APPROXIMATION ALGORITHMS FOR THE CAPACITATED VEHICLE ROUTING PROBLEM

by

Yuzhuang Hu

BS, Wuhan University, 1992

ME, Institute of Computing Technology, Chinese Academy of Sciences, 2002

a Thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Summer 2009

APPROVAL

Name: Yuzhuang Hu

Degree: Doctor of Philosophy

Title of Thesis: Approximation Algorithms for the Capacitated Vehicle Routing Problem

Examining Committee: Dr. David G. Mitchell
Chair

Dr. Binay Bhattacharya, Senior Supervisor, SFU

Dr. Ramesh Krishnamurti, Supervisor, SFU

Dr. Fabian Chudak, Supervisor, SFU

Dr. Abraham P Punnen, SFU Examiner

Dr. Samir Khuller, External Examiner,
Professor, Department of Computer Science,
University of Maryland

Date Approved: August 25, 2009

Declaration of
Partial Copyright Licence

Abstract

The Vehicle Routing Problem (VRP) is a generalization of the Traveling Salesman Problem (TSP) and is also one of the most challenging tasks in the area of combinatorial optimization. In the VRP we are given p vehicles and an undirected complete graph $G = (V, E)$ where edge weights satisfying the triangle inequality. The objective is to find a separate tour for each vehicle while minimizing the total cost of the tours. In the Capacitated Vehicle Routing Problem (CVRP), each vehicle has a limited capacity k, and it is required that the vehicle should never exceed its capacity at any point of the tours.

In this thesis we present approximation algorithms for the CVRP. The CVRP, in fact, represents a large class of TSP-like problems. We consider approximation algorithms for three variants of the CVRP, namely the Capacitated Vehicle Routing Problem with Pick-ups and Deliveries (CVRPPD), variants of the Cycle Covering Problem (CCP), and the Capacitated Vehicle Routing Problem with Multi-Depots and Multi-Vehicles.

Many practical applications, such as mail delivery, parcel delivery and pickup, and bus routing, can be modeled by the CVRPPD. In this class of problems, we focus on the k-delivery Traveling Salesman Problem, the Capacitated Dial-a-Ride Problem, and the Black and White Traveling Salesman Problem (BWTSP). We design a matching-based constant factor approximation algorithm for the BWTSP, and we propose a rule to improve the approximation ratios for the k-delivery TSP and the Capacitated Dial-a-Ride Problem.

Finding cycle covers with minimum edge costs is a fundamental graph problem. In the second part of the thesis we investigate some NP-hard variants of the CCP. These variants consider two additional constraints. One of the constraints is on the number of vertices in each cycle of a cycle cover, and the other is on the number of cycles appearing in a cycle cover. We present constant factor approximation algorithms for these problems.

Different from the default settings of the CVRP, where only a central depot is involved,

in the CVRP with Multi-Depots and Multi-Vehicles, the vehicles may start from different depots. In this category we study a model of the Multi-Depot Capacitated Vehicle Routing Problem (MDCVRP) and the Multi-Vehicle Scheduling Problem (MVSP). We propose a dynamic programming based method to design approximation algorithms for these problems.

Keywords

Approximation Algorithms *Vehicle Routing Problem* *Capacity*

To my parents, my wife, and the coming baby.

"To be or not to be: that is the question:"

— *Hamlet*, William Shakespeare, *1600*

Acknowledgments

I would like to express my deep gratitude to my senior supervisor, Dr. Binay Bhattacharya. This thesis would not be possible if without his generous support in all aspects of my graduate study in the past five years. His kindness towards students baffled description. As a novice on research, I feel guilty to bother him all the time − the latest experience is the discussion of this thesis after 10pm in his office.

I am grateful to those who served in my supervisory and examination committee. I thank Dr. Ramesh Krishnamurti for his insightful advice on my research, detailed comments on my thesis, and his patience shown when attending my depth examination, proposal, seminars and the final defense. I thank Dr. Samir Khuller, Dr. Abraham P Punnen, and Dr. Fabian Chudak for reading my thesis and giving me valuable suggestions for my future research and study.

I would like to thank Dr. Pavol Hell, who's the graduate program director when I began my Ph.D. program at SFU. I thank Pavol for his help in a critical time of my graduate study. I would also like to thank Dr. Daya Gaur. Visiting the University of Lethbridge is a wonderful experience. I thank Dr. Qiaosheng Shi for frequently discussing research problems with me in these years. I also thank Dr. Robert Benkoczi, and all other people who helped me during my Ph.D. study.

Last, but not least, I would like to thank my parents and my wife. I thank them for their unconditional and endless support to encourage and urge me to finish this thesis. I thank my father for being a father, a friend, and a mentor to me in his life. I thank him for cultivating me to be a useful person. I could never pay my debt to him, but only remember him in my heart. Finally, I would like to thank the coming baby for showing me the beauty of life. He/she would be the biggest Christmas present to me in the world.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Capacitated Vehicle Routing Problem (CVRP)

In some business sectors, transportation adds a significant or even a decisive cost to the goods or services provided to the customers. Examples include mail delivery, milk transport, and bus routing. As the distribution process is repeated daily, a tiny improvement on the vehicle routing process can lead to tremendous energy and monetary savings. In 2005, the transportation sector contributed about 4.2% of Canada's GDP, while the number is 3.7% for Canada's huge mining and oil and gas extraction industry [29]. In [57] it is shown that the annual excess travel in the United States has been estimated at about $45 billion.

In this thesis we investigate computer algorithms to minimize the vehicle routing cost. A simple model of a transportation system is the famous Traveling Salesman Problem (TSP) [74]. It is defined as follows. Given a set of points and their pairwise distances, the problem is to find the shortest tour where each point is visited exactly once. Mathematical problems related to TSP, e.g. the Hamiltonian Cycle Problem [43], were treated early in 1800s by W. R. Hamilton and Thomas Kirkman. The general form of the TSP is formulated in 1930 by Karl Menger [74]. The TSP is NP-hard [50] and has been studied intensively in the area of combinatorial optimization. The best known approximation ratio for TSP, $\frac{3}{2}$, is obtained by the Christofides-Serdyukov heuristic [20, 77].

The Vehicle Routing Problem (VRP) introduced by Dantzig and Ramser [26] is a generalization of the TSP, and models more complex real world applications. In the VRP, we are given p vehicles and an undirected complete graph G with edge costs satisfying the triangle inequality, and the objective is to find a separate tour for each vehicle while minimizing the

total edge cost of the tours. The VRP has numerous variations, and has also been extensively investigated by many researchers. In this thesis our main focus is on the Capacitated Vehicle Routing Problem (CVRP) [84], where each customer is associated with a demand and each vehicle has a uniform capacity constraint. At any point in the tours of the CVRP, the total demands of the customers that have already been serviced by a vehicle should never exceed the vehicle capacity. In summary, the optimal solution of the CVRP satisfies the following:

(1) each tour starts from and ends at a location, called *depot*.

(2) the total customer demands serviced by a vehicle in its tour is at most the vehicle capacity.

(3) each customer's demand is serviced by exactly one vehicle.

(4) the total edge cost of all the tours is minimized.

An example CVRP solution is given in Figure 1.1. In this example it is assumed that each customer has a demand of 1. The vehicle capacity is set to 7.

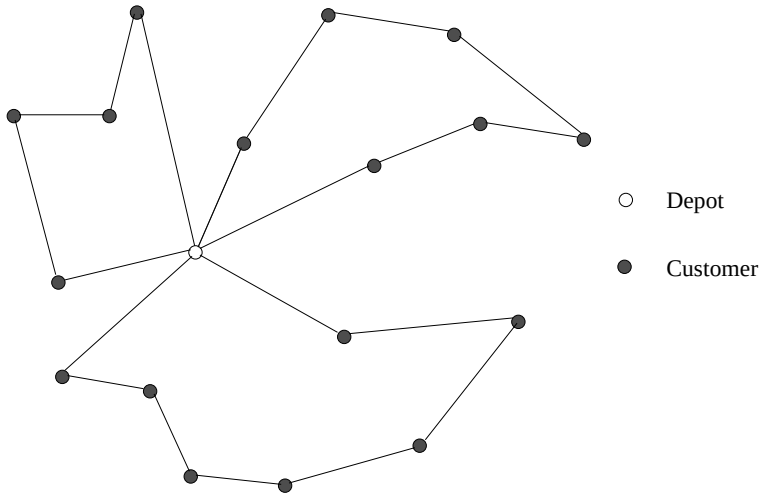

Figure 1.1: An example of the CVRP.

1.2 Approximation Algorithms

Unfortunately, the CVRP is NP-hard as it has the TSP as a special case. More precisely, the CVRP represents a large class of NP-hard problems. Because of the NP-hardness of the CVRP, it is natural to compromise the quality of the solution in order to compute a solution in reasonable time. Heuristics and meta-heuristics are of this type, however, they cannot provide formal evidence to show how "good" the solutions are. On the other hand, there is a special type of algorithm, called *approximation algorithms* in the literature [85], which is designed to produce provably good quality solutions in provably polynomial computation time. The main objective of this thesis is to design approximation algorithms with good performance guarantees for the CVRP.

In the following, we explain more about approximation algorithms. Typically an approximation algorithm runs in polynomial time, and gives solutions with theoretically bounded errors. Most commonly, these errors are multiplicative for approximation algorithms. This means that for all possible cases, the error produced by an approximation algorithm should be bounded in terms of the optimum. We give a formal definition of *performance guarantee* or *approximation ratio* for approximation algorithms as follows:

Definition 1.2.1. *An approximation algorithm has approximation ratio of $\alpha(n)$, if for any input of the problem, the cost C of its solution is within factor ρ of the cost of the optimal solution C^*, i.e.*

$$max\ (\frac{C}{C^*}, \frac{C^*}{C}) \leq \rho$$

Note that the above definition works for both minimization and maximization problems. It would be ideal if the approximation ratio ρ is a small constant, but for many hard problems, the best known approximation ratio ρ is a function of the input size n and/or some other parameters associated with the problem.

1.3 Variants of the Capacitated Vehicle Routing Problem

The CVRP itself represents a large class of graph problems. This thesis investigates the following variants of the CVRP.

1.3.1 Capacitated Vehicle Routing Problem with Pickups and Deliveries

In the Capacitated Vehicle Routing Problem with Pickups and Deliveries (CVRPPD), each vertex is associated with a pickup or delivery demand. A vehicle, or a fleet of vehicles with limited capacity k, traverses the edges and serves all the pickup and delivery demands. During the traversal, the vehicle capacity constraint should always be satisfied. In this thesis we investigate three variants of the CVRPPD, namely the k-delivery Traveling Salesman Problem (the k-delivery TSP) [15], the Capacitated Dial-a-Ride Problem [16], and the Black and White Traveling Salesman Problem (BWTSP) [10].

k-delivery Traveling Salesman Problem

Consider a distribution system which consists of a set of dispersed inventories and a set of retailers. A single type of product is stored in the inventories and needs to be distributed to the retailers. Each inventory stores, and each retailer needs a certain amount of product. The total amount of product in the inventories equals the total demands of the retailers. An example of the system is given in Figure 1.2.

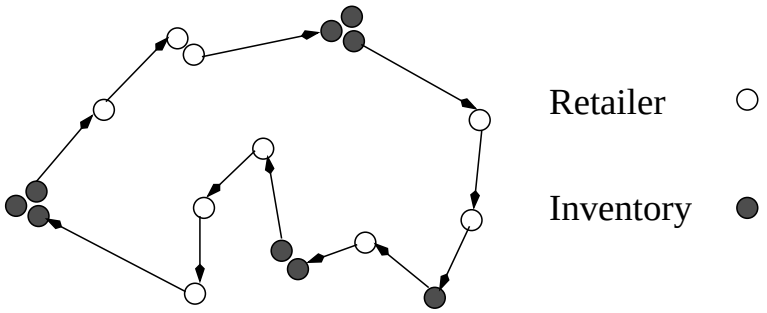

Figure 1.2: An example of the k-delivery TSP. k=3.

In Figure 1.2, a black dot represents one unit of the product in an inventory, and a white dot represents one unit of demand of a retailer. Therefore inventories and retailers are represented as clustered black and white dots respectively. There is a single truck available to pick up the product from the inventories and deliver them to the retailers. The truck has a limit capacity and can only hold at most 3 units of the product at one time. To maximize the profit, the shortest tour needs to be computed to service the retailers.

This example represents an instance of the k-delivery TSP. In the k-delivery TSP, the

vertex set is partitioned into a set of *pickup vertices* and another set of *delivery vertices*. Each pickup vertex is associated with an item, and each delivery vertex requires one item. All the items are identical, and the vehicle should gather items from the pickup vertices and deliver them to the delivery vertices.

Black and White Traveling Salesman Problem

The k-delivery TSP is symmetric, in the sense that the same amount of product is provided or needed by each pickup or delivery vertex. We consider another distribution system where this symmetry does not hold. This distribution system consists of a set of suppliers and a set of inventories. A single type of product is provided by the suppliers and needs to be stored at the inventories. An inventory can store up to k units of product, for a given integer k. The total capacity of the inventories may surpass the total amount of supply. An example of the system is given in Figure 1.3.

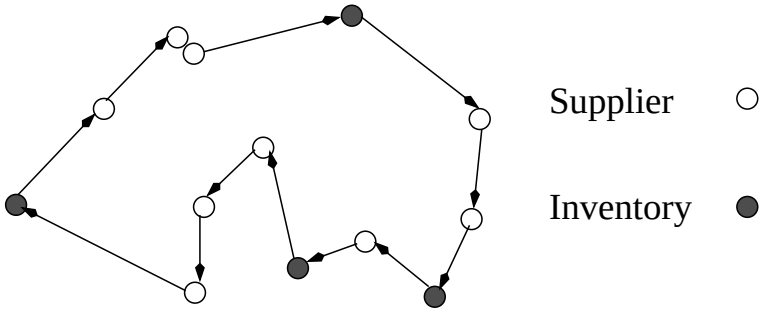

Figure 1.3: An example of the Black and White Traveling Salesman Problem. k=3.

In Figure 1.3, a black dot represents an inventory, and a white dot represents one unit of supply. Therefore suppliers are represented as clustered white dots. The example in Figure 1.3 represents an instance of the Black and White Traveling Salesman Problem (BWTSP). The BWTSP is defined on an undirected complete graph, $G = (V, E)$, where a vertex set, $V = V_B \cup V_W$, is partitioned into a set of *black vertices*, V_B, and a set of *white vertices*, V_W, and an edge set, E, with edge costs $w(e)$ for all $e \in E$ satisfying the triangle inequality. The BWTSP is to determine a minimum cost Hamiltonian tour of G subject to the following restrictions:

1. *Cardinality constraint* in which the number of white vertices on "black to black" paths

is bounded above by a positive integer constant k, and

2. *Length constraint* in which the cost of any path between two consecutive black vertices is bounded above by a positive value L.

In this thesis we show that the BWTSP cannot be approximated unless $P = NP$ if the length constraint is specified. There are some similarities between the k-delivery TSP and the BWTSP with the cardinality constraint. However an algorithm designed for the k-delivery TSP cannot be used directly for the BWTSP with the cardinality constraint. The reason is that in the BWTSP, k times the number of the black vertices may exceed the number of the white vertices in the graph. Therefore without knowing the optimal solution, we cannot determine the number of white vertices assigned to an arbitrary black vertex.

Capacitated Dial-a-Ride Problem

In the k-delivery TSP and the BWTSP, we assume that the items associated with the pickup or white vertices are identical. This assumption, however, does not apply to some real world applications. Consider some transportation systems, e.g. the shared taxi system, where customers call a vehicle service agency to request picking up an item from a source location and delivering the same item to a different destination location. To reduce the cost, the agency first gathers customer requests for a period of time in order to compute a better schedule for servicing these requests. A vehicle with limited capacity is then dispatched to service the customers based on the computed schedule. In these systems, an item picked up from one location must be delivered to its pre-specified destination location. A servicing example of this system is illustrated in Figure 1.4.

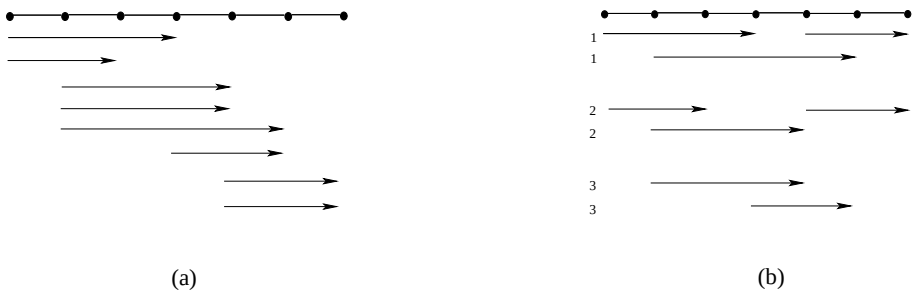

Figure 1.4: An example of the Capacitated Dial-a-Ride Problem on a path. k=2.

The example in Figure 1.4 represents an instance of the Capacitated Dial-a-Ride Problem. In the Capacitated Dial-a-Ride Problem, we are given an undirected complete graph $G = (V, E)$ with edge costs satisfying the triangle inequality, a depot d and a set of jobs, where each job is represented by a source vertex and a destination vertex in G. A vehicle with capacity k is used to serve each job by carrying the item (of one unit) from its source to its destination. The goal is to compute the shortest tour that starts from d and also returns to d after serving all the jobs. It is required that at any point of the tour the vehicle should never carry more than k items. In Figure 1.4, the problem is defined on a path and the depot lies at the left-most vertex of the path. The gathered jobs are given in Figure 1.4(a) and the order of servicing the jobs are marked in Figure 1.4(b).

1.3.2 Cycle Covering Problem

The second part of the thesis considers some variants of the Cycle Covering Problem (CCP). The CCP is a fundamental graph problem and is well related to the VRP. A cycle cover, or a 2-factor of an undirected complete graph $G = (V, E)$, is a set of disjoint simple cycles in G where all the vertices are covered. The CCP is to find a cycle cover with the minimum total edge cost.

Consider the following integer program formulation for the TSP [27]:

$$(TSPIP)\ \ Min \sum_{e \in E} c_e x_e$$

subject to:

$$\sum_{e \in \delta(i)} x_e \geq 2 \qquad \forall i \in V$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \qquad \forall S \subset V$$

$$x_e \in \{0, 1\} \qquad e \in E$$

Here $\delta(S)$ denotes the cross edges between S and $V - S$, c_e represents the cost of an edge e, x_e indicates that whether the edge e is included in the solution, and $E(S)$ denotes the set of selected edges between the vertices in S. The first set of constraints in (TSPIP) is called the *degree constraints*, and the second set of constraints in (TSPIP) is called the *subtour elimination constraints*.

The CCP can be formulated by the following integer program:

$$(CCIP)\ \ Min\ \ \sum_{e\in E} c_e x_e$$

subject to:

$$\sum_{e\in\delta(i)} x_e \geq 2 \qquad \forall i \in V$$

$$x_e \in \{0,1\} \qquad e \in E$$

The above linear program can be obtained from (TSPIP) after dropping the subtour elimination constraints. Therefore the CCP is a relaxation of the TSP. The TSP is NP-hard, however efficient algorithms exist for the CCP [30, 31]. This thesis considers two NP-hard variants of the CCP, namely the Cycle Covering Problem with Bounded Length k (CCPBL), and the p-constrained Cycle Covering Problems (pCCCP).

