance consists of a given complete graph $G = (V, E)$ with edge-lengths $l : E \to \mathbb{N}$ (where l is a metric) and a parameter $k \in \mathbb{N}$. A feasible solution is a tree cover $\mathcal{T}$ of G, where $|\mathcal{T}| \leq k$. The objective is to find a tree cover while minimizing the makespan of the cover.

Consider the case when the optimal makespan is equivalent to the length of a minimum spanning tree of G. If we knew this was the case, we could return any minimum spanning tree as a cover (as k has to be at least one). We can generalize this by saying that if we know the optimal makespan to be an α fraction of the length of the minimum spanning tree, then ideally the solution would be to construct a minimum spanning tree and break it up into α pieces “carefully” (See Lemma 3.1.1). This solution would be feasible as well since given our assumption for the optimal makespan, k can be no smaller than an α fraction of the length of a minimum spanning tree of G.

The first part of Even *et al.* ’s algorithm is to guess a value B for the makespan and remove all edges e from G such that $l(e) > B$ (possibly disconnecting the graph in the process). Since these edges are more expensive than the makespan, they will not be a part of any feasible solution with makespan B. Hence, this edge removal does no damage. In the next step, find a minimum spanning tree for each connected component of the graph. Now that we have our minimum spanning trees for each component, we try to decompose them into smaller pieces based on our guess for the makespan. Intuitively, the process can be explained as follows. One can think of the decomposition procedure as traversing each minimum spanning tree while keeping track of the total edge-length traversed. Once this length reaches a certain limit, we disconnect the traversed area (which is a subtree of the minimum spanning tree) from the tree by removing the next edge. Since we will binary search for the optimal makespan, at a given point we are at most a factor of 2 away from the optimal. Since the furthest point in a optimal solution is B away, doubling the edges to get a tour gives a lower bound of $2B$ on the length of the tree. We will first look at each of the individual ideas in detail and then tie them together to form the algorithm. We will also see that their algorithm gives us a 4approximation for the makespan.

3.1.1 Edge-decomposition of Trees

In their algorithm, Even et al. take a large tree and break it up into smaller subtrees such that the total edge length of each resulting subtree is within the range $[2B, 4B)$ where B is the current guess for makespan. This helps ensure that the makespan of the constructed cover is not too large. The following lemma by Khani and Salavatipour [25] gives a general algorithm to decompose a tree based on a given parameter β. It was proven implicitly in earlier work by Even *et al.* [11].

Lemma 3.1.1. *[Khani and Salavatipour [25]]Given a tree T with weight $l(T)$ (sum of the weights of the edges in T) and a parameter $\beta > 0$ such that all the edges of T have weight at most β, we can edge-decompose T into trees $\{T_i\}_{i=1}^{k}$ with $k \leq max(\lfloor \frac{l(T)}{\beta} \rfloor, 1)$ such that $W(T_i) \leq 2\beta$ for each $1 \leq i \leq k$.*

Proof. For any vertex $v \in V(T)$ let $v_1, \cdots, v_p$ be v's children connected via edges $e_1, \cdots, e_p$. Let $T^{'} = \cup_{i=a}^{b} T_{e_i}$ (for $a, b \in [p]$) be a subtree rooted at v. Then, *splitting away* $T^{'}$ from T is defined to be the the following procedure: Designate $T^{'}$ as a new part and remove all edges of $T^{'}$ from T. Now, T only contains vertices that are still connected to the root of T. Note that we only delete the edges of T, thus T and $T^{'}$ are edge-disjoint but not vertex disjoint.

Root the tree T at an arbitrary vertex $r \in V(T)$. For every vertex $v \in V$, denote by T_v the subtree rooted at v. Consider an edge $e = (u, v)$ where u is the parent of v in T. Use T_e to denote the subtree that contains the vertex u, the tree T_v and the edge e. A tree T is called *medium* if $l(T) \in [\beta, 2\beta)$, *heavy* if $l(T) \geq 4\beta$ and *light* otherwise.

Note that we can always split away a medium tree and put it in the decomposition. So assume all the subtrees of T are either heavy or light. Suppose T_v is a heavy subtree whose children are connected to v by edges $e_1, e_2, \cdots$ such that all subtrees $T_{e_1}, T_{e_2}, \cdots$ are light. Let i be the smallest index such that $T^{'} = \cup_{j=1}^{i} T_{e_j}$ has weight at least β. Thus $T^{'}$ is a medium subtree by construction. Then split away $T^{'}$ from T and repeat the process until there is no heavy subtree of T. This process is used repeatedly to"split" away trees till the weight of the remaining part of the tree is less than β.

If $l(T) \leq 2\beta$ then do not split away any tree since it is already a light or medium weight tree and the lemma holds trivially. Suppose the split trees are $T_1, ..., T_k$ with $k \geq 2$ and $l(T_i) \in [\beta, 2\beta)$ for $1 \leq i \leq k$. The only tree that may have weight less than β is T_k. We need to show that $k \leq \lfloor \frac{l(T)}{\beta} \rfloor$. For this, note that when T_{k-1} is split away, the weight of the remaining tree is at least 2β (otherwise we would only have $k-1$ trees in our decomposition). Hence, the average weight of all trees in the decomposition is at least β which proves that k cannot be greater than $\lfloor \frac{l(T)}{\beta} \rfloor$. □

Based on the above lemma, we can write down the following subroutine for edge decomposition. Algorithm Edge-decompose will be used as a subroutine later in the algorithms for Rooted Tree Cover (See Section 3.2) and Budgeted Tree Cover (See Section 3.2).

Algorithm 1 Edge-decompose(T, β) - Returns a set of k subtrees $\mathcal{T} = \{T_i\}_i$ such that $W(T_i) \leq 2\beta$ for each $1 \leq i \leq k$ and $k \leq max(\lfloor \frac{l(T)}{\beta} \rfloor, 1)$
.

```
1: **if** $l(T) \leq 2\beta$ **then**
2:     **Return** $\mathcal{T} = \{T\}$
3: **end if**
4: Initialization: Root $T$ at an arbitrary vertex $r \in V(T)$. $\mathcal{T} \leftarrow \emptyset$
5: **while** There exists a heavy subtree of $T$ **do**
6:     Find a heavy subtree $T_v$ such that all it's children subtrees $T_{e_1}, T_{e_2}, \cdots$ are light.
7:     $i \leftarrow$ smallest index such that $T^{'} = \bigcup_{j=1}^{i} T_{e_j}$ has weight at least $\beta$.
8:     Remove all edges of $T^{'}$ from $T$.
9:     $\mathcal{T} = \mathcal{T} \cup \{T^{'}\}$
10: **end while**
11: **Return** $\mathcal{T}$.
```

Algorithm Edge-decompose splits away medium trees till there are no heavy trees left. It is possible that we have one light tree left over after we split away all medium weight trees. So the set $\mathcal{T}$ returned by this algorithm is of the form $S \cup L$ where L is a set containing at most one light tree. Edge-decompose will be used again for the algorithms for Rooted Tree Cover (See Section 3.2) and Budgeted Tree Cover (See Section 3.2). Since the edge decomposition procedure remains the same, we only change the value of the parameter β as required. Now that we know how the trees can be decomposed, we need a way to guess the optimal makespan. For ease of explanation, we will assume that we have an upper bound on the optimal makespan and state the algorithm by Even *et al.* Once we have proven that the algorithm works under this assumption, we will see how to circumvent it.

3.1.2 What if we knew the optimal makespan?

For now, assume that we have an upper bound B on the optimal makespan B^*. Hence, $B \geq B^*$. As we will be guessing values for B, Lemma 3.1.2 can be used to verify if B is actually an upper bound on the optimal makespan. Given $G = (V, E)$, k and $B \geq B^*$, Even *et al.* do the following:

Algorithm 2 Construct k-TreeCover(G, k, B) - Compute a k-tree cover of G with cost at most $4B$

```
1: Remove all edges of length $> B$. Let $\{C_i\}_i$ be the connected components of the graph
   formed after deleting heavy edges.
2: for all $i$ do
3:     $M_i \leftarrow$ minimum spanning tree of $C_i$.
4:     $\{S^i_j\}_j \leftarrow$ medium trees from Algorithm Edge-decompose$(M_i, 2B)$ and $\{L_i\}_i \leftarrow$ any
   leftover light trees. (See Lemma 3.1.1)
5: end for
6: Return$\{S^i_j\}_{i,j} \cup \{L_i\}_i$ as the tree cover.
```

We now step through Algorithm Construct k-TreeCover. Since no tree is allowed to be of length more than B, remove all edges e such that $l(e) > B$. Then, compute a minimum spanning tree M_i for each connected component of the graph resulting from Step 1. For the fourth step, use Algorithm Edge-decompose (Algorithm 3.1.1) with $\beta = 2B$ to obtain the required decomposition. Let us try to see how many trees we are using to cover each component C_i. Use k_i to denote the number of trees T_i in the output of Algorithm Construct k-TreeCover such that $V(T_i) \cap V(M_i) \neq \emptyset$. Since we are splitting away trees of size at least $2B$, the number of trees obtained is $\lfloor \frac{l(M_i)}{2B} \rfloor + 1$. Note that we are yet to see why the number of trees thus obtained is no larger than k. Lemma 3.1.2 shows that it is indeed that case.

Let B^* denote the makespan of an optimal k-Tree cover of G. Consider $\mathcal{T}^* = \{T^*_1, T^*_2, \cdots, T^*_k\}$, an optimal k-Tree cover solution. Note that since $B \geq B^*$, T^* does not use any edges of weight greater than B. Let k^* be the number of trees from $\mathcal{T}^*$ covering the component C_i. Then Even et al. prove the following:

Lemma 3.1.2. *If $B \geq B^*$, then $k_i + 1 \leq k^*$.*

Proof. Proof. Let $T^*_1, T^*_2, \cdots, T^*_{k_i}$ be the trees from an optimal solution covering G_i . We can connect these k^*_i components together to form a single component using k^*_i edges. Since the components were formed by deleting edges of weight $> B$, each connecting edge that we add costs at most B. Hence, we obtain: $\sum_{j=1}^{k^*_i} l(T^*_i) + (k^*_i - 1) \cdot B \geq l(M_i)$. Since each tree in the optimal solution has edge weight at most B, we get $k^*_i \geq \frac{l(M_i)}{2B} + \frac{1}{2}$. Since $k_i = \lfloor \frac{l(M_i)}{2B} \rfloor \leq \frac{l(M_i)}{2B}$, the lemma follows. □

Lemma 3.1.2 shows that Algorithm Construct k-TreeCover succeeds when $B \geq B^*$. Note that the contrapositive of the above lemma immediately gives us the following:

Corollary 3.1.3. *If $\sum_i (k_i + 1) > k^*$, $B < B^*$.*

The optimal makespan is not actually known beforehand and hence, we need to know how to circumvent this difficulty. The TreeCover algorithm binary searches for the optimal makespan and starts off with a low guess B. Using Lemma 3.1.2 it first verifies if the guess is sufficiently large. If not, it *doubles* the guess and repeats till the premise of Lemma 3.1.2 holds. It then uses Algorithm Edge-decompose as a subroutine, to construct the tree cover. Even *et al.* 's algorithm for k-tree cover is presented below.

Algorithm 3 TreeCover(G, k, B) - Compute a k-tree cover of G with cost at most $4B$

```
1: Remove all edges of length greater than $B$. Let $\{C_i\}_i$ be the connected components of
   the graph formed after deleting heavy edges.
2: for all $i$ do
3:     $M_i \leftarrow$ minimum spanning tree of $C_i$.
4:     $k_i \leftarrow \lfloor \frac{l(M_i)}{2B} \rfloor$.
5:     if $\sum_i (k_i + 1) > k$ then
6:         Print: "$B$ is too low".
7:         Return: TreeCover$(G, k, 2B)$
8:     end if
9:     $\mathcal{T} \leftarrow$ AlgorithmEdge-decompose$(M_i, 2B)$
10:    $\{S^i_j\}_j \leftarrow \{T \in \mathcal{T}$ such that $l(T) \geq 2B\}$ and $\{L_i\}_i$ gets any leftover light trees.
11: end for
```

Lemma 3.1.4. *When successful, Algorithm* TreeCover *returns a k-Tree cover for G with makespan at most $4B$.*

The length of each of the subtrees returned by the decomposition is in the range $[2B, 4B)$. So Algorithm TreeCover guarantees a 4-approximation to k-TreeCover. Notice that the approximation guarantee is given in terms of our estimate B and not the optimal makespan B^*. Thus, the quality of our algorithm hinges on how well B approximates B^*. We now show that given any $\epsilon > 0$ we can find a B such that $B = B^* + \epsilon$.

Theorem 3.1.5. *For every ϵ, there is a $(4 + \epsilon)$ approximation algorithm for k-Tree cover that runs in time polynomial in the size of G and log$(\frac{1}{\epsilon})$.*

Proof. Let $n = |V|$. We need to find a $(4 + \epsilon)$-approximation algorithm for minimum tree cover that runs in time polynomial in the size of the graph and in $\log(\frac{1}{\epsilon})$. Let $l_1 \leq l_2 \leq \cdots \leq l_m$ be the edge weights sorted in non-decreasing order. Now it is clear that $B^* < n \cdot l_m$ as l_m is the length of the longest edge and n is the number of vertices. If Algorithm TreeCover reports $B^* < B = l_m$, then the total weight of all edges of weight less than $\frac{l_m \epsilon}{n^2}$ is less than $\epsilon \cdot B^*$. Since these edges are not sufficient, we can contract these edges and only use edges of weight at least $\frac{\epsilon \cdot l_m}{n^2}$. Binary searching in the range $[\epsilon \cdot l_m/n^2, n \cdot l_m]$ can be done in polynomial time.

If Algorithm TreeCover does not fail with $B = l_m$, then let i be an index such that (i) it reports $B < B^*$ for $B = l_i$, and (ii) finds a tree cover of cost $4B$ for $B = l_i + 1$.

Here, we can conclude that $B^* \in (l_i, 4l_i + 1]$. Binary searching in the range $(l_i, l_i + 1]$ is polynomial if $\frac{l_{i+1}}{l_i} \leq \frac{n^2}{\epsilon}$. If it is not the case, then to maintain polynomial running time, we run the algorithm with $B = l'$ where $l' = \frac{n^2}{\epsilon} \cdot l_i$. If the algorithm finds a tree cover of makespan at most $4l'$, then binary searching within the range $[l_i, l']$ is strongly polynomial. Else, TreeCover returns "B is too low" in which case we contract all edges of weight less than $l_i + 1$ and consider only edges of weight at most $4 \cdot l_i + 1$ i.e. edges with weights in the range $[w_i + 1, 4 \cdot w_i + 1]$. Note that binary searching in this range is again, strongly polynomial. □

Combining Lemmata 3.1.1 and 3.1.3 with this, we have proved that we get a $(4 + \epsilon)$-approximation. If the edge weights are polynomial, we get a 4-approximation.

This completes our discussion of Even *et al.* 's result for k-Tree covers. In the same paper [11], Even *et al.* looked at Rooted Tree Cover (See Definition 1.4.2). This is the other important algorithm on which our result for Budgeted Tree Cover hinges.