Cycle Covering Problem with Bounded Length k

Given an integer k and an undirected complete graph $G = (V, E)$, where each edge $e{\in}E$ is associated with a cost c_e, the Cycle Covering Problem with Bounded Length k (CCPBL) is to find a cycle cover of G with minimum total edge cost. In addition, each cycle in the cycle cover is required to have at least 3 but at most k vertices. As the length bound models the vehicle capacity, the CCPBL is a variant of the CVRP with multiple vehicles.

p-constrained Cycle Covering Problem

The p-constrained Cycle Covering Problem (pCCCP) is a variant of network design problems with downwards monotone functions [41]. In practice, p denotes the number of vehicles available to service the customers.

Given an undirected graph $G = (V, E)$ with non-negative edge weights, and a function $f : 2^V \rightarrow \{0, 1\}$, a network design problem can be formulated as the following integer program:

$$(IP)\ \ Min\ \ \sum_{e\in E} c_e x_e$$

subject to:

$$\sum_{e\in\delta(S)} x_e \geq f(S) \qquad \emptyset \neq S \subset V$$

$$x_e \in \{0,1\} \qquad e \in E$$

A downwards monotone function f has the following properties: 1) $f(V) = 0$; 2) $f(A) \geq f(B)$, if and only if $A \subseteq B \subseteq V$. In this thesis we consider a class of network design problems

which can be modeled as an integer program of the type (IP) with downwards monotone functions. For example, in the k-Cycle Covering Problem, we are given an integer k, and the objective is to find a minimum cost cycle cover of G, where each cycle in the cycle cover contains at least k vertices. For the k-Cycle Covering Problem, the corresponding downwards monotone function f can be defined as: $f(S) = 1$ if and only if S has fewer than k vertices, and 0 otherwise.

The pCCCP is obtained by adding an extra constraint to network design problems with downwards monotone functions. More specifically, given an integer p, we require that there should be at most p connected components in the optimal solution. In practice, p can be interpreted as the number of vehicles available to service the customers. For example, the p-constrained version of the k-Cycle Covering Problem models the application scenario that, we have at most p vehicles to service the customers, and each vehicle should collect at least k demands from the customers.

1.3.3 Capacitated Vehicle Routing Problem with Multi-Depots

In a typical setting of the Capacitated Vehicle Routing Problem, a central depot is given, and each vehicle will start and end at the depot. However the assumption of a central depot does not reflect the needs of many real world applications. For example, Krumke et al. [60] mentioned an application with German Automobile Club Allgemeiner Deutscher Automobile-Club (ADAC), where a fleet of vehicles is used to assist people whose cars break down. In this application, requests are batched for a period of time, and whenever a vehicle is dispatched for an accident, the vehicle will remain at the accident position before being assigned a new request received in the next period. It is easy to see that in this application the vehicles may start from different depots for a particular round of service.

This thesis investigates the following two variants of the Capacitated Vehicle Routing Problem with Multi-Depots or Multi-Vehicles.

Multi-Depot Capacitated Vehicle Routing Problem (MDCVRP)

In the MDCVRP we are given an undirected complete graph $G = (V \cup D, E)$, where V and D denote a set of customers and a set of depots respectively, and E denotes the set of weighted edges satisfying the triangle inequality. A vehicle with capacity k is located at each depot node and can be used to serve at most k customers. It is assumed here that the

number of customer nodes $|V|$ is no more than $|D| \times k$. The objective of the MDCVRP is to find a minimum cost set of tours covering all the customer nodes of V such that each tour contains at most k customers, and a distinct depot.

The known approximation results for the Vehicle Routing Problem with Multi-Depots [18, 67], considered the MDCVRP in the following settings (called MCVRP in [67]): the underlying graph G is defined similarly to that in the MDCVRP; the vehicle also has a capacity of k; however, in the MCVRP when a vehicle returns to its depot after servicing some customers, the same vehicle can start another round of servicing immediately. Because of the hard capacity constraint in the MDCVRP, the MDCVRP is a further generalization of the MCVRP.

A problem closely related to the MDCVRP, called the *Vehicle Dispatching Problem* (VDP), is studied by Krumke et al. in [60]. The VDP is defined similarly to the MDCVRP, with the only difference being that each vehicle will not return to its home base (depot).

Multi-Vehicle Scheduling Problem (MVSP)

The MVSP is a further generalization of the Single Vehicle Scheduling Problem (SVSP) in [53]. In the SVSP each vertex u is associated with a job j_u that has a release time $r(u)$ and a handling time $h(u)$. A job j_u can only be serviced after its release time $r(u)$ and the job need $h(u)$ time to finish. Again a vehicle is given to service the jobs, and the objective of the problem is to minimize the makespan. The makespan is the time when the vehicle completes all the jobs. In the MVSP the jobs are defined similarly as in the SVSP, however there are m identical vehicles to service the jobs. The objective is to minimize the makespan of all the vehicles. No capacity constraint is involved in the MVSP. We include the MVSP in Chapter 5 because our algorithm for the MVSP shares the same solution framework with that for the MDCVRP.

1.4 Approximations of metric spaces

In this thesis several approximation algorithms are designed for variants of the CVRP in trees. The motivation to work on trees comes from the following. An approximation algorithm with ratio α for the CVRP in trees implies an approximation algorithm with ratio $O(\alpha \log n)$ for the CVRP in general graphs [32]. This is due to the fact that an n-point metric space can be approximated by tree metrics with a *distortion* factor of $O(\log n)$ [32].

Let V be a set of points and let M be a metric space over V. Let u and v be two points of V. Denote $d_M(u, v)$ to be the distance between u and v in the metric M. The following definitions are from [6].

Definition 1.4.1. *A metric space N over V, α-approximates a metric space M over V if for every $u, v \in V$, $d_M(u, v) \leq d_N(u, v) \leq \alpha \cdot d_M(u, v)$.*

Definition 1.4.2. *A set of metric spaces S over V, α-probabilistically approximates a metric space M over V, if (1) for every $u, v \in V$ and $N \in S$, $d_N(u, v) \geq d_M(u, v)$, (2) and there exists a probability distribution over metric space $N \in S$ such that for every $u, v \in V$ $E(d_N(u, v)) \leq \alpha \cdot d_M(u, v)$.*

Definition 1.4.3. *A k-hierarchical separated tree (k-HST) is defined as a rooted weighted tree such that (1) the edge weight from any node to each of its children is the same, (2) and the edge weights along any path from the root to a leaf are decreasing by a factor of at least k.*

The following theorem is established in [32].

Theorem 1.4.4 (Fakcharoenphol, Rao and Talwar)**.** *Every metric space M over V can be α-probabilistically approximated by the set of 2-HSTs, where $\alpha = O(\log n)$.*

By Theorem 1.4.4, an instance I of the CVRP in general graphs can be converted to an instance I' of the CVRP in trees. Let OPT_I denote the cost of the optimal solution of I, and let $OPT_{I'}$ denote the cost of the optimal solution of I'. Then $E(OPT_{I'}) \leq \beta OPT_I$, where $\alpha = O(\log n)$. Therefore a β-approximation for the CVRP in trees also gives an $O(\beta \log n)$-approximation for the CVRP in general graphs.

There are other reasons to investigate the CVRP in trees. In practice, tree networks arise naturally in river networks and pit mine railways [63]. Another example is tree road networks in logging areas of northern Canada, where road construction costs far exceed routing costs. There are also some applications where the general CVRP can be approximated by reducing the network to a tree. For example, Basnet et al. in [8] mentioned an application of routing milk tankers in a rural area of New Zealand where the underlying network can be approximated as a tree, since road building is extremely costly in that area due to its mountainous terrain.

1.5 Layout of the Thesis

We summarize the abbreviations of the problems investigated in this thesis in Table 1.1.

Table 1.1: Abbreviations of the problems investigated.

Abbreviation	Problem
VRP	Vehicle Routing Problem
CVRP	Capacitated Vehicle Routing Problem
CVRPPD	Capacitated Vehicle Routing Problem with Pickups and Deliveries
k-delivery TSP	k-delivery Traveling Salesman Problem
BWTSP	Black and White Traveling Salesman Problem
CCP	Cycle Covering Problem
CCPBL	Cycle Covering Problem with Bounded Length
pCCCP	p-constrained Cycle Covering Problem
MDCVRP	Multi-Depot Capacitated Vehicle Routing Problem
VDP	Vehicle Dispatching Problem
SVSP	Single Vehicle Scheduling Problem
MVSP	Multi-Vehicle Scheduling Problem

The thesis is organized as follows.

In Chapters 2 and 3 we discuss several approximation algorithms for the Capacitated Vehicle Routing Problem with Pickups and Deliveries(CVRPPD). Three variants of the CVRPPD, namely the k-delivery Traveling Salesman Problem (the k-delivery TSP), the Capacitated Dial-a-Ride problem, and the Black and White Traveling Salesman Problem (BWTSP), are investigated in this thesis. In all these variants, a vehicle with capacity k is given to service a set of customers. The objective is to find a minimum cost Hamiltonian Cycle to serve the customers, and at any point of the tour the vehicle capacity should always be satisfied.

In Chapter 2 we propose a rule called the *come-back* rule, for the k-delivery TSP and the Dial-a-Ride problem in trees. Under this rule the vehicle may come back and pick up more vertices before crossing an edge when some condition occurs. The triggering conditions vary for different problems. By using the *come-back* rule, we obtain an optimal algorithm for the k-delivery TSP on paths, and improve the approximation ratio to $\frac{5}{3}$ for the k-delivery TSP in trees. We also improve the approximation ratio to 2.5 for the Capacitated Dial-a-Ride Problem on paths.

In Chapter 3 we first show that BWTSP cannot be approximated if the length constraint is specified. We then present a 4-approximation algorithm for the BWTSP when only the cardinality constraint is specified. This algorithm is based on matching and König's theorem [58]. We also improve the approximation ratio slightly for the case where the number of white vertices is exactly k times the number of black vertices.

In Chapter 4, we address two NP-hard variants of the Cycle Covering Problem (CCP), namely the Cycle Covering Problem with Bounded Length (CCPBL), and the p-constrained Cycle Covering Problem (pCCCP). In the pCCCP, given an integer p, there should exist at most p cycles in the final solution. For the CCPBL, we show that a 4-approximation can be obtained by an application of the GW-algorithm [40]. A 2-approximation algorithm is given for the pCCCP.

Chapter 5 studies two variants of the Multi-Depot Vehicle Routing Problem (MDVRP), the Multi-Depot Capacitated Vehicle Routing Problem (MDCVRP), and the Multi-Vehicle Scheduling Problem (MVSP). A framework is proposed for both the MDCVRP and the MVSP in trees. In this framework, dynamic programming is used to indirectly decompose the original problem P into a set of disjoint subproblems. Solving these subproblems gives us the desired solution for P with constant factor approximation ratios in trees. As an arbitrary metric space can be α-approximated by tree metrics (2-HSTs) with $\alpha = O(\log n)$ [32], we obtain $O(\log n)$-approximations for the MDCVRP and the MVSP in general graphs.

Chapter 6 gives the conclusion of this thesis. Some future research directions are also discussed in this chapter.

Chapter 2

Capacitated Pickup and Delivery Vehicle Routing

2.1 Introduction

In the Capacitated Vehicle Routing Problem with Pickups and Deliveries (CVRPPD), we are given an undirected complete graph *G=(V, E)* with edge costs satisfying the triangle inequality, and each vertex v is associated with a pickup or delivery demand d_v. A vehicle, or a fleet of vehicles with limited capacity k, traverses the edges of G and serves all the pickup and delivery demands. During the traversal, the vehicle should never exceed its capacity. In this chapter, we investigate approximation algorithms for two models of the CVRPPD; more specifically, we concentrate on the k-delivery Traveling Salesman Problem (k-delivery TSP) and the Capacitated Dial-a-Ride Problem on paths and in trees.

2.1.1 k-delivery Traveling Salesman Problem

In the k-delivery TSP, we are given an undirected complete graph $G = (V, E)$, where a vertex set $V = V_p \cup V_d$ is partitioned into a set V_p of *pickup vertices* and a set V_d of *delivery vertices* and an edge set E with edge costs satisfying the triangle inequality. Each vertex of V_p is associated with an item, and each vertex of V_d requires one item. A vehicle with capacity k gathers products from pickup vertices, and delivers them to delivery vertices. All the items are identical; an item picked up from a vertex in V_p can be delivered to any vertex of V_d. The objective of the k-delivery TSP is to determine a minimum cost Hamiltonian

tour of G subject to the vehicle capacity constraint.

The k-delivery TSP is also called the Capacitated Pickup and Delivery Traveling Salesman Problem (CPDTSP) in [81]. Lim et al. [81] showed that the problem can be solved optimally in time $O(n^2/min(k,n))$ on paths, where n is the number of vertices of G. The authors proved in the same paper that the k-delivery TSP is NP-hard in the strong sense even for trees with height 2, using a reduction from the 3-partition problem. A 2-approximation algorithm for the k-delivery TSP in trees is later given in [68]. This algorithm follows the rule that the vehicle would continue to pick up (or deliver) items if possible.

The best known approximation ratio 5 for the k-delivery TSP in general graphs is due to Charikar et al. in [15]. The first constant factor approximation algorithm for the k-delivery TSP in general graphs is given by Chalasani et al. in [14]. These methods use TSP tours involving a subset of vertices as lower bounds for the k-delivery TSP in general graphs.

2.1.2 Capacitated Dial-a-Ride Problem

In the Capacitated Dial-a-Ride Problem, we are given an undirected complete graph $G = (V, E)$ with edge costs satisfying the triangle inequality, a depot d and a set of jobs, where each job is represented by a source vertex and a destination vertex in G. A vehicle with capacity k is used to serve each job by carrying the item (of one unit) from its source to its destination. The goal is to compute the shortest tour that starts from d and then returns to d after serving all the jobs. It is required that at any point of the tour the vehicle should never carry more than k items.

When $k = 1$, the Capacitated Dial-a-Ride Problem is also called the Stacker-Crane Problem. The Stacker-Crane Problem on paths can be solved optimally as shown in [4]. Frederickson et al. in [34] proposed a $\frac{9}{5}$-approximation algorithm for the Stacker-Crane Problem in general graphs. Later, they improved the ratio to $\frac{5}{4}$ for trees in [33].

For the Capacitated Dial-a-Ride Problem with arbitrary vehicle capacity, the best known approximation ratio $O(\sqrt{k}\log n)$ in general graphs is given by Charikar et al. in [16]. This algorithm is in fact for a type of special structured tree, called the *height balanced tree* [16]. It is proved in [16] that from an α-approximation for some problems in height balanced trees, we can obtain an $O(\alpha \log n)$-approximation for the same problems in general graphs. For paths, Krumke et al. [61] presented a 3-approximation algorithm. In this chapter we improve this approximation ratio to 2.5.

2.1.3 Our results and solution techniques

Our results are summarized as follows:

(1)For the k-delivery TSP on paths, we obtain an optimal linear time algorithm. This improves the running time $O(n^2/min(k,n))$ of the algorithm described in [81].

(2)For the Capacitated Dial-a-Ride Problem on paths, we improve the approximation ratio to 2.5 from 3 as in [61].

(3)For the k-delivery TSP in trees, we improve the approximation ratio to $\frac{5}{3}$. The best known approximation ratio 2 is due to Lim et al. [68].

In this chapter we propose a strategy called the *come-back* rule for the two models of the CVRPPD on paths and in trees. Our algorithms are all based on this strategy. Under the *come-back* rule, the vehicle would not be allowed to cross a particular edge e if some condition is met. Such conditions vary for different problems. In other words, the *come-back* rule states that, the vehicle may not consume its load immediately, but can come back and pick up more items for a better schedule. We use the k-delivery TSP on paths as an example. Assume that the vehicle has a capacity of 100, and the vehicle carries 40 items before visiting the next vertex u. If u is a delivery vertex, then the vehicle may still choose to not serve u at this time but to come back and pick up 60 more items before visiting u again. As a consequence, the vehicle may traverse several edges without servicing any jobs, even when it has a large load. This is somewhat contrary to our intuition.

Note that for all the problems we solved, we bound the costs of our solutions to the optimum of the *preemptive* version of these problems. In the *preemptive* version of these problems, the vehicle is allowed to drop all or part of its current load temporarily at some intermediate points. The vehicle will come back and collect the dropped items later. Our algorithms also answer the question of to what extent the optimum of the *non-preemptive* version of these problems are bounded to that of the *preemptive* version of these problems.

This chapter is organized as follows. In Section 2.2, we define some notations and discuss several lower bounds for the Capacitated Dial-a-Ride Problem and the k-delivery TSP on paths and in trees. In Section 2.3, we present a linear time optimal algorithm for the k-delivery TSP and use this algorithm as the first concrete example of the *come-back* rule. In Section 2.4, we show how to utilize the *come-back* rule to improve the approximation ratio to 2.5 (from 3 as in [61]) for the Capacitated Dial-a-Ride Problem on paths. In Section 2.5, we apply the *come-back* rule in a more complicated setting and improve the approximation

ratio to $\frac{5}{3}$ (from 2 as in [68]) for the k-delivery TSP in trees.

2.2 Lower bounds

In this section, we introduce three lower bounds known in the literature [15, 16, 81], called the *flow* bound, the *Steiner tree* bound and the *TSP* bound, for the k-delivery TSP and the Capacitated Dial-a-Ride Problem in trees. These lower bounds are used widely to design exact or approximate solutions for these problems. We also set some notations for later discussion. Some notations for these lower bounds are from [16, 81].

The *flow* bound for the Capacitated Dial-a-Ride Problem is defined as follows. Consider an edge $e = (u, v)$ of the path or the tree. Let $f_1(e)$ and $f_2(e)$ be the number of jobs that need to cross e from u to v and from v to u respectively. An observation is that in any transportation scheme, the number of times the vehicle moves from u to v equals the number of times the vehicle moves from v to u. Let $\lambda_e = \max\{f_1(e), f_2(e)\}$. As the vehicle has capacity k, in the optimal solution, e is crossed at least $2\lceil \frac{\lambda_e}{k} \rceil$ times. This is due to the fact that the vehicle has to return to the depot. We refer to this as the *flow* bound for the Capacitated Dial-a-Ride Problem.

The vehicle located at depot d is required to visit all the jobs in a single tour, therefore the jobs should be connected in some way. The *Steiner tree* bound for the Capacitated Dial-a-Ride Problem considers the cost of connecting the jobs. The formal definition of the *Steiner tree* bound is as follows. A vertex in the tree network is called *interest* if it lies on a path to serve a job. Let T be the Steiner tree connecting all such vertices. Then the *Steiner tree* bound states that any Dial-a-Ride tour must traverse every edge in T at least twice. We give the Capacitated Dial-a-Ride Problem on paths as an example. Assume in an instance we only have two jobs j_1 and j_2 defined on a path and the two jobs are non-overlapping, in the sense that the two path segments P_1 and P_2 obtained from servicing the two jobs are disjoint. Then after servicing one job, say j_1, the vehicle needs to be forwarded to the source of j_2 to serve j_2, otherwise the tour is not feasible. Therefore the edges connecting P_1 and P_2 will also have to be traversed twice in any transportation.