3.2 Rooted Tree Cover

Recall that given a complete graph $G = (V, E)$, a metric $l : E \to \mathbb{N}$ and a set $R \subset V$, we call a set of trees $\{T_i\}_i$ an R-Rooted Tree Cover (See Definition 1.4.2) if the trees cover V and every tree T_i is rooted at some vertex in R. The objective is to find a rooted tree cover for G while minimizing the makespan of the cover. Even *et al.* provide a 4-approximation for RTC [11]. We proceed just like we did in the previous section. The first part of the algorithm uses Edge-decompose (Algorithm 3.1.1) to construct a tree cover for G. The second part takes care of rooting each tree in the cover.

This rooting condition introduces a new facet of assignment to the problem. Say the optimal makespan was $B^* > 0$. Consider a tree of edge weight $B^* - \epsilon$ (for some small $\epsilon > 0$) that we want to add to the optimal cover. Then for the solution to remain feasible, the assigned root can be no farther than ϵ from the tree. Similarly, for a very small tree (say of total weight ϵ), the root can be as far as $B^* - \epsilon$. This problem is solved using bipartite matching techniques (discussed in Section 1.2.1) where we try to match trees to roots that are not too far away.

3.2.1 Edge decomposition to construct covers

In the Rooted Tree Cover problem, the distance between a tree and it's corresponding root also contributes to the makespan of a solution to RTC. Hence, this time the input parameter β for Edge-decompose will be exactly our guess for the makespan unlike KTC where it was twice the guess. Thus given a tree T and a guess B for the makespan, we run Edge-decompose(T, B) to get a set of trees covering T such that the length of each tree in the cover is in the range $(B, 2B]$. Based on the proof of Lemma 3.1.1 the following lemma holds.

Lemma 3.2.1. *Given a tree T with weight $l(T)$ and an upper bound $B > 0$ on the weight of every edge $e \in E(T)$, we can edge decompose T into trees $\{T_i\}$ such that $l(T_j) < B$ and $l(T_i) \in [B, 2B)$, $\forall i = j$.*

The main difference is the definition of light, medium and heavy trees. A tree T is called *medium* if $l(T) \in [B, 2B)$, *light* if $l(T) < B$ and *heavy* otherwise. We then split away medium weight trees just as we did previously till there are no heavy subtrees left. Thus for each tree we obtain a cover consisting of at most one light subtree and one or more medium subtrees.

3.2.2 Assigning Roots to Trees

Consider an instance of Rooted Tree Cover as defined in Section 1.4.2. Let $\mathcal{T} = \{T_i\}_i$ be a set of trees returned by Algorithm Edge-decompose(T_i, B). Define the distance between a tree $T \in \mathcal{T}$ and a vertex $v \in V(T)$ to be $d(u, v)$ where $d(u, v)$ is the length of the shortest path between u and v. B is a given parameter such that a tree maybe rooted at a vertex only if they are at most B away. To assign a root from some $R \subset V$ to each tree in T , we construct a bipartite graph $H = (\mathcal{T} \cup R, E)$ where $\mathcal{T}$ contains a vertex for each $T \in \mathcal{T}$ and R. Put an edge (of unit cost) between a tree $T \in \mathcal{T}$ and a vertex $r \in R$ if the distance between T and r is at most B. Thus the problem of assigning roots to trees is equivalent to finding a $\mathcal{T}$-saturating matching. So we set up and solve the matching LP P' for H (see Section 1.2.1). The above discussion gives an algorithm that takes in a set of trees and a set of candidate roots and assigns roots to trees. AssignRoots$(\mathcal{T}, R, B)$ presents the discussion in the form of an algorithm.

Algorithm 4 AssignRoots$(\mathcal{T}, R, B)$ - Returns a set of root-tree pairs $\{(T_i, r_i)\}_i$ such that $d(T_i, r_i) \le B$ for all i.

```
1: Construct a bipartite graph $H = (\mathcal{T} \cup R, E)$ where $\mathcal{T}$ contains a vertex for each $T \in \mathcal{T}$
2: Add edges of unit cost between every tree-root pair $T \in \mathcal{T}$ and $r \in R$ if the distance
   between $T$ and $r$ is at most $B$.
3: Find a minimum cost $\mathcal{T}$-saturating matching $\mathcal{M}$ in $H$. (If no such matching exists
   $\mathcal{M} \leftarrow \emptyset$)
4: Return$\mathcal{M}$.
```

Even *et al.* prove that when B is at least the makespan of an optimal solution for Rooted Tree Cover, the matching LP P' has a feasible solution. The integrality of the matching polytope for bipartite graphs then implies that we can find a $\mathcal{T}$-saturating matching and hence, the desired assignment of roots to trees. More formally: Let B^* denote the makespan of an optimal R-rooted tree cover of G as before. Then, given a decomposition where $\{S^i_j\}_{i,j}$ are the medium weight subtrees, the following is true:

Lemma 3.2.2. *If $B \geq B^*$, then there exists a matching in the bipartite graph H that matches every subtree in $\{S_i\}_{i,j}$ to some root in R.*

Proof. To show the existence of a matching, we use Hall's Theorem (See Theorem 1.2.1). Thus we need to show that for any subset $\mathcal{S}$ of $\mathcal{T}$, the set of neighbors $N(\mathcal{S})$ is at least as large as $\mathcal{S}$. On a high level, the idea is to show that given any fixed optimal solution, we can cover all the trees obtained in our decomposition using only the roots from the optimal solution. This would clearly give us a feasible solution. We will do the proof in two parts:

Part 1: Fix an optimal solution $\mathcal{T}^* = \{T_i^*\}_{i=1}^k$ with roots $R^* = \{r_i^*\}_i \subseteq R$. Consider now a subset $\mathcal{S}$ of trees from $\mathcal{T}$. By construction, every tree $S \in \mathcal{S}$ satisfies $l(S) \in [B, 2B)$. Let $\mathcal{T}^*(\mathcal{S})$ be the set of trees from the optimal solution that cover vertices in $\mathcal{S}$ i.e. $\mathcal{T}^*(\mathcal{S}) = \{T^* \in \mathcal{T}^* : \exists S \in \mathcal{S} \text{ such that } S \cap T^* \neq \emptyset\}$. If any tree T^* (rooted at a vertex r) from the optimal solution intersects a tree S from $\mathcal{S}$, then the distance between r and S is at most B^* and by assumption, at most B. Hence, there is an edge between r and S in our bipartite graph H. This implies that $|N(S)| \geq |\mathcal{T}^*(\mathcal{S})|$. Thus, if we can prove that $|T^*(S)| \geq |S|$, we are done. We will prove this as follows:

Part 2: Since $\mathcal{T}^*(\mathcal{S})$ is a subset of the fixed optimal solution, every tree from $\mathcal{S}$ is connected to some root from the optimal solution via a tree from $\mathcal{T}^*(\mathcal{S})$. By our construction, every edge in the trees in $\{S_j^i\}$ is also an edge from some minimum spanning tree. We construct a new graph M' by deleting all edges of $\mathcal{S}$ from the minimum spanning tree and adding the edges from $\mathcal{T}^*(\mathcal{S})$. In *Part 1* we established that each tree is connected to a root via the edges of $\mathcal{T}^*(\mathcal{S})$. Thus every vertex in M' is connected to some root r in R^*. So if we identify all the roots to a single vertex, then the subgraph obtained is connected. Since this is a connected subgraph spanning all the non-root vertices, the total edge-weight is at least the edge-weight of the minimum spanning tree. Hence, $l(M') \geq l(M)$ where M is the minimum cost tree spanning the non-root vertices.