The *flow* bound for the k-delivery TSP is defined similarly as that for the Capacitated Dial-a-Ride Problem. Denote a subtree rooted at vertex u by T_u and the parent of u by $p(u)$. For each vertex u of G, we associate it with a label $a(u)$, which is set to 1 if u is a pickup vertex, and -1 if u is a delivery vertex. For paths, we define $n(u) = \sum_{t=1}^{u} a(t)$

to be the net number of items among vertices to the left of u (including u); for trees, we define $n(u) = \sum_{v \in T_u} a(v)$ to be the net number of items of all the vertices in T_u. Then any k-delivery TSP tour must traverse $e = (p(u), u)$ at least $2max\{\lceil \frac{|n(u)|}{k} \rceil, 1\}$ times. For an edge $e = (u, v)$, we define $n(e)$ to be equal to $n(u)$, if e is on a path and u is to the left of v, or e is in a tree and $v = p(u)$.

A lower bound called the *TSP* bound [15] is used for the k-delivery TSP in general graphs. The *TSP* bound simply uses the minimum-cost TSP tour involving a subset of vertices of the graph as a lower bound for the optimum. The 9.5-approximation algorithm in [14] and the 5-approximation algorithm in [16] are based on the *TSP* bound. There are no lower bounds known for the Capacitated Dial-a-Ride Problem in general graphs.

2.3 A linear time algorithm for the k-delivery TSP on paths

In this section, we propose a simple linear time algorithm for the k-delivery TSP on paths. It is also the first example of our *come-back* rule. As implied by the *flow* bound, an optimal solution of the k-delivery TSP on a path may contain $O(n^2/min(k, n))$ edges. That is the reason why the algorithm in [81] takes time $O(n^2/min(k, n))$, which is also the best possible running time if the final solution is constructed by explicitly enumerating its edge sequences. However, in this chapter we show that there is an optimal solution with a succinct representation, whose size is only linear to the number of vertices; in this way we are able to give an optimal algorithm for the k-delivery TSP on paths.

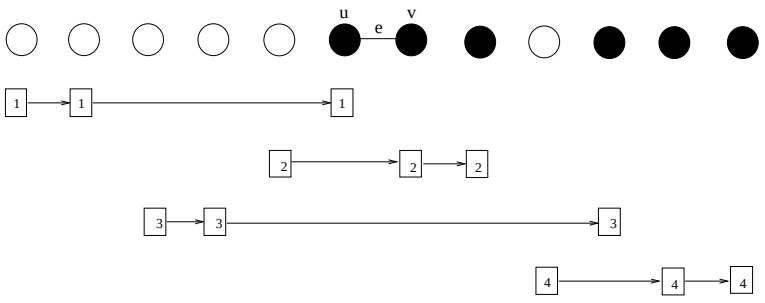

Figure 2.1: A succinct representation of an optimal solution for the k-delivery TSP on a path where k=2. A white circle represents a pickup vertex and a black circle represents a delivery vertex.

In the succinct representation (please see Figure 2.1 for an example), the solution comprises several subroutes, $r_1, r_2, \cdots, r_t$, where $1 \leq t \leq n$. Each vertex is marked to be picked up or delivered by one such subroute. For each subroute r_i, where $1 \leq i \leq t$, the vehicle travels along the path from the left-most vertex to the right-most vertex, which are marked by this subtour. In the meantime, the vehicle services all the vertices marked by the subtour. After reaching the right-most vertex, the vehicle comes back to the left-most vertex marked by the next subtour r_{i+1}, and the above process continues. It is clear that such a representation needs only $O(n)$ space.

Procedure come_back1(G**,** *start*, *direction*)
Input: An undirected graph $G = (V, E)$ defined on a path with nonnegative edge costs; *start* is the starting point of the path segment; the algorithm will process leftwards if *direction* $= 1$, and rightwards if *direction* $= -1$
Output: A succinct representation of an optimal solution for the path segment

```
1  Comment: initialization
2  init_stack(vehicle)
3  init_stack(repository)
4  route_number = i = 0
5
6  while 0 ≤ i ≤ |V| − 1
7        if the ith vertex is a pickup vertex
8              Comment: the vehicle carries less than k items
9              if vehicle.size() < k then
10                   vehicle.push(i)
11                   mark[i]=route_number
12             else
13                   repository.push(i)
```

```
14              endif
15        else if the i-th vertex is a delivery vertex then
16              if k − vehicle.size()=repository.size() mod k then
17                    m=repository.size() mod k
18                    if m=0 and vehicle.size() = 0 then
19                          m=k
20                    endif
21                    route_number=route_number+1
22                    for j=1 to m
23                          vehicle.push(repository.pop())
24                          mark[i]=route_number
25                    endfor
26              endif
27              v=vehicle.pop()
28              mark[i]=route_number
29              delivery[i]=v
30        endif
31        i = i+direction
32  endwhile
33  return mark, delivery
```

Figure 2.2: The algorithm for the k-delivery TSP on a path segment.

Our main algorithm, called come_back1, is for a path segment where $n(v)$ is 0 for the right-most vertex v, and greater than 0 for every other vertex on this segment. Its pseudo code is listed in Figure 2.2. The algorithm simply scans the segment from left to right, and maintains a route number and two stacks. The route number represents the number of the current subtour and is used to mark each vertex for a succinct representation. A stack, called *vehicle*, is used to simulate the behaviour of the vehicle, pushing a vertex to the stack has the same effect as loading one item into the vehicle, and popping from the stack means that the vehicle delivers one item. In the algorithm, a pickup vertex is put into this stack if the current vehicle load is less than k. Another stack, called *repository*, stores all the pickup vertices which cannot be served when the vehicle passes by (the vehicle is full). In the procedure, a pickup vertex is put into the *vehicle* stack if the current vehicle load is less than k. Otherwise it is stored in another stack called *repository*. Note that the vertices in the *vehicle* stack belong to the current subtour, and the vertices in the *repository* stack are serviced in later subtours.

The *come-back* rule applies when a particular delivery vertex u is to be visited. We define $S_1(t)$ and $S_2(t)$ to be the sets of items in the vehicle and the repository at time t respectively. Assume the vehicle is moving rightwards and at time t' it reaches u, which is the left end point of an edge $e = (u, v)$. For the k-delivery TSP on paths, the triggering condition of the *come-back* rule is whether $(|S_1(t')|+|S_2(t')|) \bmod k = 0$. In other words, the vehicle is allowed to cross e from u to v only if $(|S_1(t')|+|S_2(t')|) \bmod k \neq 0$. Otherwise, the current subtour is terminated and the vehicle comes back to pick up the topmost $(|S_2(t')| \bmod k)$ items in the repository. This can be implemented by increasing the route number by 1 and transferring the topmost $|S_2(t')| \bmod k$ items of the *repository* stack to the *vehicle* stack. To ease the analysis, we assume that the vehicle picks up these items on its way back from u (when the vehicle is moving leftwards).

The *come-back* rule guarantees that the solution produced by the algorithm abides by the *flow* bound for the k-delivery TSP. Therefore in this solution each edge e is traversed exactly $2\lceil \frac{n(e)}{k} \rceil$ times. Before proving this claim, we show that the solution may not be optimal if we do not come back when the triggering condition is satisfied. Consider the example in Figure 2.1. Let the first two delivery vertices be u and v in this example. Let the edge connecting u and v be e. When the second delivery vertex v is to be visited at time t, one item remains in the vehicle and three items are stored in the repository. Therefore $(|S_1(t)| + |S_2(t)|) \bmod 2 = 0$, and according to our rule the vehicle should come back and

pick up one more item before serving v. If otherwise we continue to dump the item in the vehicle to v in the first subtour, then e would have to be traversed at least 2 more times from left to right, since there are 3 pickup vertices still not served among the vertices to the left of v. Such a solution is not optimal as e is only allowed to be crossed $2\lceil\frac{n(e)}{2}\rceil = 4$ times.

The following lemma is crucial to prove that the algorithm indeed finds an optimal solution.

Lemma 2.3.1. *For an edge e of the path segment, let the subtours found by the algorithm passing through e be r_1, r_2, $\cdots$, r_m. Then in each of the routes r_3, $\cdots$, r_m, the vehicle crosses e from left to right with exactly k items. The vehicle carries the rest (more than k) of the items in r_1 and r_2.*

Proof. Let the left and right endpoints of e be u and v respectively. Notice that when the vehicle crosses e in r_1, all the delivery vertices to the left of v must have already been serviced. Otherwise, let v' be such a delivery vertex. Then the vehicle must carry nothing when visiting v', or else the vehicle can deliver one item to v'. But this is either contradictory to the definition of the path segment on which we are working, or is contradictory to the *come-back* rule. Therefore at time t_1 just before the vehicle crosses e from u to v in r_1, the number of items in the two stacks equals $n(e)$.

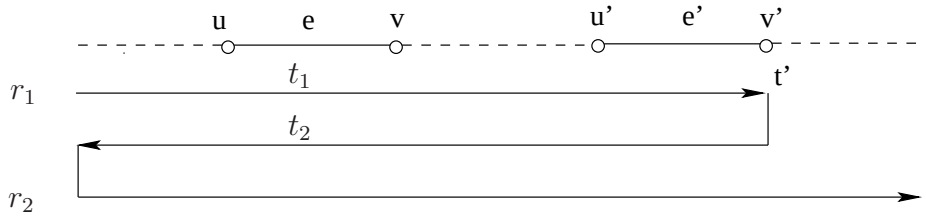

Figure 2.3: The first and second routes.

Let x be the residual $n(e)$ mod k. We already know that $|S_1(t_1)|+|S_2(t_1)|$ mod k equals x. The vehicle would cross e again in r_2, only if there exists an edge $e' = (u', v')$ such that v' is to the right of both v and u', and $|S_1(t')|+|S_2(t')|$ mod k equals 0 where t' is the time just after the vehicle services v' in r_1. In this case, the vehicle would cross e from v to u to pick up some more items. Let this time (just before the vehicle visits u) be t_2. Then we have $|S_2(t_2)| = |S_2(t_1)|$ and $(|S_1(t_2)| + |S_2(t_2)|)$ mod $k = 0$. The first equality holds because in the algorithm the repository is operated as a stack, so at time t_2 the items in $S_2(t_1)$ remain

untouched in $S_2(t_2)$, and the items in $S_2(t')$ but not in $S_2(t_1)$ (collected from some pickup vertices between v and v') must have already been transferred to the vehicle stack. The second equality holds since at time t' all the delivery vertices between v and v' must have already been serviced, so when the vehicle moves straight from v' to u (from time t' to t_2), the only change on the two stacks is to transfer some items from the repository stack to the vehicle stack. Therefore $|S_1(t_2)| + |S_2(t_2)| = |S_1(t')| + |S_2(t')|$. According to the definition of t', the equality $|S_1(t_2)| + |S_2(t_2)| \bmod k = 0$ holds.

We now claim that $|S_1(t_2)| = |S_1(t_1)| - x$. According to the definition of x, we have that $|S_1(t_1)| - x + |S_2(t_1)| \bmod k = 0$. Since $|S_2(t_2)| = |S_2(t_1)|$, we obtain that $|S_1(t_1)| - x + |S_2(t_2)| \bmod k = 0$. We know that $(|S_1(t_2)| + |S_2(t_2)|) \bmod k = 0$, and observing that both $|S_1(t_1)|$ and $|S_1(t_2)|$ are no more than k, we establish that $|S_1(t_2)| = |S_1(t_1)| - x$.

Therefore the vehicle would pick up another $k - |S_1(t_1)| + x$ items from the left of u before it crosses e again from u to v in r_2. So in r_2, the vehicle crosses e from left to right with k items; in r_1 and r_2, the vehicle crosses e from left to right with $|S_1(t_1)| + k - |S_1(t_1)| + x = k + x$ distinct items in total.

Similarly we can prove that the vehicle carries exactly k items when crossing e from left to right in each of the routes $r_3, \cdots, r_m$. We show the proof for r_3. Again the vehicle would cross e in r_3, only if there exists an edge $e'' = (u'', v'')$ such that v'' is to the right of both u'' and v', and $(|S_1(t'')| + |S_2(t'')|) \bmod k = 0$ where t'' is the time just after the vehicle services v''. Let t_3 be the time just after the vehicle crosses e from v to u in r_2 (back from u''). Similarly, as in the above discussion we have that $|S_1(t'')| + |S_2(t'')| = |S_1(t_3)| + |S_2(t_3)|$. Since the vehicle carries exactly $k + x$ items when crossing e from left to right in r_1 and r_2, we obtain that $|S_2(t_3)| \bmod k = 0$. With $(|S_1(t_3)| + |S_2(t_3)|) \bmod k = 0$, it follows that $|S_1(t_3)| \bmod k = 0$. The vehicle chooses to cross e from right to left for more items at time t_3, therefore the vehicle must be empty. The vehicle would pick up k items from the repository and cross e from left to right with k items in r_3.

□

The following theorem can be obtained by directly applying Lemma 2.3.1.

Theorem 2.3.2. *For an edge $e = (u, v)$ of the segment, the tour found by the main algorithm passes through e exactly $2\lceil \frac{n(e)}{k} \rceil$ times.*

Note that the vehicle may carry more than $(n(e) \bmod k)$ but less than k items when crossing an edge e from left to right in the first subtour. This is due to the fact that the

priority of loading items is given to the vehicle stack. However the algorithm is still optimal as we proved in Lemma 2.3.1 that the vehicle carries $k + (n(e) \bmod k)$ distinct items in total when crossing e in the first and second subtours from left to right.

Next we complete the final algorithm for the k-delivery TSP on a path. Our strategy is similar to that in [81] for the 1-delivery TSP on a path. We first identify the edges whose flow bound is zero. These edges, when removed, partition the path into disjoint subpaths. These subpaths can be determined in linear time. We then apply the come_back1 routine to each of these subpaths to obtain the optimal routes. Note that the vehicle will traverse each edge with 0 flow bound exactly twice, the same as in any optimal solution.

Therefore we establish the following theorem.

Theorem 2.3.3. *The optimal route of an instance of the k-delivery TSP can be determined in linear time.*

2.4 Approximation algorithms for the Capacitated Dial-a-Ride Problem on paths

In this section we show how to utilize the *come-back* rule to improve the approximation ratio to 2.5 for the Capacitated Dial-a-Ride Problem on paths. Our algorithm first decomposes the original problem into independent subproblems, then the *come-back* rule is applied for each subproblem. We also show that the best known 3-approximation algorithm (called the KRW algorithm in the following) can be integrated within our decomposition framework. The approximation ratio 2.5 is obtained by balancing the two solutions produced by our algorithm, and the KRW algorithm fused with our decomposition strategy.

2.4.1 A decomposition strategy for the Capacitated Dial-a-Ride Problem on paths

Our algorithm is iterative; in each iteration, the algorithm deals with a smaller size subproblem. The subproblem is formed by selecting a suitable subset of arcs (jobs) from the original graph. These arcs are removed from the graph once the subproblem is solved. The process is then repeated. We show the schema of the decomposition in Figure 2.4.

Let $J^{\rightarrow}$ be the set of jobs whose sources are to the left of their destinations. Similarly, let $J^{\leftarrow}$ be the set of jobs whose sources are to the right of their destinations. Our algorithm

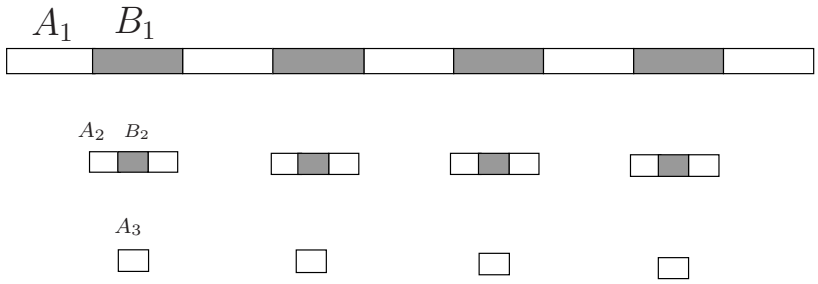

Figure 2.4: A decomposition of a capacitated Dial-a-Ride instance.

deals with the jobs in $J^{\rightarrow}$ and $J^{\leftarrow}$ separately. In Figure 2.4, it is assumed that all the jobs are from $J^{\rightarrow}$. The algorithm and the analysis can be easily adapted for the jobs in $J^{\leftarrow}$.

We illustrate the first two rounds of the decomposition in Figure 2.4. We say a job j crosses an edge e if and only if e is on the path to service j. We define an A-segment (B-segment) to be a maximal set of contiguous edges crossed by at most (more than) k jobs. The path then can be viewed as a sequence of alternating A-segments and B-segments. In Figure 2.4, A_1 denotes all such A-segments and B_1 denotes all such B-segments. In the first round of our algorithm, a set S of jobs are first selected by applying a job selection procedure on the original problem instance. S includes all the jobs crossing some edges of the A-segments, and possibly also some jobs which do not cross any edges of the A-segments. More details of the job selection procedure are described in Figure 2.5. After determining S, we relabel the A-segments and B-segments of the path with respect to the jobs in S. A feasible schedule is then computed to service the jobs in S based on the partition of the segments. The procedure to compute the schedule is described in Figure 2.6. In the second round, only the segments in B_1 need to be considered. Note that jobs in different segments can be scheduled separately. The sets A_2 and B_2 are formed similarly. In general, if letting $\lambda = \lceil \frac{\max_e(\lambda_e)}{k} \rceil$ (recall that λ_e is the maximum number of jobs crossing e in one direction), then there are λ such iterations. Therefore $\|A_1\| + \|B_1\| = L$ (L is the total edge cost of the path), and $\|A_i\| + \|B_i\| = \|B_{i-1}\|$, for $2 \leq i \leq \lambda$. Here $\|A_i\|$ and $\|B_i\|$ denote the total edge cost of A_i and B_i segments respectively.

We use the *flow* bound cost $2\sum_e \lceil \frac{\lambda_e}{k} \rceil c_e$ to approximate the Capacitated Dial-a-Ride Problem on paths. In iteration i, only a subset S of the jobs is selected to form the subproblem for this iteration. The set S includes all the jobs which cross some A-segments,

and some additional jobs that lie within the B-segments. The purpose of including these additional jobs is to ensure that, after the removal of these jobs from the graph at the end of each iteration, the number of crossings of the remaining jobs over any edge in a B-segment decreases by at least k. This guarantees that we can use $\|A_i\| + \|B_i\|$ as the lower bound to service these jobs. This also allows us to focus on designing approximation algorithms for the subproblem represented by S. Assume we have an α-approximation algorithm for each subproblem, namely it finds a transportation with cost at most $\alpha \cdot (\|A_i\| + \|B_i\|)$, then it is easy to see that the total cost of solving all the subproblems is bounded by $\alpha \cdot \sum_e \lceil \frac{\lambda_e}{k} \rceil c_e$.