Let $l(\mathcal{T}^*(\mathcal{S}))$ denote $\sum_{T^* \in \mathcal{T}^*(\mathcal{S})} l(T^*)$ and $l(\mathcal{S})$ denote $\sum_{S \in \mathcal{S}} l(S)$. Then, $l(M') \geq l(M)$ implies $l(\mathcal{T}^*(\mathcal{S})) \geq l(S)$. Note that this is simply because $\mathcal{T}^*(\mathcal{S})$ and $\mathcal{S}$ were constructed by decomposing M' and M respectively. Observe that the total length $l(\mathcal{T}^*(\mathcal{S})) \leq B^*|\mathcal{T}^*(\mathcal{S})|$. Therefore, $B^*|\mathcal{T}^*(\mathcal{S})| \geq l(\mathcal{T}^*(\mathcal{S})) \geq l(\mathcal{S}) \geq B \cdot |\mathcal{S}|$. Now, since $B \geq B^*$, it follows that $|\mathcal{T}^*(\mathcal{S})| \geq |\mathcal{S}|$.

Parts 1 and *2* prove $|N(\mathcal{S})| \geq |\mathcal{T}^*(\mathcal{S})|$ and $|\mathcal{T}^*(\mathcal{S})| \geq |\mathcal{S}|$ respectively. Hence, we have $|N(\mathcal{S})| \geq |\mathcal{S}|$. □

Following is the algorithm given by Even *et al.* for the Rooted Tree Cover problem. It takes the graph G, the set of roots R and a guess B for the makespan as input and, outputs an R-Rooted Tree Cover for G. Note that we are again assuming that we can obtain a guess $B \geq B^*$ where B^* denotes the makespan of an optimal solution for Rooted Tree Cover.

Steps 1 through 3 of the algorithm are self-explanatory. For step 4, we use Lemma 3.2.1 and thus successfully decompose each of the trees into subtrees of edge length at most $2B$.

Algorithm 5 RootedTreeCover(G, R, B) - Compute a Rooted tree cover of G with cost at most $4B$

```
1: Initialization: $\mathcal{T} \leftarrow \emptyset$
2: Remove all edges of length greater than $B$. Let $\{C_i\}_i$ be the connected components of
   the graph formed after deleting heavy edges.
3: $M \leftarrow$ minimum spanning tree of graph obtained from $G$ by contracting roots in $R$ to
   a single node.
4: for all $i$ do
5:     Let $\{T_i\}_i$ be the set of trees formed by uncontracting $R$.
6:     $\mathcal{T} \leftarrow \mathcal{T} \bigcup$ AlgorithmEdge-decompose$(M_i, B)$
7:     $\mathcal{M} \leftarrow$ AssignRoots$(\mathcal{T}, R, B)$
8:     if $\mathcal{M} == \emptyset$ then
9:         Print: "$B$ is too low".
10:        Return: RootedTreeCover$(G, R, 2B)$
11:    end if
12: end for
13: Return: $\mathcal{M}$
```

In Step 6, we set up a bipartite matching problem and use it's solution to find a root $r \in R$ for every tree. The existence of a solution to this matching problem is guaranteed by our assumption $B \geq B^*$ and Lemma 3.1.2.Just as before, Algorithm RootedTreeCover can be modified to reach a correct estimate for the makespan. We start with a low value for B. If the LP set up in Step 6 is not feasible, we return "Algorithm failed because $B < B^*$". (See Lemma 3.2.2) In this case, we double our guess B for the makespan and try again. . Otherwise, we continue and return the decomposition found by the algorithm. It is now easy to see that this algorithm returns a 4-approximation for k-Tree Cover. This is because the length of each of the subtrees returned by the decomposition is in the range $[B, 2B)$. Hence, the following lemma holds:

Lemma 3.2.3. *When Algorithm Rooted-Tree-Cover is successful, it finds an R-rooted tree cover of cost at most $4B$.*

Proof. By construction, each tree in the cover given by the algorithm has a distinct root in R and all the nodes are covered (i.e. each node belongs to at least one tree). The weight of each tree used in the matching problem lies in the range $[B, 2B)$. The distance of the roots assigned to the trees is at most B. Finally, the weight of each of the light trees is no more than B and there is at most one light tree attached to a medium tree. Thus, no tree in our cover has an edge weight greater than $4B$. □

Note that if the edge weights are not polynomial, we can again use an approach identical to Theorem 3.1.5 to obtain a $(4 + \epsilon)$-approximation. We state it formally here:

Theorem 3.2.4. *For every $\epsilon > 0$, there is a $(4+\epsilon)$-approximation algorithm for the Rooted Tree Cover that runs in time polynomial in the size of G and $log(\frac{1}{\epsilon})$.*

Now that we have seen the two algorithms that our result is based on, we move on to Budgeted Tree Cover. In the following chapter, we present a 5-approximation algorithm for the problem. As stated before, the algorithm combines ideas from both the k-Tree Cover and Rooted Tree Cover problems.

Chapter 4

Budgeted Tree Cover

Let us look back at the k-Tree Cover problem. Instead of thinking that we have a parameter k that limits the number of trees, k can be thought of as a budget. Then we pay a unit cost for rooting each tree in a cover and we are not allowed to exceed the budget. Hence, we will not be able to use more than k trees in our cover. We generalize this to a version where rooting costs are non-uniform.

This generalization can be tied back to the Nurse Station Location problem. The Nurse Station Location problem consists of a set of possible locations to be set up as nurse stations. There is a cost associated with setting up a nurse station at a given location. The goal is to find a set of locations so as to set up these stations as cheaply as possible while ensuring that each patient is visited by some nurse. We can think of each of the patient locations and possible nurse locations as vertices in a graph. The distances between different locations form the edge weights, the cost of setting up a station at a given location gives us weights on the vertices of the graph and the budget forms the final parameter. A feasible solution is a set of rooted trees that cover all the vertices of the graph. A rooted tree can be converted into a route for the nurse by shortcutting (See Section 2.1) and the root of the tree represents the location of a nurse's station. More traditionally, one can think of a vehicle routing problem where we pay non-uniform costs for the vehicles and are given a fixed budget. A feasible solution is an assignment of vehicles to routes such that the total cost of our vehicles does not exceed the budget. An optimal solution is a feasible solution with the least makespan.

An instance of the problem consists of a complete graph $G = (V, E)$ with a edge lengths $l : E \to \mathbb{N}$ (where l is a metric), weights $w : V \to \mathbb{N}$ and a budget $K \in \mathbb{N}$. A feasible solution to the problem is a set of tree, root pairs (called a cover) $T = \{(T_i, r_i)\}_i$ such that the total weights of the roots does not exceed K. In the Budgeted Tree Cover (BTC) problem, we want to find a feasible solution to the problem while minimizing the makespan.

Definition 4.0.5. Budgeted Tree Cover:(BTC) Consider a complete graph $G = (V, E)$ with a edge lengths $l : E \to \mathbb{N}$ (where l is a metric), weights $w : V \to \mathbb{N}$ and a budget $K \in \mathbb{N}$. A feasible solution to the problem is a set of tree, root pairs (called a cover)

$T = \{(T_i, r_i)\}_i$ such that $\sum_{r_i \in R} w(r_i) \leq K$. The objective is to find a feasible solution to the problem while minimizing the makespan.

4.1 The Algorithm's Components

We mimic Even et al.'s strategy [11]. We guess a value B for the makespan and remove all edges e from G such that $l(e) > B$ (possibly disconnecting the graph in the process). We then find a minimum spanning tree for each connected component of the graph. Each spanning tree is then decomposed into trees of total edge length between $2B$ and $4B$ using Algorithm AlgorithmEdge-decompose$(M_i, 2B)$. A root is assigned to each of the trees obtained from the decompostion by solving a certain bipartite matching problem. We will first look at each of the individual ideas in detail and then tie them together to form the algorithm for our problem. We will also prove that the algorithm gives us an 5-approximation for the problem.

4.1.1 Edge-decomposition of Trees

We first delete all the heavy edges. Now it is possible that the graph is disconnected. The minimum spanning tree M_i for each component is then decomposed into smaller subtrees using Algorithm AlgorithmEdge-decompose(M_i, B) where the parameter β is set to B (which is our guess for the makespan). Trees obtained from the decomposition are categorized into light, medium and heavy based on their edge-length. A tree T is *light* if $l(T) < 2B$, *medium* if $l(T) \in [2B, 4B)$ and *heavy* otherwise.

4.1.2 Assigning Roots to Trees

BTC requires each tree in the cover to be rooted. The algorithm initally breaks up the graph into trees and later assigns a root to each tree in the cover. The previous section covered the decomposition procedure that builds the trees. In this section we will see how the roots are assigned. On a high level, this is done by constructing a bipartite graph and computing a particular type minimum cost matching in the graph.

Consider a BTC instance as defined in Section 1.4.3. Let $\mathcal{T} = \{T_i\}_i$ be a set of trees obtained by the edge-decomposition procedure from Lemma 3.1.1. As before, the distance between a tree $T \in \mathcal{T}$ and a vertex $v \in V$ is defined as $min_{u \in V(T)} d(u, v)$. Say we are given a parameter B such that a tree maybe rooted at a vertex only if they are at most B away. The assignment procedure is very similar to Algorithm AssignRoots (Algorithm 4.1.2). We make a change in the construction of the bipartite graph. When adding an edge between a root r and a tree T, the edge is assigned a cost equivalent to the weight of r. The Algorithm AssignWeightedRoots details this construction.

Algorithm 6 AssignWeightedRoots($\mathcal{T}, K, B$) - Returns a set of root-tree pairs $\{(T_i, r_i)\}_i$ such that $d(T_i, r_i) \leq B$ for all i and $\sum_i r_i \leq K$

```
1: Initialize: $\mathcal{T} \leftarrow \emptyset$
2: Construct a bipartite graph $H = (\mathcal{T} \cup R, E)$ where $\mathcal{T}$ contains a vertex for each $T \in \mathcal{T}$
3: Add edges between every tree-root pair $T \in \mathcal{T}$ and $r \in R$ if the distance between $T$
   and $r$ is at most $B$.
4: for all vertices $r \in R$ do
5:     if $e \in \delta(r)$ then
6:         Assign $e$ the cost $w(r)$.
7:     end if
8: end for
9: Find a minimum cost $\mathcal{T}$-saturating matching $\mathcal{M}$ in $H$. (If no such matching exists
   $\mathcal{M} \leftarrow \emptyset$)
10: if $\sum_{e \in \mathcal{M}} > K$ then
11:     Return "Algorithm failed: $B < B^*$".
12: end if
13: $\mathcal{T} \leftarrow \mathcal{T} \cup \{(T_i, r_i)\}_i$ where $T_i$ and $r_i$ are the matched pairs of vertices.
14: Return $\mathcal{T}$
```

4.1.3 What if we knew the optimal makespan?

For now, assume that we have a guess B for the optimal makespan B^* such that $B \geq B^*$. We will see how we may obtain such a value for B in Section 4.2. So given a guess $B \geq B^*$, we do the following:

Since no tree is allowed to be of length more than B, we first remove all edges e such that $l(e) > B$. We then compute a minimum spanning tree M_i for each connected component of the graph resulting from Step 1. For the decomposition, we use Algorithm Edge-decompose with $2B$ as the parameter and that gives us the required decomposition. The assignment of roots to trees is done by using Algorithm AssignWeightedRoots. Note that only the trees of medium edge-length participate in the matching subroutine. Hence, for each light component L_i, we look for a medium weight subtree S_i with root s such that $V(L_i) \cap V(S_i) \neq \emptyset$ and assign s to be the root of L_i . This works because in an optimal solution, a root r will cover exactly one of these light components. To see why, observe that the light components were formed by the removal of edges of length greater than the optimal makespan B^*. Thus if a tree from the optimal solution covers two different light components, the total edge length of the tree needs to be $> B^*$ which is not possible since the optimal makespan is B^*. Another important observation that follows from the same reasoning is that for any light tree L, it's assigned root r from the optimal solution belongs to $V(L)$. This is because other components are at least B^* away and thus cannot be covered by r.

However, we still need to prove that given a guess $B \geq B^*$ for the makespan, there

Algorithm 7 ConstructBudgetedTreeCover(G, K, B) - Compute a tree cover of G with makespan at most $4B$ and total cost of roots at most K

```
1: Remove all edges of length $> B$. Let $\{C_i\}_i$ be the connected components of the graph
   formed after deleting heavy edges.
2: for all $i$ do
3:     $M_i \leftarrow$ minimum spanning tree of $C_i$.
4:     $\{S^i_j\}_{i,j} \leftarrow$ medium trees from Algorithm Edge-decompose$(M_i, 2B)$ and $\{L_i\}_i \leftarrow$ any
   leftover light trees. (See Lemma 3.1.1)
5: end for
6: $\mathcal{T} \leftarrow$ AssignWeightedRoots$(\{S^i_j\}_{i,j}, K, B)$
7: for all light trees in $\{L_i\}_i$ do
8:     Find medium weight subtree $S_i$ with root $s_i$ such that $V(L_i) \cap V(S_i) \neq \emptyset$.
9:     Assign $s_i$ to be the root of $L_i$.
10: end for
11: $\mathcal{T} \leftarrow \mathcal{T} \cup \{(L_i, s_i)\}_i$
12: Return $\mathcal{T}$ as the tree cover.
```

exists a feasible assignment of roots to trees. Recall that the matching procedure involves constructing a bipartite graph $H = (\mathcal{T} \cup R, E)$ as described in Section 4.1.2. Thus, we need to prove the following lemma:

Lemma 4.1.1. *If $B \geq B^*$, then there exists a $\mathcal{T}$-saturating matching in H such that $\sum_{r \in R'} w_r \leq K$ where $R' \subseteq R$ is the subset of R saturated by the matching.*

Proof. Fix an optimal solution $\mathcal{T}^* = \{T^*_i\}_{i=1}^k$ with roots $R^* = \{r^*_i\}_i$. Let $R' = R \setminus R^*$. Delete all the vertices of R' from H to form the graph H. Since the remaning vertices from R are exactly the vertices from R^*, we know that $\sum_{r \in R^*} f_r \leq K$. Thus, if we can show the existence of a $\mathcal{T}$ saturating matching that only uses roots from R^*, we are done. By Hall's condition, all we need to show is that for any subset $\mathcal{S}$ of $\mathcal{T}$, the set of neighbours $N(\mathcal{S})$ is at least as large as $\mathcal{S}$.

Consider now a subset $\mathcal{S}$ of trees from $\mathcal{T}$. Define $T^*(S)$ to be the set of trees from the optimal solution that cover the vertices in $\mathcal{S}$ i.e. $T^*(S) = \{T \in \mathcal{T} : \exists S \in \mathcal{S}$ such that $V(S) \cap V(T) \neq \emptyset\}$. We claim that $|N(\mathcal{S})| \geq |T^*(\mathcal{S})|$. (Proven as Claim 4.1.2)

Thus to prove $|N(\mathcal{S})| \geq |\mathcal{S}|$, we just need to show that $|\mathcal{T}^*(\mathcal{S})| \geq |\mathcal{S}|$.

Let $|S| = k$ and $|\mathcal{T}^*(\mathcal{S})| = l$. For brevity, we assume that the graph did not get disconnected on the removal of edges with weight $> B$. The argument generalizes even if the graph is disconnected as we can just apply this procedure to each connected component. Let U be the set of all the edges from $T^*(S)$ as defined above. By way of contradiction, assume that $l < k$. Use MST to denote the minimum spanning tree constructed in Step 2 of Algorithm BudgetedTreeCover. Call two trees T_i and T_j "close" if they are connected by an edge from the minimum spanning tree MST i.e. if $\exists u \in T_i$ and $v \in T_j$ such that

$(u, v) \in E(MST)$. Then build a new graph $\mathcal{H}$ that spans $\mathcal{S}$ as follows: Add all the edges and vertices from $\{T_i^*\}_{i=1}^{l}$. Then, while there exist T_i^*, T_j^* that are close but not in the same component of H, add the cheapest connecting edge between i and T_j. The maximum edge cost for H now is $\leq lB + (l1)B < 2kB$ since we assumed $l < k$. Thus, the total edge-length of $\mathcal{H}$ is less than the total edge-length of $\mathcal{S}$.