The procedure in Figure 2.5 shows the details of selecting the jobs for each iteration of our algorithm.

The job selection procedure is for a particular B-segment B_i. It firstly places all the jobs crossing some A-segments of B_i (with respect to all the remaining jobs) in S. Then we check whether there is an edge e in a B-segment of B_i (with respect to all the remaining jobs) that is crossed by $k_1 < k$ jobs in S. If such an edge e exists, then we arbitrarily choose $k - k_1$ additional jobs crossing e. These jobs must have their two ends in the same B-segment.

The following simple lemma is guaranteed by lines 10-12 of the job selection procedure. It is crucial for designing approximation algorithms for the subproblems. After relabeling the A-segments and B-segments, it is equivalent to say that every selected job must cross an edge of some A-segment.

Lemma 2.4.1. *In each subproblem, a selected job must cross at least one edge which needs to be traversed by at most k selected jobs.*

Finally we need to show that our decomposition strategy is correct. We propose the following lemma without proof since it is directly implied by the selection procedure.

Lemma 2.4.2. *The number of crossings of the remaining jobs on an edge e before the beginning of iteration i, is at most $\lceil \frac{\lambda_e}{k} \rceil - (i-1)$.*

2.4.2 The come-back rule for the Capacitated Dial-a-Ride Problem on paths

With the clearly defined subproblems, we are ready to present an approximation algorithm for the Capacitated Dial-a-Ride Problem on paths. The input to the algorithm is a subproblem which can be viewed as a sequence of alternating A-segments and B-segments (starting

Procedure select_jobs(G, B_i)
Input: an undirected graph $G = (V, E)$ defined on a path, with nonnegative edge costs, and a B-segment B_i
Output: a set S of jobs

1 $J_A \leftarrow$ all the jobs crossing some A-segments of B_i
2 $J_B \leftarrow$ all the jobs not in J_A
3
4 $S \leftarrow J_A$
5 while $\exists e \in B_{i+1}$ s.t. e is crossed by $k_1 < k$ jobs in S do
6 $\quad S \leftarrow S + \{$ $k - k_1$ jobs crossing e in J_B $\}$
7 $\quad J_B \leftarrow J_B - \{$ the $k - k_1$ jobs selected in the previous step$\}$
8 endwhile
9
10 while $\exists j \in S$ s.t. every edge crossed by j is traversed by $\geq k$ jobs in S do
11 $\quad S \leftarrow S - \{$ j $\}$
12 endwhile
13
14 relabel the A-segments and B-segments with respect to S

Figure 2.5: A procedure for selecting a subset of jobs to form a subproblem.

and ending with A-segments), defined with respected to the selected jobs in S. Let $\|A\|$ and $\|B\|$ denote the total edge cost of all the A-segments and B-segments respectively, then we will show that the algorithm come_back2 outputs a transportation with cost bounded by $\|A\| + 3\|B\|$.

Algorithm come_back2(G, S_p)
Input: an undirected graph $G = (V, E)$ defined on a path, with nonnegative edge costs, and a sequence S_p of alternating A-segments and B-segments
Output: a schedule servicing all the selected jobs with cost at most $\|A\| + 3\|B\|$

```
1 for each segment s in S_p
2     if s is an A-segment then
3         service all the jobs originating or ending in s
4     else
5         service all the jobs ending in s
6         go to the starting vertex of s
7         pick up all the jobs originating in s
8     endif
9 endfor
```

Figure 2.6: The algorithm for solving a subproblem for the Capacitated Dial-a-Ride Problem.

The algorithm in Figure 2.6 adopts the *come-back* rule. The triggering condition is whether the ending (right-most) point of a B-segment is reached for the first time. So when the current segment s is an A-segment, the vehicle would pick up any new jobs it encountered in s and also deliver some of its load to s if these jobs end in s. When s is a B-segment, the vehicle would first service all the jobs ending in s till the endpoint of s is reached. Then the vehicle would come back to the starting point of s and begin to service all the new jobs with source points in s. An example of the schedule found by the algorithm is given in Figure 2.7.

It is critical to show that the come-back strategy produces a feasible transportation, namely the vehicle should never carry more than k items at any point of the route. We

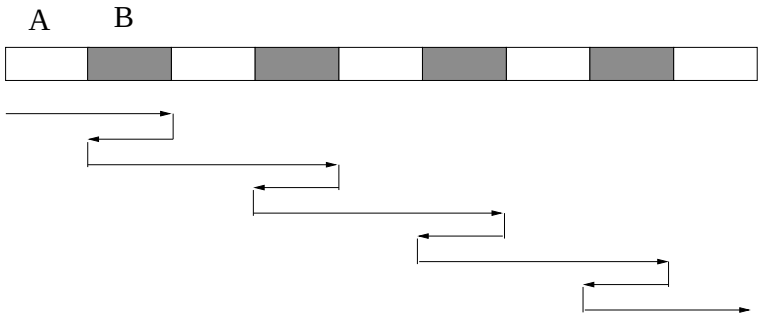

Figure 2.7: A Capacitated Dial-a-Ride example of the come-back strategy.

prove the correctness of the come-back strategy in Lemma 2.4.3.

Lemma 2.4.3. *The schedule produced by algorithm come_back2 is feasible.*

Proof. Assume the vehicle is passing through a segment s. According to the algorithm, all the jobs that end prior to the starting vertex of s should have already been serviced. Then the lemma holds if s is an A-segment, for the number of jobs that originate in or before s and cross some edges of s should be no more than k. This is due to the definition of an A-segment. If s is a B-segment, then the next segment s' must be an A-segment. Since the vehicle first services all the jobs that end in s, the number of items in the vehicle before collecting the new jobs in s, plus the number of new jobs originating from s, is at most k. This is because these jobs must cross some edges of s', and according to the definition of an A-segment, the number of such jobs should be less than or equal to k.

□

The algorithm come_back2 can also be adapted easily for the jobs in $J^{\leftarrow}$. After solving each subproblem by the algorithm, we obtain two sets of routes $R^{\rightarrow}$ and $R^{\leftarrow}$, where $R^{\rightarrow}$ represents the set of routes for the jobs in $J^{\rightarrow}$, and $R^{\leftarrow}$ represents the set of routes for the jobs in $J^{\leftarrow}$. We treat each route in $R^{\rightarrow}$ and $R^{\leftarrow}$ as a new job with source and destination being the starting and ending vertices of the route respectively, and use the optimal algorithm in [4] for the Capacitated Dial-a-Ride Problem on paths with unit vehicle capacity, to find another set of edges to connect these routes and form the final solution.

For the sake of completeness, we briefly explain the algorithm in [4]. Firstly a set of balancing directed edges are added for each path edge $e = (u, v)$, to make the number of

directed edges from u to v equals the number of directed edges from v to u. This is from the observation that in the optimal solution, if the vehicle traverses e in one direction, then it needs to visit e again from the other direction. The graph then becomes Eulerian, and the Euler tour of a strongly connected component yields an optimal transportation for the involved jobs. The second step of the algorithm is to connect these strongly connected components, by forming a new graph with the strongly connected components as its vertices, and the cost of the edge between two vertices corresponding to strongly connected components c_1 and c_2, equals the minimum distance between a vertex of c_1 and a vertex of c_2. It is not difficult to see that a minimum spanning tree of the new graph represents a set of path edges that connect the strongly connected components with the minimum cost.

The same algorithm can be applied to connect the routes in $R^{\rightarrow}$ and $R^{\leftarrow}$. According to our decomposition strategy, the number of crossings on a particular edge e is at most the number of crossings on e in the optimal solution. Thus some balancing directed edges can be added to the original graph to form strongly connected components among these routes. Clearly, the total cost to connect these strongly connected components is at most $2L$. Let $\|A_\lambda\| = max(\|A_\lambda^{\rightarrow}\|, \|A_\lambda^{\leftarrow}\|)$. Since the solution for the subproblems in an iteration under the decomposition strategy has a cost of at most $\|A_i\| + 3\|B_i\|$, the cost of the final solution of this algorithm is bounded by $\sum_{i=1}^{\lambda-1}(\|A_i^{\rightarrow}\| + 3\|B_i^{\rightarrow}\| + \|A_i^{\leftarrow}\| + 3\|B_i^{\leftarrow}\|) + \|A_\lambda^{\rightarrow}\| + \|A_\lambda^{\leftarrow}\| + 2L$. As implied in the proof of Theorem 2.4.5, this algorithm alone is a 3-approximation for the Capacitated Dial-a-Ride Problem on paths.

2.4.3 The KRW algorithm

We improve the ratio to 2.5 by balancing the solutions produced by our algorithm, and the KRW algorithm in [61]. In the following, we sketch their algorithm and show that it can be integrated with our decomposition strategy.

For the jobs in $J^{\rightarrow}$, the KRW algorithm first finds a set of transportation segments where each segment consists of several non-overlapping jobs. Thus a vehicle with unit capacity can service the jobs of one such segment by proceeding forward along the segment. We denote such a segment by an unit-capacity segment. The algorithm then groups every k unit-capacity segments to form a set $R^{\rightarrow}$ of new transportation segments, where each new segment can be serviced by forwarding a vehicle with capacity k. The segment set $R^{\leftarrow}$ for the jobs in $J^{\leftarrow}$ is formed similarly. Treating each segment in $R^{\rightarrow}$ and $R^{\leftarrow}$ as a new job, the problem can be transformed to the Capacitated Dial-a-Ride Problem on

paths with unit vehicle capacity. The second step of the algorithm in [61] is to use the algorithm in [4] to find a set of augmenting edges with minimum cost to paste the segments together. For each vertex v, define $v+1$ to be its right neighbor. Since the set of arcs $\{(v, v+1), (v+1, v) : 1 \leq v \leq n-1\}$ is a feasible set of connecting arcs, it is easy to see that the augmenting edges found in the second step are bounded by the optimal solution.

To obtain a 3-approximation, the cost of the edges in the segments needs to be bounded by twice the optimum. The main idea of [61] is to find two unit-capacity segments at a time, where each involved path edge is covered by at least one of the two segments. For a subset of jobs $J \subseteq J^{\rightarrow}$, let $\alpha(J)$ and $\omega(J)$ be the left- and right-most vertices involved in $J^{\rightarrow}$ respectively. Note that $\alpha(J)$ must be the source of a job, and $\omega(J)$ must be the destination of the same or another job. Let S_1 and S_2 be the two unit-capacity segments the algorithm finds in one iteration. Firstly, a job j_1 with $\alpha(J^{\rightarrow})$ as its source vertex is included in S_1. Then from the jobs whose source vertices lie between the source and destination of j_1, the job j_2 with the right-most destination is chosen to be in S_2. Similarly, among all the jobs whose source vertices lie between the destinations of j_1 and j_2, the job j_3 with the right-most destination is chosen to be included in S_1. By repeating such a process, the two segments are assigned a job at a time alternately until $\omega(J^{\rightarrow})$ is reached. Then the chosen jobs are removed from $J^{\rightarrow}$, and a new iteration begins if $J^{\rightarrow}$ is still not empty. It is mentioned in [61] that the above greedy selection strategy is crucial to the performance guarantee. An example of a run of the algorithm is shown in Figure 2.8. The right part of the figure shows the six unit-capacity segments constructed by the algorithm, and also the order of selecting the jobs.

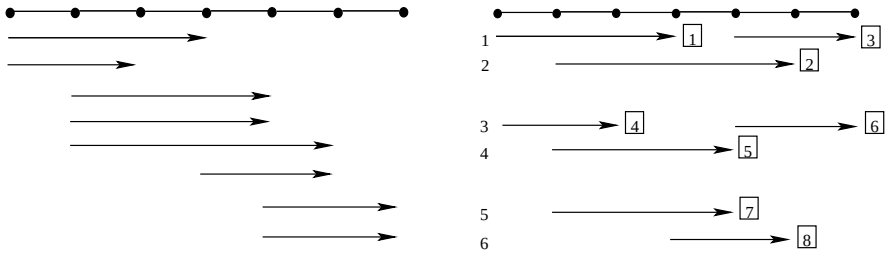

Figure 2.8: The 3-approximation KRW algorithm applied to a graph.

In one iteration, the KRW algorithm in [61] always finds two unit capacity segments, which together cover all the edges between the left-most source and right-most destination

vertices of the remaining jobs. However, there may exist some edges which are not covered by any job. In this case, the KRW algorithm would add some pseudo jobs covering these edges to the original graph. Therefore each unit capacity segment corresponds to a transportation where a vehicle with unit capacity moves straight from the left-most source to the right-most destination of the remaining jobs just before this segment is formed.

The KRW algorithm can be integrated seamlessly into our decomposition framework. Recall that in each subproblem formed by a run of the job selection procedure in Figure 2.5, a selected job j must cross at least one edge which needs to be traversed by at most k selected jobs. This property allows us to bound the cost of the solution produced by the KRW algorithm. More details are given in the proof of Lemma 2.4.4.

Lemma 2.4.4. *For a subproblem with A-segment set A_i and B-segment set B_i, the solution given by the KRW algorithm has a cost bounded by $2\|A_i\| + 2\|B_i\|$.*

Proof. Assume $2k$ unit-capacity segments have been created in a run of the KRW algorithm. Let S be the set of jobs crossing a particular edge e. According to the KRW algorithm, e must be covered by every two unit-capacity segments found by the KRW algorithm. Notice that pseudo jobs are added only if all the jobs crossing e have already been consumed. Therefore if $|S| \leq k$, all the jobs in S must be serviced within the $2k$ unit-capacity segments. Since in a subproblem a job must cross an edge that needs to be traversed by at most k selected jobs, every job is included in the $2k$ unit-capacity segments. This completes the proof. □

2.4.4 Performance analysis

Our final solution is the tour with the smaller cost from the two tours produced by the algorithm come_back2 and the KRW algorithm in [61]. We prove that the approximation ratio of this tour is 2.5 in Theorem 2.4.5.

Theorem 2.4.5. *The cost of the transportation with smaller cost from the two tours produced by the algorithm come_back2 and the algorithm in [61], is bounded by 2.5 times the optimum.*

Proof. As mentioned above, the cost of the tour after running algorithm come_back2, is at most

$$\sum_{i=1}^{\lambda-1}(\|A_i^{\rightarrow}\| + 3\|B_i^{\rightarrow}\| + \|A_i^{\leftarrow}\| + 3\|B_i^{\leftarrow}\|) + \|A_\lambda^{\rightarrow}\| + \|A_\lambda^{\leftarrow}\| + 2L \qquad (1)$$

Since $\|A_1^{\rightarrow}\| + \|B_1^{\rightarrow}\| = L$, $\|A_\lambda^{\rightarrow}\| = \|B_{\lambda-1}^{\rightarrow}\|$, and $\|A_i^{\rightarrow}\| + \|B_i^{\rightarrow}\| = \|B_{i-1}^{\rightarrow}\|$ for $2 \leq i \leq \lambda - 1$, we have

$$\sum_{i=1}^{\lambda-1}(\|A_i^{\rightarrow}\| + 3\|B_i^{\rightarrow}\|) + \|A_\lambda^{\rightarrow}\| + L = 2L + \sum_{i=1}^{\lambda-1} 3\|B_i^{\rightarrow}\|$$

Similarly,

$$\sum_{i=1}^{\lambda-1}(\|A_i^{\leftarrow}\| + 3\|B_i^{\leftarrow}\|) + \|A_\lambda^{\leftarrow}\| + L = 2L + \sum_{i=1}^{\lambda-1} 3\|B_i^{\leftarrow}\|$$

Thus (1) is less than or equal to

$$4L + \sum_{i=1}^{\lambda-1} 3\|B_i^{\rightarrow}\| + \sum_{i=1}^{\lambda-1} 3\|B_i^{\leftarrow}\|. \qquad (2)$$

By combining the main idea in [61] and our decomposition strategy, the cost of the solution is bounded by

$$\sum_{i=1}^{\lambda-1}(2\|A_i^{\rightarrow}\| + 2\|B_i^{\rightarrow}\| + 2\|A_i^{\leftarrow}\| + 2\|B_i^{\leftarrow}\|) + 2\|A_\lambda^{\rightarrow}\| + 2\|A_\lambda^{\leftarrow}\| + 2L \qquad (3)$$

Eliminating $\|A_i^{\rightarrow}\|$ and $\|A_i^{\leftarrow}\|$ as above, (3) is less than or equal to

$$6L + \sum_{i=1}^{\lambda-1} 2\|B_i^{\rightarrow}\| + \sum_{i=1}^{\lambda-1} 2\|B_i^{\leftarrow}\|. \qquad (4)$$

The optimal solution has a cost at least twice

$$\begin{aligned}\|A_1^{\rightarrow}\| + 2\|A_2^{\rightarrow}\| + \cdots + \lambda\|A_\lambda^{\rightarrow}\| =& \|A_1^{\rightarrow}\| + 2(\|B_1^{\rightarrow}\| - \|B_2^{\rightarrow}\|) + \cdots + \lambda\|A_\lambda^{\rightarrow}\| \\ =& \|A_1^{\rightarrow}\| + 2\|B_1^{\rightarrow}\| + \|B_2^{\rightarrow}\| + \cdots + \|B_{\lambda-1}^{\rightarrow}\| = L + \sum_{i=1}^{\lambda-1} \|B_i^{\rightarrow}\|\end{aligned}$$

Similarly, the optimal solution has a cost at least twice of $L + \sum_{i=1}^{\lambda-1} \|B_i^{\leftarrow}\|$.
Since $\sum_{i=1}^{\lambda-1} \|B_i^{\rightarrow}\| \leq max(\sum_{i=1}^{\lambda-1} \|B_i^{\rightarrow}\|, \sum_{i=1}^{\lambda-1} \|B_i^{\leftarrow}\|)$, we have

$$OPT \geq 2L + \sum_{i=1}^{\lambda-1} \|B_i^{\rightarrow}\| + \sum_{i=1}^{\lambda-1} \|B_i^{\leftarrow}\|. \qquad (5)$$

Adding (2) and (4), the total cost of the two solutions is at most

$$10L + \sum_{i=1}^{\lambda-1} 5\|B_i^{\rightarrow}\| + \sum_{i=1}^{\lambda-1} 5\|B_i^{\leftarrow}\| \qquad (6)$$

From (5) and (6), the approximation ratio of the final solution is at most

$$\alpha \leq \frac{1}{2} \cdot \frac{10L + \sum_{i=1}^{\lambda-1} 5\|B_i^{\rightarrow}\| + \sum_{i=1}^{\lambda-1} 5\|B_i^{\leftarrow}\|}{2L + \sum_{i=1}^{\lambda-1} \|B_i^{\rightarrow}\| + \sum_{i=1}^{\lambda-1} \|B_i^{\leftarrow}\|} = 2.5.$$

□

2.5 Approximation algorithms for the k-delivery TSP in trees

In this section we study the k-delivery TSP in trees. As shown in [81], the problem is NP-complete even in a tree of height 2. We present a $\frac{5}{3}$-approximation algorithm called the half-load algorithm for the k-delivery TSP in arbitrary trees. This algorithm is a $(\frac{3}{2} - \frac{1}{2k})$-approximation for the k-delivery TSP in trees of height 2. The half-load algorithm is also based on the *come-back* rule, albeit in a more complicated form.