We now claim that each tree T of $\mathcal{S}$ lies in a connected component of $\mathcal{H}$. (see Claim 4.1.4).

Claim 4.1.2. *Let H be a bipartite graph constructed as described in Algorithm* AssignWeightedRoots. *For any $U \subseteq V(H)$, define $N(U)$ to be the neighbours of U. Given any $\mathcal{S} \subseteq \mathcal{T}$ and an optimal cover $\mathcal{T}^*$, $|N(\mathcal{S})| \geq |\mathcal{T}^*(\mathcal{S})|$ where $\mathcal{T}^*(\mathcal{S})$ is the set $\{T \in \mathcal{T} : \exists S \in \mathcal{S}\}$ such that $V(S) \cap V(T) \neq \emptyset$.*

(Proved later). Assuming the above claim, construct a new graph $\mathcal{H}'$ by removing all the edges of U from the minimum spanning tree MST and adding in the edges from $\mathcal{H}$. Note that $\sum_{e \in \mathcal{H}'} = w(MST) - \sum_{e \in U} w(e) = w(MST) - \sum_{e \in U} w(e) + \sum_{e \in \mathcal{H}} w(e)$. Thus, we have a graph spanning all the vertices of the graph that weighs less than the minimum spanning tree. So if we can prove that this graph is connected, we have a contradiction.

We know all pairs of vertices (u, v) such that $\{u, v\}$ is an edge in the minimum spanning tree are connected. Thus we only need to worry about the pairs of vertices (u, v) such that the path P_{uv} between them is of the form $u, w_1, \cdots, w_p, v$ where $w_1, \cdots, w_p \in V(S)$. Since these vertices are neighbours of each other in MST , they lie in the same component of H (by our claim). Thus each of $w_1, \cdots, w_p$ belong to the same connected component. We only removed edges from U i.e. edges in $\mathcal{T}^*(\mathcal{S})$, the edges (u, w_1) and (w_p, v) are from the minimum spanning tree. Thus, $\mathcal{H}$ is still connected since we have not removed any edges from $\mathcal{H}$. So, we have a graph spanning all the vertices such that the edge weight is less than that of the minimum spanning tree. However, this is a contradiction to the minimality of the minimum spanning tree. Hence, $|\mathcal{T}^*(\mathcal{S})| \geq |\mathcal{S}|$. □

Thus, there exists an assignment of trees to optimal roots and the lemma holds.

Note that if the graph gets disconnected upon removal of edges with length $> B$, we can repeat the same argument for the minimum spanning tree of each component.

We now prove the claims that we made in our proof of the lemma above.

Claim 4.1.3. *(4.1.2) Let H be a bipartite graph constructed as described in Section 4.1.2. For any $U \subseteq V(H)$, define $N(U)$ to be the neighbours of U. Given any $\mathcal{S} \subseteq \mathcal{T}$ and an optimal cover $\mathcal{T}^*$, $|N(\mathcal{S})| \geq |\mathcal{T}^*(\mathcal{S})|$ where $\mathcal{T}^*(\mathcal{S})$ is the set $\{T \in \mathcal{T} : \exists S \in \mathcal{S}\}$ such that $V(S) \cap V(T) \neq \emptyset$.*

Proof. Every tree $S \in \mathcal{S}$ satisfies $w(S) \in [2B, 4B)$. If any tree T (rooted at a vertex r (from the optimal solution) intersects a tree $S \in \mathcal{S}$, then the distance between r and S is at most the length of the optimal makespan which is $\leq B$. Hence, there is an edge between r and T (by construction of H). This implies that $|N(S)| \geq |\mathcal{T}^*(\mathcal{S})|$. □

Claim 4.1.4. *Let S be a subset of trees. Each tree $T \in S$ lies in a connected component of H.*

Proof. By way of contradiction, assume there are vertices u and a neighbour $v \in N(u)$ such that u and v are in different components of $\mathcal{H}$. But since u and v are neighbours, the two trees they belong to are close. Thus there is a path between u and v in $\mathcal{H}$ by our construction of $\mathcal{H}$. Thus u and v belong to the same component of H. □

4.2 Algorithm and Approximation Guarantee

The detailed algorithm is presented as Algorithm BudgetedTreeCover. The following is a high level overview. Start off with a low initial guess B for the makespan. Use Lemma 3.1.1 to break the graph up into a set of trees. Run the subroutine from Section 3.1.3 to try and assign roots to each of the resulting trees. If the algorithm returns a matching, declare that the algorithm succeeded and output the set of rooted trees returned by the algorithm as the solution. If not, declare that the algorithm failed, set $B \leftarrow 2B$ and repeat Subroutine 4.1.3 with the new guess. We repeat this procedure till a feasible matching is returned. We can obtain a value for B that is arbitrarily close to B^* (i.e.$B = B^* + \epsilon$ for any given $\epsilon > 0$) using the techniques used by Even et al. in Theorem 3 of [11].

Algorithm 8 BudgetedTreeCover(G, K, B) - Compute a tree cover of G with cost at most $4B$ and total cost of roots at most K

```
1: Remove all edges of length greater than $B$. Let $\{C_i\}_i$ be the connected components of
   the graph formed after deleting heavy edges.
2: for all $i$ do
3:     $M_i \leftarrow$ minimum spanning tree of $C_i$.
4:     $\mathcal{T} \leftarrow$ AlgorithmEdge-decompose$(M_i, 2B)$
5:     $\{S^i_j\}_{i,j} \leftarrow (T, r)$ from $\mathcal{T}$ such that $l(T) \in [B, 2B)$ and $\{L_i\}_i \leftarrow$ any leftover light
   trees. (See Lemma 3.1.1)
6: end for
7: $\mathcal{M} \leftarrow$ AssignRoots$(\{S^i_j\}_{i,j}, K, B)$
8: if $\mathcal{M} == \emptyset$ then
9:     Print: "$B$ is too low".
10:    Return: BudgetedTreeCover$(G, K, 2B)$
11: end if
12: Return: $\mathcal{M}$
```

The following section proves the approximation guarantee of our algorithm. We will assume that the value of B in the following section is arbitrarily close to the optimum.

4.2.1 Proving the approximation guarantee

Theorem 4.2.1. *Algorithm* BudgetedTreeCover *produces a tree cover of graph* $G = (V, E)$ *of makespan at most* $(5+\epsilon)$ *times the optimal (for any* $\epsilon > 0$*)that runs in time polynomial in the size of* G *and log*$(\frac{1}{\epsilon})$

Proof. The weight of each subtree created by the decomposition procedure is in the range $[2B, 4B)$ and by construction of our bipartite graph H, each root is at most a distance of B away from the tree. Finally, for the each of the leftover light trees L, the root is contained inside the vertex set of L and hence, the distance between the tree and root is 0.

Thus the makespan of our cover is $< 4B + B = 5B$. Since $B = B^* + \epsilon$, we get that the makespan is $5(B^* + \epsilon)$. Thus we get a $(5+\epsilon)$-approximation. □

This finishes our discussion of our algorithm for the Budgeted Tree Cover problem. It can be seen from the discussion that this algorithm is a very basic and direct generalization Algorithm k-TreeCover (Section 3.1)and Algorithm RootedTreeCover (Section3.2) Hence, it is not surprising that the approximation ratio is worse-off than the two we saw before - in particular, we have a 5-approximation (instead of 4 like Rooted Tree Cover). The next chapter discusses a recent result of Khani and Salavatipour [25] that provides a 3-approximation for k-Tree Cover. The goal is to try and present certain ideas that might help improve the approximation guarantee for BTC.

Chapter 5

Ideas for future work

This chapter focuses mainly on an idea for improving upon the 5-approximation algorithm for BTC. The second half of the chapter talks about other open problems to investigate.

5.1 A better algorithm for BTC?

The work of Khani and Salavatipour improves upon algorithm for k-tree cover by Even *et al.* and algorithm for minimum tree cover by (See Section 1.4.3). Khani and Salvatipour give a 3-approximation algorithm for KTC and a 2.5-approximation algorithm for Minimum tree cover [25] (They refer to the problem as Bounded Tree Cover). We will focus on the k-tree cover problem in an effort to find ways to improve the approximation guarantee for BTC. In the algorithms by Even *et al.* [11] and Arkin *et al.* [1] the first step is to remove all edges of weight more than our guess for the makespan. The main idea of Khani and Salavatipour is to filter the edges more aggresively. So the algorithm deletes any edges of weight more than $B/2$ where B is the current guess for the makespan. On a high level the idea is to later introduce only some of these deleted edges later if need be. Since we have deleted some of these heavy edges that were included in Even *et al.* 's algorithm, the constructed cover has a lower makespan.

5.1.1 A high level overview

Recall that an instance of KTC consists of a graph $G = (V, E)$ with lengths $l : E \rightarrow \mathbb{N}$ that form a metric. A feasible solution to the problem is a tree cover of G with no more than k trees. The objective is to find a feasible solution with minimum makespan. Just as algorithms by Even *et al.* , the algorithm by Khani and Salavatipour binary searches for the optimal makespan. Let the optimal makespan be B^*. Then assuming that we have an upper bound B on the makespan, we delete all edges from G that have length more than $B/2$. This possibly disconnects the graph into serveral components $\{C_i\}_i$. Define the *tree* *length* of a component C_i to be the length of the minimum spanning tree of C_i. They call a component *light* if it's tree length is less than B, *medium* if the length is in the range $[\frac{3B}{2}, 3B)$ and *heavy* otherwise.

For every light component C_i, they do one of three things: Construct a minimum spanning tree of C_i and include it in our cover or connect C_i to another light component C_j with an edge of length at most B and then add a minimum cost tree spanning both C_i and C_j to the cover or connect it to a heavy component and decompose the resulting new graph (using AlgorithmEdge-decompose and Lemma 3.1.1) into medium subtrees and add them to the cover. Based on this, we can see that none of the trees in our cover have edge-length longer than $3B$. Thus we only need to ensure that there are no more than k trees. The proof relies on using a fixed optimal solution and showing that each tree in the constructed set can be covered by some tree in the optimal solution. As the optimal solution has no more than k trees, this would prove that the solution is feasible and a 3-approximation.

5.1.2 Aggresive filtering for Budgeted Tree Cover?

The natural question to ask is whether filtering more aggressively can help the approximation guarantee for Budgeted Tree Cover. Given an upper bound B on the optimal makespan B^*, Algorithm BudgetedTreeCover starts with deleting all edges with length more than B. The main lemma proving the correctness of the algorithm is Lemma 4.1.1. The lemma's proof hinges upon using an optimal solution and showing that every tree in our constructed cover can be covered by trees from the fixed optimal solution. Thus if a tree T^* from the optimal solution covers some tree T in the constructed solution then T^* does not cover any other trees. Intuitively, this one condition helps us prove that each tree in the constructed solution can be covered by some root in the optimal solution. This is because every other component is at least B^* away (by construction). If we could cover more than one tree using one optimal root, then we would have extra roots from the optimal solution to prove $|\mathcal{T}^*(\mathcal{S})| \geq |\mathcal{S}|$ where $\mathcal{S}$ is a subset of our constructed cover and $\mathcal{T}^*(\mathcal{S})$ is the set of trees from the optimal solution covering $\mathcal{S}$.

Thus if we were filtering aggressively and deleting any edges of length more than $B/2$, then the distance between two components is at least $\geq B/2$. So in a cover with makespan B an optimal root could cover two components, each at a distance of $B/2$. Thus we can now use a single root to cover "half" of a heavy component. The main idea for improving the approximation guarantee is to alter Algorithm AssignWeightedRoots such when assigning roots, we can try to cover a heavy component using two roots.

5.1.3 Altering the root assignment process

First we review Algorithm AssignWeightedRoots. Let $\mathcal{T}$ be the set of trees returned by Algorithm Edge-decompose$(T_i, 2B)$. The distance between a tree $T \in \mathcal{T}$ and a vertex

$v \in V(T)$ is defined as before. To assign a root from some $R \subset V$ to each tree in T , we construct a bipartite graph $H = (\mathcal{T} \cup R, E)$ where $\mathcal{T}$ contains a vertex for each $T \in \mathcal{T}$ and R. Put an edge of cost $w(r)$ between a tree $T \in \mathcal{T}$ and a vertex $r \in R$ if the distance between T and r is at most B. The assignment is then done by finding a minimum cost $\mathcal{T}$-saturating matching with edge cost at most K.

The above process is then altered as follows: $\mathcal{T}$ is obtained from by Algorithm Edge-decompose$(T_i, 2B)$. Construct a hypergraph $H = (\mathcal{T} \cup R, E_r \cup E_\Delta)$. Here E_Δ is a set of *hyperedges* $\{r, s, T\}$ for $r, s \in R$ and $T \in \mathcal{T}$ such that the distance between r and T and s and T is at most B, E_r is a set $\{r, T\}$ for $r \in R$ and $T \in \mathcal{T}$ where the distance between r and T is at most B. Then one needs to find a hypergraph matching $F \subseteq E$ of cost at most K such that every vertex T in $\mathcal{T}$ has at most two edges from F incident with it and every vertex r in R is incident with at most one edge. There are several questions to be answered: Firstly, can this hypergraph problem be solved? If yes, then given a decomposition can we find a hypergraph matching with edge-cost no more than the optimal makespan? Finally, does this approach actually give an improvement on the approximation guarantee?

5.2 Other avenues to investigate

A closely related problem is that of Bounded Tree Cover. It is a dual problem of the k-Tree Cover problem. In k tree cover, we limit the number of vertices and minimize the makespan. In the Bounded Tree Cover problem, the makespan is given as a parameter and the objective is to minimize the number of trees in the cover. Formally we have a graph $G = (V, E)$ with non-negative weights $c : E \to \mathbb{R}$ and a parameter $B > 0$. A feasible solution is a tree-cover of the graph $\mathcal{T}$ with makespan at most B. The objective is to find a feasible solution that minimizes the size of $\mathcal{T}$. Khani and Salavatipour gave a 2.5-approximation algorithm for Bounded Tree cover. They use the procedure for k-Tree cover and guarantee that the makespan is under B and k is minimized. However, Khani and Salavatipour do not establish any lower bounds on the approximation ratio and hence, it would be interesting to see if the ratio can be improved or if any lower bounds can be proven.

Another interesting avenue is Capacitated Vehicle Routing with Non-Uniform Speeds. Gørtz *et al.* gave a constant-factor approximation for uniform capacities. However, they mention that their technique fails when the capacities are not identical. This is because they sort the locations based on their distance from the depot and then assign vehicles to locations based on their speeds. Hence, new ideas will be necessary to design a solution for Heterogeneous-CVRP with non-uniform capacities.

The above discussion only scratches the surface of the possible open questions associated with routing problems. Several other related problems such as Capacitated Steiner Trees and Facility Location based problems are open to further investigation and pose very interesting questions. Vehicle Routing is a highly applicable (and “difficult”) problem and new variants still arise in practice. While the mathematics involved in improving current work is fascinating, coming up with more complex scenarios and tackling them (Stacker-Crane variants, for example) provides nearly unlimited room for study and exploration.