The following notations are used to describe the algorithms. We call a branch positive (or negative) if this branch contains a positive (or negative) net number of items. Similarly, a vertex u is positive (or negative) if the subtree T_u is positive (or negative), and an edge $e = (p(u), u)$ is positive (or negative) if u is positive (or negative). For an edge $e = (p(u), u)$, we say some vertices are picked up from (or delivered to) e or u, if we pick up (or deliver) these vertices from (or to) the subtree T_u; Moreover, we say e has a load of $n(e)$, or e contains $n(e)$ vertices, if exactly $n(e)$ net number of items need to be picked up or delivered through e. Finally we denote the *flow* bound for a particular edge e (the number $2\lceil\frac{n(e)}{k}\rceil$) by FB_e. To ease the explanation, we denote $\frac{1}{2} \cdot FB_e$ by LB_e and we say the vehicle visits (crosses, traverses) e if the vehicle moves from $p(u)$ to u.

2.5.1 Exploring the symmetry of the k-Delivery TSP

Our improvements for the k-delivery TSP in trees explore the symmetry inherent in the problem. In the k-delivery TSP, the underlying graph has only two types of vertices. If we flip the type of each vertex, to get a new graph, say G', then any feasible solution of the k-delivery TSP on G' can be converted to a feasible solution of G with the same edge cost, by just reversing its edge directions. Therefore we have the following lemma.

Lemma 2.5.1. *Any feasible solution of the k-delivery TSP on G' can be converted to a feasible solution of G with the same edge cost, by reversing its edge directions.*

This property allows us to design approximation algorithms for the k-delivery TSP in trees in the following way. We partition the edge set E of the graph into two sets S and $E - S$. Consider an approximation algorithm which is particularly "good" to S, in the sense that in the tour produced by the algorithm, each edge e in S and $E - S$ is traversed at most α and β times LB_e respectively, for some $\alpha < \beta$. It is then easy to see that after flipping the type of each vertex and obtaining the second tour on G', each edge e would be crossed

at most $(\alpha+\beta)LB_e$ times on average. Therefore the approximation ratio is reduced to $\frac{\alpha+\beta}{2}$ from β after running the same algorithm on both G and G'. In the half-load algorithm, S is defined to be the set of positive edges of G. For the k-delivery TSP in trees of height 2, the proposed algorithm achieves $\alpha = 1$ and $\beta = 2$.

2.5.2 Preprocessing step of the half-load algorithm

The half-load algorithm contains two phases, the planning phase and the actual route generating phase. In the planning phase, the original graph is transformed to a multi-graph G'' as follows. Firstly, in G'' all the pickup and delivery demands are assumed to be located at the leaves. This can be done by adding a pseudo vertex u' for each non-leaf vertex u of G, and attaching an edge between u and u' with zero cost. Secondly, each positive tree edge $e = (p(u), u)$ is split into several pseudo edges, as shown in Figure 2.9. These pseudo edges are incident on $p(u)$ and u and their total load is equal to $n(e)$. Intuitively they record the number of items the vehicle would pick up during each of its visits to T_u. In a tree of height 2, an edge $e = (p(u), u)$ is split into $\lceil \frac{n(e)}{k} \rceil$ pseudo edges. Each of the first $\lfloor \frac{n(e)}{k} \rfloor$ edges has load of k. The last pseudo edge of e contains the residual of $n(e)$ mod k.

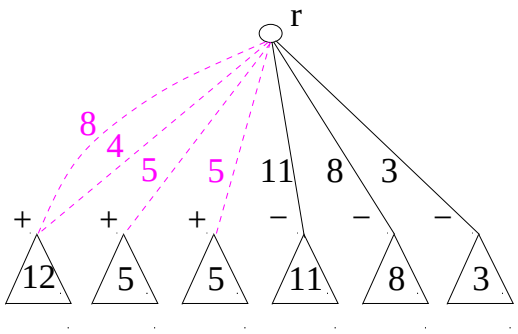

Figure 2.9: An example of building pseudo edges (k=8).

In the example in Figure 2.9, the capacity k is set to 8. A pseudo edge of the transformed tree is represented by a dashed line, and a subtree is represented by a triangle with a number showing the net number of items inside this subtree. Since the first positive tree edge has a load of more than k, it is split into two edges with loads 8 and 4 respectively. Other tree edges do not split, since they are either negative or their loads are less than k.

Given a list L of pseudo edges, we use a procedure called *merge* to produce a new set of pseudo edges, where each new edge (except possibly one) has a load more than $\frac{k}{2}$. We list the pseudo code of the merge procedure in Figure 2.10.

Procedure merge(L)
Input: a list L of pseudo edges
Output: a list of pseudo edges s.t. all except possibly one have loads $> \frac{k}{2}$

```
1 while ∃e_i, e_j ∈ L s.t. n(e_i), n(e_j) ≤ k/2 do
2       n(e_i) ← n(e_i) + n(e_j)
3       e_i.link ← concatenate the link lists of e_i and e_j
4       L ← L - {e_j}
5 endwhile
6
7 return L
```

Figure 2.10: The merge procedure for a list of pseudo edges.

This procedure repeatedly merges two arbitrary pseudo edges with loads $\leq \frac{k}{2}$, if possible. A set S of pseudo edges, which are merged together is treated as a single pseudo edge e', and when $n(e')$ items are requested from e', they are actually picked up from the pseudo edges in S. This can be implemented by creating a *group link* for each group of merged pseudo edges. In addition, each pseudo edge $e = (p(u), u)$ is associated with a link, called the *child link*, to remember where its load is from. More specifically, e is linked with a set S of edges which are incident on u and some children of u. In the second phase, when the load of e is requested, these links can be used to locate all the pseudo edges that ever participated in generating e (reachable from e by following the links).

Recall that the task of the planning phase is to split the positive tree edges into positive pseudo edges. The second actual route generating phase is based on these positive pseudo edges. We list the pseudo code of the planning phase in Figure 2.11.

The pseudo edges and their links are built recursively in a bottom-up fashion. Pseudo edges are firstly created from leaves, and then spread to higher levels of the tree. For a positive vertex u, given the lists L_+ and L_-, which contain all the positive (pseudo) and

Procedure half_load_plan(T_u)
Input: an undirected tree T_u, with nonnegative edge costs
Output: a multi-graph with positive pseudo edges

```
1  L+ ← ∅
2  if u is a leaf then
3        create a new pseudo edge e from to u its parent with n(e) = 1
4        return e
5  endif
6
7  for each positive child c of u
8        L+ ← merge(L+ ‖ half_load_plan(Tc))
9  endfor
10
11 L− ← all the negative tree edges from u to its children
13 e′ ← a new edge from u to p(u), which is linked with the edges of L− and several
   edges of L+ whose load is just enough to serve the edges of L−
16 remove these edges from L+
17
18 while L+ is not empty do
19       create a new edge e′ from u to its parent
20       e′ ← a copy of the head e of L+
21       L+ ← L+ − {e}
22 endwhile
23
24 L+ ← all the newly created pseudo edges
25 return merge(L+)
```

Figure 2.11: The planning phase of the half-load algorithm for the k-delivery TSP in trees.

negative edges from u to its children respectively, the algorithm firstly computes the internal schedule for the negative edges, by selecting a minimal subset S of edges from L_+, whose load is just enough to consume the total load of the edges in L_-. A new pseudo edge e' is created from u to its parent and is linked with the edges in S and L_-. The *merge* procedure is then applied to the rest of the positive pseudo edges, and each resulting pseudo edge e is propagated one level up by placing a copy e' of e between u and $p(u)$. We also link e' with e through the child link of e'. An example of building L_+ for a positive vertex u is shown in Figure 2.12.

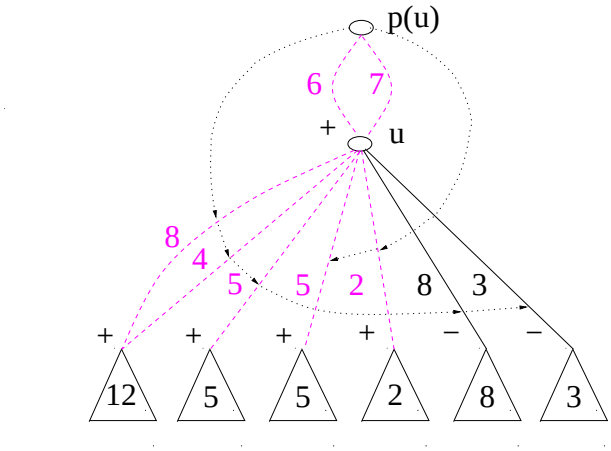

Figure 2.12: An example of building pseudo edges in the planning phase (k=8).

In this example, it is assumed that the pseudo edges from u to its children have already been built. Two pseudo edges e_1 and e_2 will be built for the tree edge $e = (p(u), u)$. Counting from left to right, let the 7 edges from u to its children be $e'_1, e'_2, \cdots, e'_7$. The load of e'_1 and e'_2 is enough to service the two negative edges e'_6 and e'_7. The net load of e'_1, e'_2, e'_6 and e'_7 is 1, therefore we create a linked list L that includes e'_1, e'_2, e'_3, e'_6 and e'_7. e_1 is then created and linked with the edges in L through the child link of e_1. Similarly, we create e_2 and link e_2 with the remaining two pseudo edges e'_4 and e'_5. When picking up the 6 and 7 vertices through e_1 and e_2 respectively in the second phase of our algorithm, we actually follow the child links of e_1 and e_2 to locate these vertices.

The merging step is necessary for our $\frac{5}{3}$-approximation algorithm for the k-delivery TSP in trees with arbitrary heights. It also keeps the number of positive pseudo edges in the

order of $O(\frac{n^2}{k})$. Implied by the approximation ratio, the time and space complexities of the *half-load* algorithm are both $O(\frac{n^2}{k})$. Please refer to Lemma 2.5.4 for more details.

2.5.3 Come-back rule for the k-delivery TSP in trees

The *come-back* rule of the half-load algorithm is applied in the scenario when we are given two lists L_+ and L_- and a vertex u, where the negative edges in L_- are from u to its children, and they must be serviced by the items from the positive pseudo edges in L_+. The pseudo code of the *come-back* rule for the k-delivery TSP in trees is listed in Figure 2.13. To ease the explanation, we assume that the tree is of height 2. This procedure also works for trees with arbitrary heights. The explanations for the general case will be given later in this chapter.

Procedures, *pickup* and *deliver*, are called in the algorithm. For trees of height 2, picking up from (delivering to) an edge $e = (p(u), u)$ can be implemented by simply picking up (delivering) the predetermined number of items from (to) the leaves of the branch. The vertices not going outside this branch with the vehicle can be serviced during the first visit of the vehicle to the branch.

The algorithm runs in iterations, and in each iteration we maintain two variables e_1 and e_2 pointing to the first edges of L_+ and L_- respectively. Recall that under the *come-back* rule, the vehicle may not be allowed to cross an edge e if some condition occurs. The triggering condition in this algorithm is whether the load of e_1 fits the remaining capacity of the vehicle. Therefore in each iteration of the algorithm, if the current load of the vehicle plus the load of e_1 is $\leq k$, then the vehicle will come back and pick up the load of e_1. Otherwise, the vehicle begins to deliver all or part of its current load to the head e_2 of L_-. Whenever the load of e_1 or e_2 is fully consumed, it is removed from L_+ or L_-. The above process continues until L_+ or L_- or both become empty.

An example is given in Figure 2.14 to show the effectiveness of our *come-back* strategy. In this example, $k = 8$, the input to the algorithm is a list L_+ which contains edges $e_1, e_2, \cdots, e_5$ with loads 6, 6, 6, 6, 2 respectively, and a list L_- which contains edges e'_1, e'_2, e'_3, e'_4 with loads 5, 7, 7, 7 respectively. It is not difficult to verify that only e'_3 will be traversed twice if following the *come-back* rule. Therefore $e_1, \cdots, e_5, e'_1, \cdots, e'_5$ will be traversed 1, 1, 1, 1, 1, 1, 1, 2, 1 times respectively. If we flip the types of the vertices and run the algorithm in 2.13, then only e_1 needs to be visited twice. In this case $e_1, \cdots, e_5, e'_1, \cdots, e'_5$ will be traversed 2, 1, 1, 1, 1, 1, 1, 1, 1 times respectively. However, if we follow the strategy in [16]

Procedure come_back3(α, L_+, L_-)
Input: an undirected tree of arbitrary height, with nonnegative edge costs; an initial vehicle load α; two lists L_+ and L_- contain some positive and negative edges respectively.
Output: a tour satisfying the capacity constraint

```
1  Comment: the vehicle may already have a load
2  while L+ and L- are not empty do
3        e1 ← the head of L+
4        e2 ← the head of L-
5
6        if α + n(e1) ≤ k then
7              Comment: pick up the vertices from e1
8              pickup(α, e1)
9              L+ ← L+ - {e1}
10       else if α ≥ n(e2) then
11             Comment: deliver to e2
12             deliver(L+, α, e2)
13             L- ← L- - {e2}
14       else
15             Comment: deliver all the vehicle load to e2
16             deliver(L+, α, e2)
17             n(e2) ← n(e2) - α
18       endif
19 endwhile
20
21 return
```

Figure 2.13: Come-back rule for the k-delivery TSP in trees.

and [68] where the vehicle would continue to pick up (or deliver) items if possible (which we call the *full-load* strategy in the following), then $e_1, \cdots, e_5, e'_1, \cdots, e'_4$ will be traversed 1, 2, 2, 1, 1, 1, 2, 2, 2 times respectively. It is not difficult to see that the *come-back* rule outperforms the *full-load* strategy clearly in this example.

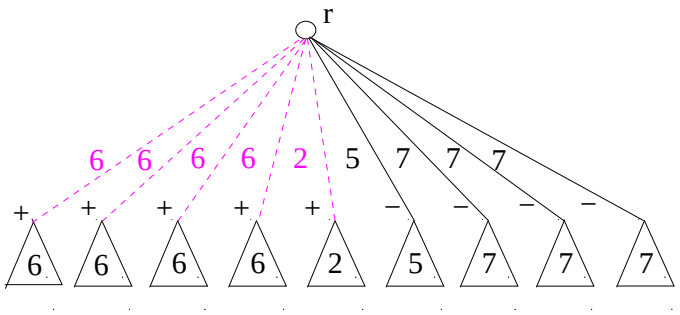

Figure 2.14: An example of the come-back strategy with k=8.

On average, $e_1, \cdots, e_5, e'_1, \cdots, e'_4$ will be visited $\frac{3}{2}$, 1, 1, 1, 1, 1, 1, $\frac{3}{2}$, 1 times respectively in the two tours after applying the *come-back* rule. We prove in Lemma 2.5.2 that the approximation ratio is $\frac{3}{2}$ of the come_back3 algorithm for the k-delivery TSP in trees of height 2.

Lemma 2.5.2. *The come_back3 algorithm in Figure 2.13 approximates the k-delivery TSP in trees of height 2 within $\frac{3}{2}$ of its optimum.*

Proof. It is not difficult to see that the vehicle always obeys the capacity constraint. According to the *come-back* rule, each positive pseudo edge in L_+ is only visited once. Therefore all we need to show is that after applying the algorithm, each negative edge in L_- is traversed at most twice.

In the algorithm in Figure 2.13, the vehicle starts to deliver its load, if and only if its load plus the load of the next positive edge is more than k. Because L_- consists of real negative tree edges, during every visit on $e \in L_-$, this check is enforced. Also note that if the vehicle starts to visit e for the first time, then the subsequent delivery tasks occur also on e, until all the load of e is satisfied. Thus during every two consecutive visits of the vehicle to a negative tree edge e, the vehicle carries more than k vertices. Let t be the number of traversals the vehicle makes on e, we show that $t \leq 2 * LB_e$. It is trivially true if $t = 1$, so in the following subcases, we assume the vehicle capacity $k \geq 2$.

Subcase 1: t is odd. Assume that for every two consecutive visits except the last one, the vehicle dumps $k+1$ pickup vertices to e in total. This is also the worst case our algorithm could have on e. Therefore $LB_e \geq \frac{t-1}{2} + 1 = \frac{t+1}{2}$, which is equivalent to $t \leq 2 * LB_e - 1$.

Subcase 2: t is even. Assume that for every two consecutive visits, but except the last two, the vehicle dumps $k+1$ pickup vertices to e in total. Therefore $LB_e \geq \frac{t-2}{2} + 1 = \frac{t}{2}$, which is equivalent to $t \leq 2 * LB_e$.

Because each positive edge becomes negative in G', in the two solutions, e is visited at most $3 * LB_e$. □

It is possible to further improve the ratio to $\frac{3}{2} - \frac{1}{2k}$ for the k-delivery TSP in trees of height 2. In this method, the negative edges are also split into pseudo edges, each with a load of no more than k, and these pseudo edges are sorted in non-decreasing order of their edge costs in L_-. When the vehicle starts to dump its load on a negative edge of L_-, it is known that the vehicle load l_1 plus the load l_2 of the next positive pseudo edge is more than k. The algorithm chooses to dump only $l_1 + l_2 - k$ vertices to the head of L_-, then the vehicle would pick up the l_2 vertices from the next positive edge, and dump the k vertices in the vehicle to the tail of L_-. This process is continued until all the vertices are served.

Assume a pseudo edge e of L_- is traversed m times, where $1 < m \leq k$. Then for each of the first $m-1$ visits, an edge e' in L_-, which has a cost no smaller than that of e, will be traversed optimally. In total there would be m edges involved when visiting e, and these edges will be traversed $2m-1$ times. In any solution these edges have to be traversed at least m times. Given that the cost of e is the smallest among that of these edges, the approximation ratio is $\frac{2m-1}{m} \leq 2 - \frac{1}{k}$. Since the negative edges will become positive and therefore will be traversed optimally after flipping the types of the vertices, we achieve an approximation ratio of $(\frac{3}{2} - \frac{1}{2k})$ for the k-delivery TSP in trees of height 2.

2.5.4 Pickup procedure for the half-load algorithm

Assume L_+ and L_- contain the edges in the first level of the transformed tree, then our solution can be expressed as *come_back3*$(0,\ L_+, L_-)$. The half-load algorithm would be complete if we have appropriate pickup and deliver procedures.

Both our pickup and deliver procedures run recursively. Given a positive pseudo edge $e = (p(u), u)$, the pickup procedure is applied when the vehicle with a load α tries to cross e to pick up $n(e)$ items from T_u. It is also assumed that the actual route has already been

Procedure pickup(α, e)
Input: a positive pseudo edge $e = (p(u), u)$; the vehicle load α satisfies $\alpha + n(e) \leq k$.
Output: routes showing how to pick up the $n(e)$ vertices from e.

```
 1 if u is a leaf then
 2       pick up u
 3       α ← α + 1
 4       return
 5 endif
 6
 7 Comment: edges in L+ and L− involved in generating e and
 8 they are from u to some children of u
 9 come_back3(α, merge(L+), L−)
10 for each edge e′ of L+
11       Comment: pick up the rest vertices
12       pickup(α, e′)
13 endfor
14
15 return
```

Figure 2.15: Pickup procedure for the half-load algorithm for trees.

built for the α vertices in the vehicle. Let S be the set of edges involved in generating e, and let the edges of S induce subtree T_S. Then the vehicle should service all the delivery vertices of T_S (by calling come_back3) and leave e (from u to $p(u)$) with $\alpha + n(e)$ items. Note that because of the non-preemptive nature, the vehicle may enter e and leave e with no items in common. An example of picking up vertices from a pseudo edge is given in Figure 2.16.

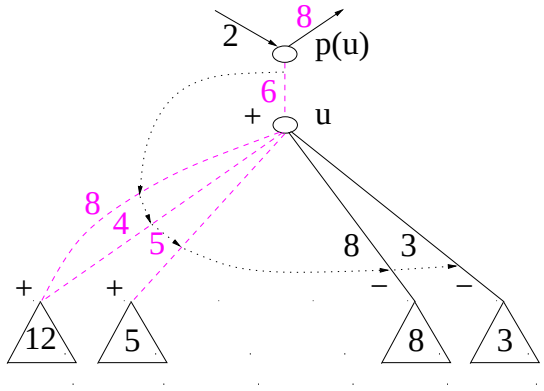

Figure 2.16: An example of picking up vertices from an edge $e = (p(u), u)$. $k = 8$ and the vehicle has an initial load of 2.

The example in Figure 2.16 is a subgraph of the example in Figure 2.12. We want to show how to pick up 6 vertices from the first positive pseudo edge $e_1 = (p(u), u)$ in Figure 2.12. Before visiting e_1, the vehicle is assumed to already carry 2 vertices; since $k = 8$, the vehicle should be able to leave $p(u)$ with 8 vertices. We only consider the schedule among the edges from u to its children involved in generating e_1. Let $e_1^+, e_2^+, e_3^+, e_1^-$ and e_2^- be the 5 edges from u to its children as shown in Figure 2.16, respectively from left to right. In this example, the vehicle visits these edges in the order of e_2^+, e_1^-, e_1^+, e_1^-, e_2^-, e_3^+. The 8 vertices leaving $p(u)$ with the vehicle are from e_1^+ and e_3^+.

The route implied by the planning phase for T_S assumes zero initial vehicle load. One may doubt whether the tour is still feasible if the vehicle already has a load. Recall that in the come_back3 procedure, further picking up through a positive pseudo edge e is allowed, only if the load of e fits the rest of the vehicle capacity. Therefore when the algorithm decides to pick up the $n(e)$ items, it is always guaranteed that the vehicle can service all the vertices in S and come back to $p(u)$ without breaking the capacity constraint. This also

shows the effectiveness of our planning phase.

2.5.5 Deliver procedure for the half-load algorithm

The *deliver* procedure is applied when the vehicle with α items tries to service a negative edge $e = (p(u), u)$. The pseudo code of the deliver procedure is given in Figure 2.17.

Procedure deliver(L'_+, α, e)
Input: α vertices in the vehicle need to be delivered to $e = (p(u), u)$; L'_+ is a list of pseudo edges not in T_u
Output: routes showing how the vertices in the vehicle are delivered to e

```
1  if u is a leaf then
2        service u
3        α ← α − 1
4        return
5  endif
6
7  Comment: edges in L+ and L− are from u to its children
8  L+ ← L+ ‖ L'+
9
10 come_back3(α, merge(L+), L−)
11 return
```

Figure 2.17: Deliver procedure for the half-load algorithm.

The deliver function is similar to the pickup function, with the major difference being that it needs an additional parameter L'_+. In the algorithm, L'_+ is maintained to contain some positive pseudo edges (not in T_u), from which $n(u)$ delivery vertices of T_u can be served. We call our algorithm the *half-load* algorithm for two reasons. One reason is that when a negative edge e is being serviced, according to the *come-back* rule, the vehicle should take more than k vertices during two consecutive visits to e. It can be viewed as that the vehicle is at least half full whenever it visits e. The other reason is that, when the vehicle

comes back to u to pick up more items, say from an edge e', the deliver procedure guarantees that the vehicle carries more than $\frac{k}{2}$ distinct items on its way from e' to e (except possibly one of the visits).

For the first reason, before servicing the delivery vertices of T_u, the edges of L'_+ are attached to the end of the list L_+ that consists of the positive pseudo edges from u to its children. According to the *come-back* rule, before visiting a negative edge e' of L_-, e.g. when the vehicle gathers the vertices from the last positive pseudo edge of the original L_+, the deliver procedure requires the information of the next positive pseudo edge of L'_+. This information will be used to decide whether the vehicle should come back and pick up some more supplies from outside T_u, in order to guarantee that e' is traversed no more than twice the optimum. Besides the first edge of L'_+, if some delivery vertices of T_u are still not serviced, more edges of L'_+ are further needed for T_u.

An example showing the necessity of stitching L'_+ and L_+ is given in Figure 2.18. For simplicity, only the residual $n(e)$ mod k is shown for each negative edge e in this example. It is easy to see that all the positive pseudo edges are needed for serving the negative edges incident on u.

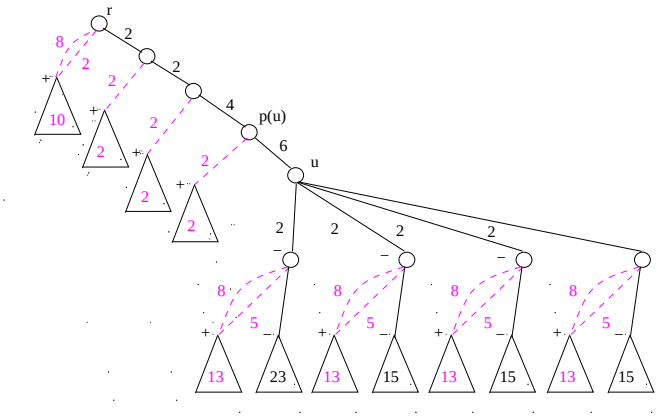

Figure 2.18: An example of delivering vertices to an edge $e = (p(u), u)$. $k = 8$ and r is the root of the tree.

For the second reason, after the stitching of L'_+ and L_+, the algorithm merges two positive pseudo edges if they both have loads no more than $\frac{k}{2}$. Thus it is maintained that, when the procedure come_back3 is called, L_+ only contains at most one pseudo edge with

load of no more than $\frac{k}{2}$. Since the algorithm runs recursively, the priority of merging is given to positive pseudo edges in higher levels of the tree. The merging and also the way of merging in the algorithm are important for the performance guarantee. They together ensure that if the vehicle comes back and picks up some more vertices before serving a negative edge e', then not only e', but also the other edges along the path are traversed no more than twice their optimum. A formal proof for the correctness of the strategy is shown in Lemma 2.5.3.

Lemma 2.5.3. *In the half-load algorithm, for two consecutive visits on a negative tree edge $e = (p(u), u)$ in the direction of $p(u) \rightarrow u$, the vehicle carries more than k vertices from outside T_u.*

Proof. In the *half-load* algorithm after stitching L'_+ and L_+, the new list L_+ includes the available positive pseudo edges (from outside T_u) which can be used to service e. In addition to the proof of Lemma 2.5.2, in trees of general heights, the crossing of e from $p(u)$ to u might be triggered by insufficient load of the vehicle when it tries to service a negative edge inside T_u. In this case the vehicle would come back and cross e from u to $p(u)$ to pick up more items from outside T_u. We claim that the pickup vertices (not in T_u) passing through e from $p(u)$ to u, are picked up consecutively from the edges of the list L'_+ starting from its head, and all except possibly one of the edges in L'_+ contain more than $\frac{k}{2}$ vertices.

The first part of the claim holds, because in the deliver procedure, the edges of L'_+ are attached to the end of L_+. So after all the positive edges in the original L_+ are consumed, the pickup vertices (not in T_u) crossing e from $p(u)$ to u, must be picked up consecutively from the edges of the list L'_+. The latter part of the claim holds, because the deliver procedure gives merging priority to higher level pseudo edges of the tree. So no matter L'_+ is obtained during the planning phase, or during an earlier call of the pickup or deliver procedure, the merging step has always been deployed on L'_+. This completes the proof. □

While the proof follows naturally from the algorithm, in the following we further explain why the way of merging in the deliver function is important for the performance guarantee. Suppose there is an edge in L'_+, say e', which has a load of no more than $\frac{k}{2}$ when the deliver function is called for T_u. This edge might be merged with another edge in L_+ to form a new pseudo edge containing more than $\frac{k}{2}$ vertices. When this new edge is requested, the vehicle will cross e from u to $p(u)$ to pick up the vertices in e'. For this traversal on e, the vehicle

contains only no more than $\frac{k}{2}$ vertices from outside T_u. However, it is the only exception, since all other positive pseudo edges of L'_+ contain more than $\frac{k}{2}$ vertices. These edges may also be created by merging some pseudo edges with loads of no more than $\frac{k}{2}$, but since the merging priority is given to pseudo edges in higher levels of the tree, they must have been formed and stay in L'_+ when the deliver function is called for T_u. Thus the vertices in one such edge are all from outside T_u, and they will cross e with the vehicle all at one time. Note that for the last visit on e, although the vehicle may contain more than $\frac{k}{2}$ vertices, it might dump less than $\frac{k}{2}$ vertices on e.

Consider the tree in Figure 2.18. Recall that only the residual $n(e)$ mod k is shown for each negative edge e. Let the four negative children of u be v_1, v_2, v_3, and v_4 respectively (from left to right). For the edge $e = (p(u), u)$, $LB_e = 2$. Assume the pseudo edges in Figure 2.18 are not merged. Before serving each negative edge $e_i = (u, v_i)$, where $1 \leq i \leq 4$, the first three positive pseudo edges in L_+ have loads 8, 5, 2. It is easy to see that in this example, the edge $e = (p(u), u)$ is traversed 2.5 times of LB_e if the come-back rule is deployed. If the priority of merging is not given to edges in higher levels of a tree, e.g., if each positive pseudo edge with load 5 is merged with a positive edge with load 2, then e will also be traversed 2.5 times LB_e.

The *half-load* algorithm alone is a 2-approximation for the k-delivery TSP in trees. The details of the performance guarantee for the *half-load* algorithm are shown in the proof of Lemma 2.5.4. They are important for the analysis of our final theorem.

Lemma 2.5.4. *The half-load algorithm is a 2-approximation for the k-delivery TSP in trees.*

Proof. First, it is not difficult to see that the tour is feasible, since in the tour all the vertices are served and the capacity constraint is always obeyed. Given an edge e of the tree, we prove that the number of traversals the *half-load* algorithm makes on $e = (p(u), u)$ is no more than $2 * LB_e$. Let r be the residual of e ($n(e)$ mod k). We have the following two cases:

Case 1: T_u is positive. Let the list L_+ contain all the pseudo edges on e. Each pseudo edge e' in L_+ is traversed only once in the algorithm, because the algorithm will not traverse this edge if the load of the vehicle plus the load of e' is more than the capacity. Thus the number of traversals the algorithm makes on e is just the number of pseudo edges in L_+. According to our merging rule in the planning phase, all pseudo edges, except possibly one,

must have loads more than $\frac{k}{2}$. Assume the vehicle carries exactly $\lfloor \frac{k}{2} \rfloor + 1$ vertices each time during the first $|L_+| - 1$ visits on e, and let the last edge in L_+ be e'. This is the worst case our algorithm could have on e. When k is odd, the vehicle should carry at least $\lfloor \frac{k}{2} \rfloor + 1$ vertices during the last visit on e. In the following we show that $|L_+| \leq 2 * LB_e - 1$ when k is even. The case when k is odd can be argued similarly.

Subcase 1: $|L_+|$ is odd. For the first $|L_+| - 1$ visits, the lower bound of the number of traversals on e (part of LB_e) is $\frac{|L_+|-1}{2}$, if excluding $|L_+| - 1$ vertices from these visits. There must be one additional visit for these excluded vertices and the vertices in e', e.g., if the number of these vertices equals to the residual of e. So in total $LB_e \geq \frac{|L_+|-1}{2} + 1 = \frac{|L_+|+1}{2}$, which is equivalent to $|L_+| \leq 2 * LB_e - 1$.

Subcase 2: $|L_+|$ is even. For the first $|L_+| - 2$ visits, if excluding $|L_+| - 2$ vertices, the lower bound of the number of traversals on e (part of LB_e) is $\frac{|L_+|-2}{2}$. The vehicle must also carry more than k vertices during the last two visits, so there must also be two additional visits in LB_e. Thus in total $LB_e \geq \frac{|L_+|-2}{2} + 2 = \frac{|L_+|+2}{2}$, which is equivalent to $|L_+| \leq 2 * LB_e - 2$.

Case 2: T_u is negative. Unlike the first case, in this case we may have two visits on e, where the vehicle dumps less than $\frac{k}{2}$ pickup vertices on e. However, according to Lemma 2.5.3, the total vehicle load is more than k during every two consecutive traversals on e. The proof then follows much in the same way as the proof of Lemma 2.5.2.

□

2.5.6 The $\frac{5}{3}$-approximation for the k-delivery TSP in trees of arbitrary heights

In this section, we prove that the smallest cost tour from three tours, namely obtained by applying the *full-load* algorithm on G (or G'), and the *half-load* algorithm on G and G', is bounded by $\frac{5}{3}$ times the optimum.

We explain the algorithm in [68] (which we call the *full-load* algorithm) as follows. For a vertex u, let $L_+ = \{c_1^+, c_2^+, \cdots, c_i^+\}$ and $L_- = \{c_{i+1}^-, \cdots, c_l^-\}$ be two lists consisting of the positive and negative children of u respectively. We also assume that in L_+ and L_- the vertices and the subtrees rooted at these vertices are sorted arbitrarily. In the *full-load* algorithm, the vehicle would pick up k items at a time *consecutively* from the subtrees $T_{c_1^+}, \cdots, T_{c_i^+}$, and deliver the k items *consecutively* to the subtrees $T_{c_{i+1}^-}, \cdots, T_{c_l^-}$. This also means that the vehicle services the subtrees in the sorted order, and when the vehicle

starts to pick up (deliver), the vehicle would continue to load (consume) items if possible. For the sake of completeness, we show our implementation of the *full-load* algorithm in the appendix.

The *full-load* algorithm has its advantages and disadvantages. The advantage is that after the vehicle gathers k items, it traverses several subsequent edges optimally. The disadvantage is that, before the vehicle is full, and after delivering these k vertices begins, some edges of the tree might have to be traversed twice their optimum. It is not difficult to see that the *half-load* algorithm exchanges its advantages and disadvantages with the *full-load* algorithm. This explains intuitively why we balance three tours as our final solution. We give our analysis of the approximation ratio in Lemma 2.5.5.

Lemma 2.5.5. *In the solution produced by the full-load algorithm, each edge $e = (p(u), u)$ is traversed at most $LB_e + 1$ times.*

Proof. Consider the case when e is positive. In the *full-load* algorithm, when the vehicle starts to visit an edge, it will traverse the edge consecutively for picking up vertices until all the load of this edge is serviced. Problems may occur if the vehicle has a load before its first visit on e. However, in this case the vehicle can carry less than k vertices only during its first and last visits on e. For all other visits, the vehicle picks up exactly k vertices from e. Thus in the solution e is visited at most $LB_e + 1$ times. The other case can be proved similarly. □

We formally prove the approximation ratio $\frac{5}{3}$ of our final solution in Theorem 2.5.6. This theorem also implies that the time and space complexities of the *half-load* algorithm are $O(\frac{n^2}{k})$.

Theorem 2.5.6. *The solution with the smallest cost from 3 tours, namely after applying the full-load algorithm on G (or G'), and applying the half-load algorithm on both G and G', is a $\frac{5}{3}$-approximation for the k-delivery TSP in trees of general heights.*

Proof. Given an edge e of the tree, we prove that the number of traversals in the 3 tours together on e is no more than 5 times its optimum. Let Sol_1 be the solution after applying the *full-load* algorithm on G (or G'), and Sol_2 and Sol_3 be the solution after applying the *half-load* algorithm on G and G' respectively. As shown in Lemma 2.5.5, e is traversed at most $LB_e + 1$ times in Sol_1. For Sol_2 and Sol_3, we have the following cases:

Case 1: e is positive. In this case, e is visited at most $2 * LB_e - 1$ times in Sol_2 according to Lemma 2.5.4. e is negative in G' , so e is crossed at most $2 * LB_e$ times in Sol_3 (established in the proof of Lemma 2.5.4). Summing together, e is traversed at most 3 times LB_e in Sol_1 and Sol_2, and therefore at most 5 times LB_e in the three solutions.

Case 2: e is negative. Because of the symmetry, if e is negative in G, its corresponding edge e' is positive in G', so similarly we can prove that the total number of traversals in Sol_1 and Sol_3 is bounded by thrice LB_e. Thus we have proved that e is traversed at most $5 * LB_e$ times in the 3 solutions. □

Chapter 3

The Black and White Traveling Salesman Problem

3.1 Introduction

In this chapter, we consider an extension of the classical Traveling Salesman Problem (TSP). The problem is defined on an undirected graph, $G = (V, E)$, where a vertex set, $V = V_B \cup V_W$, is partitioned into a set of *black vertices*, V_B, and a set of *white vertices*, V_W, and an edge set, E, with edge costs $w(e)$ for all $e \in E$ satisfying the triangle inequality. The Black and White Traveling Salesman Problem (BWTSP) is to determine a minimum cost hamiltonian tour of G subject to the following restrictions:

1. *Cardinality constraint* in which the number of white vertices on "black to black" paths is bounded above by a positive integer constant k, and

2. *Length constraint* in which the cost of any path between two consecutive black vertices is bounded above by a positive value L.

Clearly, the BWTSP reduces to the classical TSP when $L = k = \infty$, and is therefore NP-hard. An application of the directed BWTSP arises in short-haul airline operations ([78, 69]). The flight leg between two stations p and q is determined by a white vertex v_{pq} and a maintenance station s corresponds to a black vertex v_s. An arc represents a leg-leg, leg-maintenance, or maintenance-leg sequence. The problem is to determine a flying sequence such that the number of takeoffs and landings, as well as the total operating cost between

any two maintenance sequences are bounded as above. The undirected case has applications in telecommunications ([24, 87]). Cosares et al. [24] and Wasem [87] describe an application of the undirected BWTSP arising in the design of telecommunication ring networks, in which black vertices are "ring offices" and white vertices are "hubs". In order to achieve a survivable synchronous optical network (SONET) architecture, any two consecutive ring offices on the network must be separated by at most k hubs and a length not exceeding L. Another particular case of the BWTSP is the Vehicle Routing Problem (VRP) where each client has unit demand, the vehicle has capacity k, and maximal route length of the vehicle is at most L.

Attempts have been made to optimally solve the BWTSP for small size problems ([12, 87]). Ghiani, Laporte and Semet recently developed an exact branch-and-cut algorithm for the undirected case ([38]). Mak and Boland [69] have proposed a simulated annealing algorithm for the directed BWTSP and have applied it to instances involving 36 vertices. Bourgeois, Laporte and Samet [12] proposed five heuristic algorithms for the BWTSP, along with extensive computational comparisons.