APPENDICES

Appendix A

A (very) short discussion of Integer Linear Programs

Linear programming originated in the 1950's as a way of modelling optimization problems. A *linear program* (LP) is the problem of maximizing (or minimizing) a linear objective function over the set of all vectors satisfying a given set of linear inequalities and equations [30]. In 1979, Leonid Khachiyan proved that linear programs can be solved in polynomial time. A linear program is called an *integer* linear program (ILP) if the variables are only allowed to take on integer values. A solution to the linear program is an assignment of integers to the variables such that all the constraints are satisfied. For the purposes of this essay, it will be enough to restrict ourselves to boolean variables i.e. the variables can either be set to 0 or to 1. This case is referred to as $0,1$-integer linear programming. $0,1$-integer linear programming was shown to be NP-Hard by Karp and is one of Karp's "21 NP-Complete problems" [24].

An $0,1$-integer linear program is canonically written down as the following maximization problem.

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{s.t.} \quad & A \cdot x \leq b \\ & x \in \{0,1\} \end{aligned}$$

The corresponding linear programming *relaxation* can be obtained by letting the variables x take on any value between 0 and 1 (thus "relaxing" one of the constraints).

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{s.t.} \quad & A \cdot x \leq b \\ & x \geq 0 \end{aligned}$$

Note that every ILP can be written down in the canonical form because of *duality* of linear programs. In this case the original linear program is referred to as the *primal* problem and the corresponding program obtained via duality is called the *dual.* The dual of the above linear programming relaxation is the following linear program.

$$\begin{aligned} \text{minimize} \quad & b^T y \\ \text{s.t.} \quad A^T \cdot y & \geq c \\ y & \geq 0 \end{aligned}$$

The idea is that we have one variable in the dual for each constraint in the primal and one constraint for each variable in the primal. The most basic implication of duality is that the dual of a dual is the primal. Another important concept is that of *strong duality* which says that if the dual has an optimal solution then so does the primal and that they both have the same optimal value. A special kind of duality that arises very frequently in the study of approximation algorithms and in various combinatorial optimization problems is the relation between *covering* and *packing* LPs. The above minimization program is a typical representation of a covering LP while the dual maximization program is a typical packing LP. For example, the LP for minimum vertex cover is a minimization program and dual of the LP gives us a linear program for the maximum independent set problem.

References

[1] Esther M. Arkin, Refael Hassin, and Asaf Levin. Approximations for minimum and min-max vehicle routing problems. *J. Algorithms*, 59(1):1–18, 2006.

[2] Sanjeev Arora and Boaz Barak. *Complexity Theory: A Modern Approach*. Cambridge University Press, 2009.

[3] Sanjeev Arora and George Karakostas. A 2+epsilon approximation algorithm for the k-mst problem. In *SODA*, pages 754–759, 2000.

[4] Avrim Blum, R. Ravi, and Santosh Vempala. A constant-factor approximation algorithm for the k-mst problem, 1996.

[5] William J. Cook. *In Pursuit of the Traveling Salesman*. Springer, 2006.

[6] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver, editors. *Combinatorial Optimization*. John Wiley and Sons, 1998.

[7] G. B. Dantzig and J. Ramser. The truck dispatching problem. *Management Science*, 6(1):8091, 1959.

[8] M. Desrochers, J. K. Lenstra, M. W. P. Savelsbergh, and F. Soumis. Vehicle routing with time windows: optimization and approximation. In *Vehicle Routing: Methods and Studies*, pages 65–84. North-Holland, 1988.

[9] Reinhard Diestel. *Graph Theory*. Springer-Verlag, Berlin, 2000.

[10] Jack Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of Research of the National Bureau of Standards*, B(69):125–130, 1965.

[11] Guy Even, Naveen Garg, Jochen K´onemann, R. Ravi, and Amitabh Sinha. Min-max tree covers of graphs. *Oper. Res. Lett.*, 32(4):309315, 2004.

[12] M. Fischetti, H. W. Hamacher, K. Jrnsten, and F. Maffioli. Weighted k-cardinality trees:complexity and polyhedral structure. In *Networks*, 24, pages 11–21, 1994.

[13] Greg N. Frederickson, Matthew S. Hecht, and Chul E. Kim. Approximation algorithms for some routing problems. *Siam J. Comput.*, 7(2):178–193, 1978.

[14] Harold N. Gabow and Ronald Fagin, editors. *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*. ACM, 2005.

[15] Naveen Garg. A 3-approximation for the minimum tree spanning k vertices. In *FOCS*, pages 302–309, 1996.

[16] Naveen Garg. Saving an epsilon: a 2-approximation for the k-mst problem in graphs. In Gabow and Fagin [14], pages 396–402.

[17] Michel Goemans and David P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24:296–317, 1992.

[18] Leslie Ann Goldberg, Klaus Jansen, R. Ravi, and José D. P. Rolim, editors. *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 14th International Workshop, APPROX 2011, and 15th International Workshop, RANDOM 2011, Princeton, NJ, USA, August 17-19, 2011. Proceedings*, volume 6845 of *Lecture Notes in Computer Science*. Springer, 2011.

[19] Bruce Golden and Arjang Assad, editors. *Vehicle Routing: Methods and Studies*. Studies in Management Science and Systems. North-Holland, 1988.

[20] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[21] Inge Li Gørtz, Marco Molinaro, Viswanath Nagarajan, and R. Ravi. Capacitated vehicle routing with non-uniform speeds. In Günlük and Woeginger [22], pages 235–247.

[22] Oktay Günlük and Gerhard J. Woeginger, editors. *Integer Programming and Combinatoral Optimization - 15th International Conference, IPCO 2011, New York, NY, USA, June 15-17, 2011. Proceedings*, volume 6655 of *Lecture Notes in Computer Science*. Springer, 2011.

[23] M. Haimovich and A. H. G. Rinnooy Kan. Bounds and heuristics for capacitated routing problems. *Mathematics of Operations Research*, 10(4):pp. 527–542, 1985.

[24] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.

[25] M. Reza Khani and Mohammad R. Salavatipour. Improved approximation algorithms for the min-max tree cover and bounded tree cover problems. In Goldberg et al. [18], pages 302–314.

[26] Samir Khuller, Balaji Raghavachari, and Neal E. Young. Balancing minimum spanning and shortest path trees. *Algorithmica*, 14(4):305–321, 1995.

[27] Donald Knuth. *The TEXbook.* Addison-Wesley, Reading, Massachusetts, 1986.

[28] Leslie Lamport. *LATEX — A Document Preparation System.* Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[29] Jan Karel Lenstra, David Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Math. Program.*, 46(3):259–271, February 1990.

[30] Jirí Matousek and Bernd Gärtner. *Understanding and Using Linear Programming.* Springer, 2006.

[31] R. Ravi, R. Sundaram, M.V. Marathe, SS Ravi, and D.J. Rosenkrantz. Spanning trees short or small. *arXiv preprint math/9409222*, 1994.

[32] Alexander Schrijver. On the history of combinatorial optimization (till 1960). In G.L. Nemhauser K. Aardal and R. Weismantel, editors, *Discrete Optimization*, volume 12 of *Handbooks in Operations Research and Management Science*, pages 1 – 68. Elsevier, 2005.

[33] David B. Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Math. Program.*, 62(3):461–474, 1993.

[34] Paolo Toth and Daniele Vigo, editors. *The Vehicle Routing Problem.* Monographs on Discrete Mathematics and Applications. SIAM, 2002.

[35] Vijay Vazirani. *Approximation Algorithms.* Springer, 2005.

[36] David P. Williamson and David B. Shmoys. *The Desigh of Approximation Algorithms.* Cambridge University Press, 2012.

[37] Alexander Zelikovsky and D. Lozevanu. Minimal and bounded trees. In *Tezele Cong. XVIII Acad. Romano-Americane*, pages 25–26. Kishinev, 1993.