In this chapter we are interested in designing efficient approximation algorithms with guaranteed performances for the BWTSP. We first show that the BWTSP cannot be approximated if the length constraint is specified. However, approximation algorithms with guaranteed performances can be designed when only the cardinality constraint is specified. The BWTSP with the cardinality constraint $k = 1$ occurs in routing papers with different names: the bipartite Traveling Salesman Problem or the k-delivery TSP where $k = 1$. Anily and Hassin [2] have shown a 2.5-approximation algorithm for another generalization of this problem, known as the Swapping Problem. Their algorithm finds a perfect matching M, consisting of edges that connect black and white vertices, and it uses the Christofides-Serdyukov heuristic [20] to find a tour, T, of the black vertices. The final route consists of visiting the black vertices in the sequence specified by the tour T, using the matching edges in M. Later, Chalasani and Motwani [14] developed a 2-approximation algorithm for 1-delivery TSP using combinatorial properties of bipartite spanning trees and matroid intersection. We expand the idea proposed in [2] and present a $(4-\frac{3}{2k})$-approximation algorithm, when the number of white vertices between two consecutive black vertices is bounded above by k. The bound can be slightly improved to $(4 - \frac{15}{8k})$, if the number of white vertices is exactly $k \cdot |V_B|$. When $|V_W| = 2 \cdot |V_B|$, the bound can be improved to 2.5.

The organization of this chapter is as follows: In Section 3.2, we show that the BWTSP is NP-hard when the length constraint is specified. Section 3.3 deals with the BWTSP when only the cardinality constraint is specified. Various approximation algorithms are provided for different variants of the cardinality constraint.

3.2 BWTSP with length constraint

In this section we show that the BWTSP with length constraint can not be approximated in the following theorem.

Theorem 3.2.1. *There is no polynomial time approximation algorithm for the BWTSP with length constraint unless $P = NP$.*

Proof. We first show that the following problem is NP-complete. Given a complete weighted graph G, with black and white vertices, satisfying the triangle inequality, determine whether G has a BWTSP route wherein the cost of the path between two consecutive black vertices in the cycle is no more than L. The above result then implies that the problem of designing approximation algorithms for the BWTSP is NP-hard, if the length constraint is specified.

Let us consider an instance of the Hamiltonian Path Problem. Let $G = (V, E)$ be the input graph with $|V| = n$. Consider the following graph G'. G' has n black vertices V_B and n copies of V (say, $V_1, V_2, \ldots, V_n$) which are all white. Suppose $L = n + 1$ and $k = \infty$. The cost of the edge between u and v is fixed as follows:

(i) if $u \in V_B$ and $v \in V_B$, *w(u,v) = 2*,

(ii) if $u \in V_B$ and $v \in V_i$, for any i, *w(u,v) = 1*,

(iii) if $u \in V_i$ and $v \in V_j$, for any $i \neq j$, *w(u,v) = 2*,

(iv) if $u \in V_i$, $v \in V_i$ and $(u, v) \in E$, *w(u,v) = 1*, and

(v) if $u \in V_i$, $v \in V_i$ and $(u, v) \notin E$, *w(u,v) =2*.

G' is a complete weighted graph satisfying the triangle inequality. Denote the vertices in V_B to be $v'_1, v'_2, \cdots, v'_n$, then in G' a black vertex $v'_i (1 \leq i \leq n)$ in fact corresponds to the vertex set V_i. We give an example of G' in Figure 3.1. In this example small circles with black and white fills represent black and white vertices respectively, and the sets of

vertices are illustrated by large dashed circles. A BWTSP tour with length constraint 5 is also shown in this figure by solid lines.

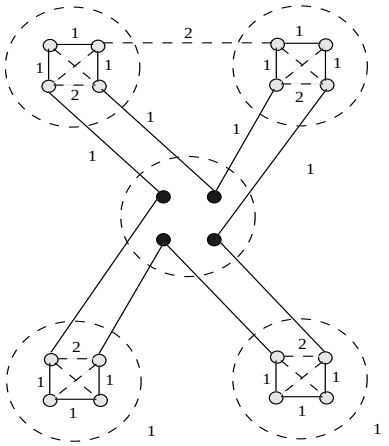

Figure 3.1: An example of the new graph G'.

In the following we prove that G' has a BWTSP route satisfying the length constraint, if and only if, the graph G has a hamiltonian path.

$\Longleftarrow$ assume that the graph G has a hamiltonian path P. Without loss of generality, let the path P visits the vertices in the order of $v_1, v_2, \cdots, v_n$. A BWTSP tour of G' satisfying the length constraint can be constructed as follows. For the vertices in $V_i (1 \leq i \leq n)$, we find the same path P, and connect the copies of v_1 and v_n in V_i to two black vertices v'_i and v'_{i+1} respectively. For the copy v_n in V_n, we connect it to v'_1. It is not difficult to verify that the length constraint is satisfied in this tour.

$\Longrightarrow$ assume that the graph G' has a BWTSP tour P' satisfying the length constraint. As the number of black vertices equals to the number of sets of white vertices and any edge between two white vertices is at least 1, there must be exactly n white vertices between two consecutive black vertices in P'. Since the edges between two vertices in V_i and V_j respectively where $i \leq j$ have large cost, the n vertices between any two consecutive black vertices must be from the same set of white vertices. Therefore any path segment between two consecutive black vertices defines a hamiltonian path in G.

Now assume we have a polynomial time approximation algorithm A for the BWTSP with the length constraint. As the length constraint is "hard", in the sense that given an arbitrary graph G, the solution produced by algorithm A on the graph G' constructed as

above must obey the length constraint. But from the above discussion, this also means that algorithm A would be able to compute a hamiltonian path of G in polynomial time. This completes the proof as the Hamiltonian Path Problem is NP-hard.

□

3.3 BWTSP with only the cardinality constraint specified

As it is impossible to find approximate solutions to the BWTSP when the length constraint is specified, we consider the case where only the cardinality constraint is satisfied. In other words, Given a graph $G = (V, E)$ where $V = V_B \cup V_W$, $V_B \cap V_W = \emptyset$, with the edges satisfying the triangle inequality, determine a minimum cost traveling salesman tour such that the number of white vertices between two consecutive black vertices in the tour is at most a given integer k.

Let $|V_B| = n$ and $|V_W| = m$. Without any loss of generality, we assume that $m \leq k \cdot n$, otherwise an instance does not have a feasible solution. Also note that if $m \leq k$, any Hamiltonian cycle satisfies the cardinality constraint and we get the classical TSP. Therefore, we assume that $k < m \leq k \cdot n$.

3.3.1 Lower bounds

In this subsection we establish two lower bounds for the BWTSP with the cardinality constraint. The first lower bound which is called the *TSP* bound in the sequel is as follows:

Lemma 3.3.1. *Given an instance of the BWTSP, the cost of the optimal traveling salesman tour that visits only a subset of vertices of the instance is at most the cost of the optimal BWTSP tour satisfying the cardinality constraint.*

Proof. This is due to the fact that the triangle inequality holds in the underlying graph, and the cardinality constraint is not considered in the optimal traveling salesman tour for only a subset of vertices of the graph. □

Let L^* be the length of the optimal tour of the BWTSP in $G = (V, E)$, satisfying the cardinality constraint. Let L_B^* and L_W^* denote the lengths of the optimal traveling salesman tour of the black and white vertices respectively. Applying Lemma 3.3.2, we have $L^* \geq L_B^*$ and $L^* \geq L_W^*$. Albeit simple, this fact is one of the basis of our algorithm. As shown in Section 3.3.2, our algorithm starts from a TSP tour involving all the white vertices.

The *TSP* bound is especially important for the general case of the BWTSP with the cardinality constraint, where $k < m < k \cdot n$. In this case there might be less than k vertices between two consecutive black vertices in the optimal BWTSP tour. This brings some difficulties to design approximation algorithms for the problem, as we don't have any information about the number of white vertices between two arbitrary black vertices. We circumvent this difficulty by introducing "dummy" white vertices to the original graph, and take advantage of the *TSP* bound to bound the cost of the traveling salesman tour involving these "dummy" vertices. Details will follow in Section 3.3.2.

The second lower bound is based on a structure called k-factor in this chapter. We define a k-factor of G as a set of edges $E_k \subseteq E$, such that for each black vertex $v \in V_B$, $\sigma(v) \leq k$, and for each white vertex, $v \in V_W$, $\sigma(v) = 1$, where $\sigma(v)$ is the number of edges of E_k incident on v. An example k-factor is given in Figure 3.2.

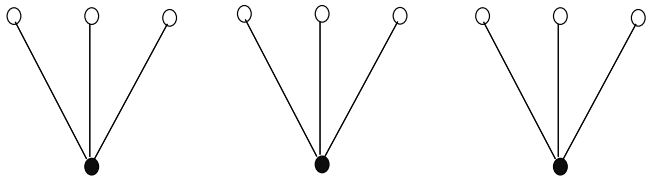

Figure 3.2: An example k-factor. k=3.

The second lower bound called the *k-factor* bound in the sequel is as follows:

Lemma 3.3.2. *Given an instance of the BWTSP, the cost of the optimal k-factor is at most $\frac{k}{2}$ times the cost of the optimal BWTSP tour satisfying the cardinality constraint.*

Proof. Given a tour T_{BW} of black and white vertices, a white vertex w is said to be closer to a black vertex u than to a black vertex v in T_{BW} if the number of vertices between w and u in T_{BW} is less than the number of vertices between w and v in T_{BW}. Suppose in the optimal tour we connect each white vertex to the closest black vertex. If black to black path in the optimal tour has an odd number of white vertices in between, the middle white vertex can be connected to either of the black vertices. Our strategy of connection is such that each black vertex is allowed to be connected to one such middle vertex. This way each black vertex is connected to at most k white vertices. Thus, the obtained set of edges is a k-factor. An example of such a k-factor is given in Figure 3.3.

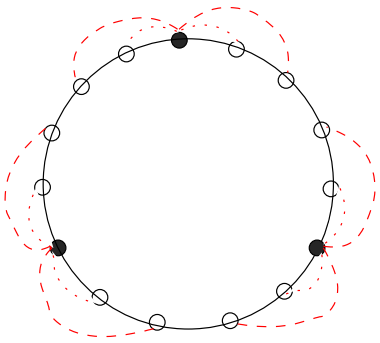

Figure 3.3: An example of the k-factor bound. k=4.

Let $L^*_{E_k}$ be the total cost of edges connecting the black and white vertices, using the above rule on the optimal tour of the BWTSP. We can estimate the cost of each edge between black and white vertices by using the triangle inequality. We claim that $L^*_{E_k} \leq \frac{k}{2}L^*$. Consider the following two cases.

Case 1: k is even. In this case the k-factor can be decomposed into $\frac{k}{2}$ different 2-factors each with cost at most L^*. In Figure 3.3, the dashed edges form one such 2-factor, and the dotted edges form another 2-factor. Therefore we establish that $L^*_{E_k} \leq \frac{k}{2}L^*$.

Case 2: k is odd. In this case the k-factor can similarly be decomposed into $\frac{k-1}{2}$ different 2-factors each with cost at most L^*. There are two ways of connecting the remaining white vertices to the black vertices. One way is to assign white vertices to black vertices in the clockwise order, and the other way is to assign the white vertices in the counter clockwise order. One of them must have cost at most $\frac{1}{2}L^*$. Therefore $L^*_{E_k} \leq \frac{k}{2}L^*$ also holds when k is odd.

□

3.3.2 Approximation algorithm when $k < m \leq k \cdot n$

We describe our approximation algorithm below. In the following we assume that in any tour of G there are at most k white vertices between two consecutive black vertices. Each step is followed by a brief discussion and implementation details if needed.

Algorithm $BWTSP(n, m)$

Step 1: Construct a minimum cost flow instance K as follows.

We first construct a complete bipartite graph K where one part of K contains the black vertices, and the other part contains the white vertices. We then add a vertex s to K, and connect s to each black vertex by an edge with cost of 0 and capacity of k. We further associate a positive integer $|V_W|$ (supply) with s, and -1 (demand) with each white vertex in K.

Step 2: Find a minimum cost k-factor E_k of K.

The minimum cost k-factor E_k can be computed after running minimum cost flow algorithms on K. The total cost of the edges in E_k, denoted by $\|E_k\|$, is at most $L^*_{E_k}$. Note that as there may not be enough white vertices, some black vertices in the k-factor may be assigned less than k white vertices.

Step 3: Transform graph G to graph $\hat{G}$ as follows.

Let h_v be the degree of black vertex v in the induced graph (V, E_k). For each black vertex v add $k - h_v$ "dummy" white vertices, and associate them with black vertex v in the following way. Each dummy white vertex is connected to v with edge cost zero. The dummy vertices are connected to other black and white vertices with edge costs being the same as the edge costs with v. This way we get $k \cdot n$ white vertices in total. It is easy to show that the triangle inequality is still satisfied in $\hat{G}$. Intuitively the dummy vertices are copies of their corresponding black vertices. An example of adding dummy vertices is shown in Figure 3.4.

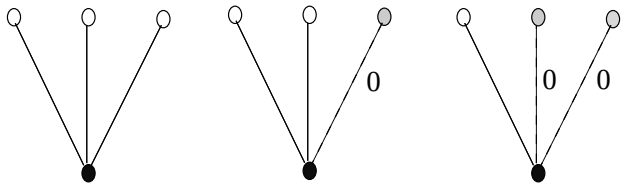

Figure 3.4: Adding dummy vertices. Dummy vertices are represented by small circles with grey fills. Dummy edges are represented by dashed edges in the minimum cost 3-factor.

Note that we add dummy vertices to the original graph only based on the minimum k-factor. To understand better our way of introducing the dummy vertices, we may try adding dummy vertices into the optimal BWTSP tour. In this way we can make the

number of white vertices equals exactly k times the number of black vertices, and also leave the optimal solution after introducing the dummy vertices unchanged. However as we do not have any information about the optimal BWTSP tour, it is not possible to design polynomial time algorithms in this way. In later steps we show that introducing dummy vertices based on the minimum k-factor will lead to a 4-approximation for the BWTSP with the cardinality constraint.

Step 4: We find a near optimal hamiltonian tour $\hat{T}_W$ in graph $\hat{G}_W = (\hat{V}_W, \hat{E}_W)$.

This tour fixes the order of the white vertices in the proposed tour of the BWTSP. We use the Christofides-Serdyukov algorithm [20] to obtain $\hat{T}_W$. Let $L_{\hat{T}_W}$ denote the length of $\hat{T}_W$. From the discussions in step 3, the optimal TSP tour $\hat{T}^*_W$ involving all the white vertices (including the dummy vertices) of $\hat{G}_W$ has the same cost as the optimal TSP tour of G, and therefore the cost of $\hat{T}^*_W$ is less than L^*. So we have $L_{\hat{T}_W} \leq 1.5L^*$.

Step 5: Partition the tour $\hat{T}_W$ into paths P_i on k vertices, $i = 1, 2, \ldots n$ of minimum cost.

Let $P_i = (u_{i1}, u_{i2}, \ldots, u_{ik})$, $i = 1, 2, \ldots, n$ be the minimum cost paths. Since there exist k different ways to partition tour $\hat{T}_W$, the total cost of the paths $P_i, i = 1, 2, \ldots, n$ is at most $\frac{k-1}{k}\ L(\hat{T}_W)$.

Step 6: Construct a bipartite multigraph H in the following way.

One part contains the vertices V_B and the other part contains n vertices $y_1, y_2, \ldots, y_n$ (called super nodes in the sequel) where the element y_i represents path P_i computed in step 5. Now (u, y), $u \in V_B$ and $y \in \{y_1, y_2, \ldots, y_n\}$, is an edge in H, if and only if there exists an edge (u, v') in E_k (computed in step 2) such that $u \in V_B$ and v' is a vertex of the path represented by y. Thus H is a bipartite multigraph and each vertex of H is of degree k.

An example of the graph is illustrated in Figure 3.5. In the figure super nodes corresponding to the paths obtained in step 5 are represented by large dashed circles. It is easy to see that each vertex in the graph has degree of 3.

Step 7: Find a proper edge coloring of H in k colors.

Each vertex in H has degree k. According to König [58], the chromatic index of a bipartite multigraph with maximum degree h is h. It is also shown in [58] that in

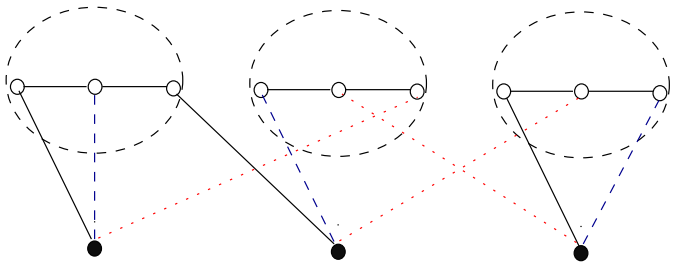

Figure 3.5: An example of the multigraph H. $k = 3$.

a bipartite multigraph, there exists a matching that saturates all the vertices with the maximum degree. Therefore, the edges of H can be colored using k colors, and the set of edges of the same color covers the vertices of H. For example, in Figure 3.5 the edges can be partitioned into 3 perfect matchings between two parts of the vertices (the solid edges, the dashed edges and the dotted edges). Let $C_1, C_2, \ldots, C_k$ be the partition of the set of edges of E_k where C_i contains all the i-colored edges. Note here that each C_i determines an assignment between black vertices and paths $P_1, P_2, \ldots, P_n$.

Step 8: Select the set C_q from $C_1, C_2, \ldots, C_k$ with minimum length.

Clearly, $\|C_q\| \leq \frac{1}{k}\|E_k\|$. Therefore, $\|C_q\| \leq \frac{1}{2}L^*$.

Step 9: Let v_i be the black vertex assigned to P_i. Construct two hamiltonian tours $R_1 = (v_1, u_{11}, u_{12}, \ldots, u_{1k}, v_2, u_{21}, u_{22}, \ldots, u_{2k}, \ldots, v_i, u_{i1}, u_{i2}, \ldots, u_{ik}, v_{i+1}, \ldots, v_n, u_{n1}, u_{n2}, \ldots, u_{nk})$ and $R_2 = (u_{11}, u_{12}, \ldots, u_{1k}, v_1, u_{21}, u_{22}, \ldots, u_{2k}, v_2, \ldots, u_{i1}, u_{i2}, \ldots, u_{ik}, v_i, \ldots, u_{n1}, u_{n2}, \ldots, u_{nk}, v_n)$. Remove the dummies from R_1 and R_2, and take the tour, say R, with the minimal cost as a BWTSP tour of G. As in the new graph the triangle inequality still holds, and the dummy vertices only appear on the paths of white vertices, they can be removed without increasing the total cost by taking appropriate shortcuts. We show an example of the two tours in Figure 3.6.

In Figure 3.6, the edges (between the black and white vertices) in the two tours are represented by solid and dashed lines respectively. The edges between white vertices remain the same in the two tours.

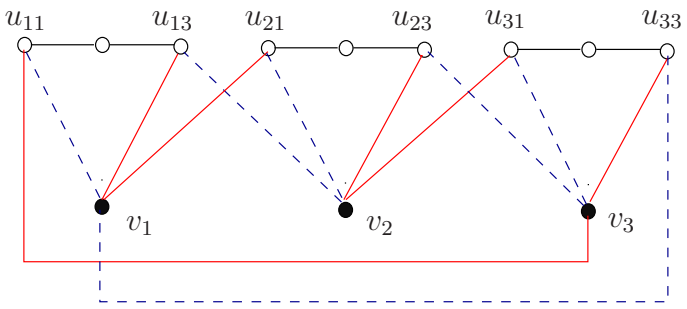

Figure 3.6: An example of the two tours constructed by algorithm BWTSP1. $k = 3$.

3.3.3 Performance analysis

We show the approximation ratio of the above algorithm in the following theorem:

Theorem 3.3.3. *The black and white traveling salesman problem with only the cardinality constraint can be approximated to within* $(4 - \frac{3}{2k})$*, where* k *is the maximum number of consecutive white vertices that can appear in the route.*

Proof. We can estimate the total cost of each tour by separately estimating the cost of the edges between the white vertices and between the black and white vertices. The total length of the edges between the white vertices is the total cost of n paths obtained in step 4 and is at most $\frac{k-1}{k}\ L_{\hat{T}_W}$.

We now estimate the total edge cost of the edges between black and white vertices in tour R. Let us first consider the total cost of edges connected to the black vertex v_i in routes R_1 and R_2. For route R_1, suppose $(v_i, u_{ij}) \in C_q$ for some j, $1 \leq j \leq k$, then

$$\begin{aligned} w(u_{i-1,k}, v_i) + w(v_i, u_{i1}) \quad \leq \quad & w(u_{i-1k}, u_{i1}) + L(u_{i1}, u_{i2}, \ldots, u_{ij}) + w(u_{ij}, v_i) + \\ & L(u_{i1}, u_{i2}, \ldots, u_{ij}) + w(v_i, u_{ij}). \end{aligned}$$

Here $L(P_i)$ indicates the cost of path P_i.

For tour R_2 we have

$$\begin{aligned} w(u_{ik}, v_i) + w(v_i, u_{i+1,1}) \quad \leq \quad & w(u_{ij}, v_i) + L(u_{ij}, \ldots, u_{ik}) + w(u_{ij}, v_i) + \\ & L(u_{ij}, \ldots, u_{ik}) + w(u_{ik}, u_{i+1,1}). \end{aligned}$$

Since $\min\{L_{R_1}, L_{R_2}\} \leq \frac{L_{R_1}+L_{R_2}}{2}$ we can now write

$$\begin{aligned}
\min\{L_{R_1}, L_{R_2}\} &\leq (4\|C_k\| + 2\frac{k-1}{k}L_{\hat{T}_W} + 2L_{\hat{T}_W})/2 \\
&\leq 2\|C_k\| + \frac{k-1}{k}L_{\hat{T}_W} + L_{\hat{T}_W} \\
&\leq (4 - \frac{3}{2k})L^*.
\end{aligned}$$

□

Running Time

The following computations dominate the running time of the algorithm.

1. Computing near optimal hamiltonian tour $\hat{T}_W$ of $\hat{G}_W = (V_W, \hat{E}_W)$ (Step 4).

   The running time of Christofides-Serdyukov's algorithm [20] to compute $\hat{T}_W$ is dominated by the Perfect Matching Problem in a subgraph of $\hat{G}_W$ which, in the worst case, takes $O(|V_W|^3)$ time[66]. Since the dummy white vertices are copies of some black vertices, it takes $O((m+n)^3)$ time to compute $\hat{T}_W$.

2. Computing a k-factor E_k (Step 2).

   This problem has been shown to be equivalent to a Minimum Cost Flow Problem in a graph involving $m+n$ vertices (excluding the source). Therefore, we can find the minimum cost k-factor of K in $O((m+n)^3)$ time.

3. Finding a proper edge coloring of bipartite multigraph H (Step 6).

   We can use the algorithm proposed by Cole and Hopcroft [23] to find an edge coloring of the bipartite multigraph in $O(m \log k)$ time.

Thus, the approximation algorithm proposed to solve the cardinality constrained $BWTSP$ in a graph with n black vertices and m $(k < m \leq k \cdot n)$ white vertices takes $O((m+n)^3)$ time to compute.

3.3.4 Approximation algorithms when $m = k \cdot n$

In this section we discuss ways of improving the performance bounds in the case when $m = k \cdot n$. Let E^*_{BW} be the $2n$ edges of the optimal BWTSP solution L^* connecting the black and white edges. The cost of the k-factor, described in Section 3.3.1, can be alternately bounded by $\|E^*_{BW}\| + \frac{k-2}{2} * L^*$. The argument for this new bound is almost the same once the edges of E^*_{BW} are separated. Also note that G and $\hat{G}$ are the same in Step 3 of $BWTSP(n, m)$. Let $\alpha = \frac{\|E^*_{BW}\|}{L^*}$, then we can write

$$\min\{L_{R_1}, L_{R_2}\} \leq 2\|C_k\| + \frac{k-1}{k} L_{\hat{T}_W} + L_{\hat{T}_W} \leq (4 - \frac{7}{2k} + \frac{2\alpha}{k})L^*.$$

We describe below another algorithm, called $BWTSP2(n, k \cdot n)$, where the number of white vertices between two consecutive black nodes is exactly k. Most of the steps of Algorithm $BWTSP(n, m)$ are the same in the new algorithm. Steps 3, 4 and 5 are replaced by step 3-4-5 and steps 8 and 9 are replaced by steps 8-9(a) and 8-9(b). As before, each new step is followed by a brief discussion and comments, if needed.

Algorithm $BWTSP2(n, k \cdot n)$

Step 3-4-5: Partition the white vertices into a set of paths, each containing exactly k vertices, using the algorithm of Goemans and Williamson [40].

As described in [40], the Exact Path Partitioning Problem (partitioning $G_W = (V_W, E_W)$ into disjoint paths, each path containing exactly k vertices) can be approximated to within $4(1 - \frac{1}{k})(1 - \frac{1}{|V|})$, i.e. within $4(1 - \frac{1}{k})$. Thus the set of paths we found in this step has a total cost of less than $4(1 - \frac{1}{k})(1 - \alpha)L^*$. This step can be performed in time $O(n^2 \log n)$ [40] which was later improved to $O(n^2)$ [35]. Let P_W be the set of paths.

Step 8-9(a): Find a near-optimal tour T_B in $G_B = (V_B, E_B)$.

Step 8(b): Construct a tour involving the edges of C_q, P_W and T_B.

This tour uses the edges of C_q and the edges of the paths in P_W, found in step 3-4-5, twice and the edges of T_B once. Thus we can get a feasible solution of the BWTSP with a cost less than $(1.5 + 8(1 - \frac{1}{k})(1 - \alpha) + \frac{2}{k}(\alpha + \frac{k-2}{2}))L^*$, i.e. less than $(10.5 - \frac{10}{k} - (8 - \frac{10}{k})\alpha)L^*$.

We show the approximation ratio of the above algorithm in the following theorem:

Theorem 3.3.4. *The Black and White Traveling Salesman Problem with only the cardinality constraint can be approximated to within* $(4 - \frac{15}{8k})$*, where* k *is the maximum number of consecutive white vertices that can appear in the route and the number of white vertices is exactly* k *times the number of black vertices.*

Proof. We now have two solutions with the costs of $(4 - \frac{7}{2k} + \frac{2\alpha}{k})L^*$ and $(10.5 - \frac{10}{k} - (8 - \frac{10}{k})\alpha)L^*$ respectively. We can choose the one with the smaller cost to be our final solution. It is interesting to verify that the two cost functions have the same value for all k when $\alpha = \frac{13}{16}$. Substituting $\frac{13}{16}$ for α, the approximation ratio is then $4 - \frac{15}{8k}$. The gain of $\frac{3}{8k}$ from our previous ratio of $4 - \frac{3}{2k}$ is meaningful for small k.

□

Further improvement when $m = 2n$

In the following we show that for $m = 2n$ $(k = 2)$, the approximation bound can be improved to 2.5. This is due to the fact that both the white-white edges (denoted by E^*_{WW}) and the black-white edges (denoted by E^*_{BW}) of an optimal BWTSP solution L^*_{BW} can be efficiently approximated. Note that there are n edges in E^*_{WW} and $2n$ edges in E^*_{BW}. The algorithm is formally described below.

Algorithm $BWTSP3(n, 2 \cdot n)$

Step 1: Construct a bipartite graph K in the following way. One part contains n black vertices of V_B and the other part contains all $2n$ white vertices of V_W. Only the edges of G connecting the black and white vertices are present in K.

Step 2: Find a minimum cost 2-factor E_2 of K.

Since E^*_{BW} is a 2-factor, then $\|E_2\| \leq \|E^*_{BW}\|$.

Step 3: Find a minimum cost perfect matching M of $G_W = (V_W, E_W)$.

Clearly $\|M\| \leq \|E^*_{WW}\|$. Consider the induced graph $(V, M \cup E_2)$. This graph is a collection of cycles. Each cycle involving black and white vertices is a tour (on a subset of vertices) satisfying the cardinality constraint. We represent each cycle by $CYCLE(v)$ where v is an arbitrary black vertex in the cycle. Let V_A be the set of

arbitrary black vertices chosen to represent the cycles. We note that $\|M \cup E_2\| = \|M\| + \|E_2\| \leq \|E^*_{BW}\| + \|E^*_{WW}\| = L^*$.

Step 4: Find a near-optimal TSP tour T_A of G_A, the subgraph of G induced by the vertices of V_A.

Step 5: Starting from an arbitrary black vertex $v \in V_A$ we make the round of all vertices of $CYCLE(v)$. After that we move to the next black vertex in tour T_A. The order in which the vertices appear in this walk define a tour T.

The cost of T is bounded by the total cost of the edges of $M \cup E_2$ and edges in tour T_A. So we have $L_T = L_{T_A} + \|M\| + \|E_2\| \leq \frac{5}{2}L^*$ and therefore, the algorithm $BWTSP3(n, 2n)$ is a 2.5-approximation algorithm.

Chapter 4

Variants of the Cycle Covering Problem

4.1 Introduction

Given an undirected complete graph $G = (V, E)$, where each edge $e \in E$ is associated with a cost c_e, the Cycle Covering Problem (CCP) is to find a set of disjoint simple cycles in G with the minimum total cost of their edges. These simple cycles together cover all the vertices in V. A cycle cover is also called a two-factor in graph theory, as each vertex has degree two in a cycle cover.

Finding cycle covers with minimum edge costs is a fundamental graph problem. It is a natural generalization of the matching problem, as a perfect matching is also a one-factor. Moreover, cycle covers are relaxations of Hamiltonian tours. The Hamiltonian Cycle Problem is just the CCP with the additional constraint that there should be only one cycle in the final solution. This fact has been utilized in designing combinatorial algorithms for the Traveling Salesman Problem (TSP) [39]. In these algorithms a cycle cover is firstly computed, and the cycles are then gradually patched together to form a Hamiltonian tour.

The CCP without any restrictions can be solved efficiently [30, 31]. In this thesis we investigate the CCP with two constraints, one is on the number of vertices in each cycle of a cycle cover, and the other is on the number of cycles appearing in a cycle cover. More specifically, we study the Cycle Covering Problem with Bounded Length k, and the p-constrained Cycle Covering Problem where the second constraint is involved. In this section

we introduce the two problems. As these problems are NP-hard, we present approximation algorithms with bounded performance guarantees for the two problems in later sections.

4.1.1 The Cycle Covering Problem with Bounded Length k

Given an integer k and an undirected complete graph $G = (V, E)$, where each edge $e \in E$ is associated with a cost c_e, the Cycle Covering Problem with Bounded Length k (CCPBL) is to find a set of disjoint simple cycles in G with the minimum total cost of their edges and these simple cycles together cover all the vertices in V. Each cycle is required to have at least 3 but at most k vertices.

While the CCP without any restrictions can be solved efficiently [30, 31], the CCPBL is NP-hard [44]. In [44], the complexity of the L-restricted Two-Factor Problem is considered. The L-restricted Two-Factor Problem is to compute a minimum cost set of disjoint simple cycles covering a given general graph G, subject to the constraint that each cycle must have a size in a given set L. In [44], it is proved that for almost all sets L, the L-restricted Two-Factor Problem is NP-hard. More specifically, if $L^- = \{3, 4, \cdots\} - L$, then the L-restricted Two-Factor Problem is NP-hard unless $L^- \subseteq \{3, 4\}$. Bodo Manthey in [70, 71] showed that the L-restricted Two-Factor Problem is APX-hard for undirected graphs with edge costs zero, one.

The only known approximation algorithm for problems related to the L-restricted Two-Factor Problem is due to Bodo Manthey et al. [70]. In this paper the authors studied a problem called the L-cycle Covering Problem, which is similar to the L-restricted Two-Factor Problem, with the only difference being that the L-cycle Covering Problem is defined on an undirected complete graph G with edge costs satisfying the triangle inequality. It is mentioned in [70] that it is sufficient to assume that L is finite. If we further assume that the greatest common divisor of the numbers in L is 1, then the approximation ratio of the algorithm in [70] can be expressed as $4(p_L + 1)$, where p_L is the Frobenius number of L. Given a set L of natural numbers with greatest common divisor 1, the Frobenius number is the largest natural number that cannot be expressed as a non-negative integer combination of the numbers in L.

In this thesis we are interested in the CCPBL. The length bound can be interpreted as the vehicle capacity, therefore this problem can be viewed as a variant of the CVRP with multi-vehicle. There are no approximation algorithms previously known for this problem. In this thesis we show that a 4-approximation for the CCPBL can be achieved by applying

a general approximation technique, called the GW-algorithm [40], designed for constrained forest problems. Our analysis of the approximation ratio will follow after a brief sketch of the GW-algorithm in later sections.

4.1.2 The p-constrained Cycle Covering Problem

The p-constrained Cycle Covering Problem is a variant of network design problems with downwards monotone functions. Given an undirected graph $G = (V, E)$ with non-negative edge weights, and a function $f : 2^V \rightarrow \{0, 1\}$, a network design problem can be formulated as the following integer program:

$$(IP)\ \ Min \sum_{e \in E} c_e x_e$$

subject to:

$$\sum_{e \in \delta(S)} x_e \geq f(S) \qquad \emptyset \neq S \subset V$$

$$x_e \in \{0, 1\} \qquad e \in E$$

where $\delta(S)$ denotes the cross edges between S and $V - S$, c_e represents the cost of an edge e, and x_e indicates whether the edge e is included in the solution.

A downwards monotone function f has the following properties: i) $f(V) = 0$; ii) $f(A) \geq f(B)$, if $A \subseteq B \subseteq V$. In this chapter we consider a class of network design problems that can be modeled as an integer program of the type (IP) with downwards monotone functions. A simple example of such problems is the minimum spanning tree problem, where the corresponding downwards monotone function f can be defined as $f(S) = 1$ if $S \subset V$ and 0 otherwise. Other examples include the Location-Routing Problem and the k-Cycle Covering Problem. We present below further details on the last two problems.

The k-Cycle Covering Problem: In this problem, we are given an integer k and an undirected complete graph $G = (V, E)$, where the edge costs satisfy the triangle inequality. The k-Cycle Covering Problem is to find a cycle cover of G with the minimum total cost where each cycle in the cycle cover contains at least k vertices. For the k-Cycle Covering Problem, the corresponding downwards monotone function f can be defined as: $f(S) = 1$ if S has less than k vertices, and 0 otherwise.

The Location-Routing Problem: In this problem [65] we are given an undirected complete graph $G = (V, E)$ and $D \subseteq V$ which denotes a set of depots. We assume here that the graph edge costs satisfy the triangle inequality. A non-negative cost (called opening

cost in the following) is associated with each depot in D. We need to select a set of depots from D and find a cycle cover of the vertices of G (a set of disjoint simple cycles that cover all the vertices in V). Each cycle in the cover must contain a selected depot. The goal is to minimize the cost of the cycle edges and the opening cost of the selected depots. Note that the unselected depots are treated as non-depot vertices in V.

We obtain a new augmented graph $G' = (V \cup D', E \cup E')$ from G as follows. For each depot node u in D, we add a copy u' of u to G', and create two new edges from u to u', each with cost equal to half of the opening cost of u. We define the downwards monotone function f for the Location-Routing Problem on G' as: $f(S) = 1$ if $S \subseteq V$, and 0 otherwise. With this function, every cycle in the optimal solution of (IP) on G' must have an edge connecting a depot vertex u to its copy u' in D'. This corresponds to opening the depot u in G.

In this chapter we consider adding an extra constraint (called the cardinality constraint in this chapter) to network design problems with downwards monotone functions. This constraint is on the number of connected components in the optimal solution. More specifically, given an integer p, we require that there should be at most p connected components in the optimal solution. For example, when the cardinality constraint is imposed for the k-Cycle Covering Problem, not only that each cycle in the cycle cover should contain at least k vertices, but also there should be at most p cycles in the cycle cover. It is easy to see that the new constraint has applications in vehicle routing, where only p vehicles are available to service the customers. To ease the explanation, we abbreviate the problems with the new cardinality constraint as *the p-constrained Path/Tree/Cycle Covering Problems*, which correspond to the cases when each connected component in the final solution is required to be a path/tree/cycle respectively. As no results are known for the p-constrained Path/Tree/Cycle Covering Problems, in the following we instead introduce previously known results for the network design problems with downwards monotone functions.

4.1.3 Three 2-approximation algorithms for network design problems with downwards monotone functions

The Location-Routing Problem is NP-hard, as it reduces to the classical TSP when there exists only one depot. The k-Cycle Covering Problem is polynomial time solvable for $k \leq 3$, and is NP-hard for $k \geq 4$ (Imielinska et al. [47] proved for $k = 4$, Vornberger [86] proved for $k = 5$, and Pulleyblank et al. [25] proved for $k \geq 6$). There exist three 2-approximation

algorithms for network design problems with downwards monotone functions:

GW-algorithm [40]: The proposed algorithm is a generalized approximation technique for the constrained forest problems. The GW-algorithm can solve a large class of graph problems. In addition to downwards monotone functions, it gives a $2(1-\frac{1}{n})$-approximation for problems which can be formulated as (IP) when f is a proper function or an uncrossable function. More details of the GW-algorithm will be provided in Section 4.2.1 and the Appendix.

Lightest-edge-first algorithm [41, 47]: The first approximation result on network design problems with downwards monotone functions is due to Goemans et al. [41]. In fact they generalized the algorithm by Imielinska et al. [47] for the k-Cycle Covering Problem, and showed that it is a 2-approximation algorithm for network design problems with downwards monotone functions. In this algorithm an arbitrary minimum spanning tree F of the graph is first selected. The edges of F are then examined in the ascending order of their edge costs. An edge e will be deleted from F if after its removal each connected component of F still has at least k vertices. In the sequel we refer to this algorithm as *the lightest-edge-first algorithm*.

Heaviest-edge-first algorithm [42]: This algorithm is similar to the lightest-edge-first algorithm, with the only difference being that the minimum spanning tree edges are examined in the non-increasing order of their edge costs. In this chapter we call this algorithm *the heaviest-edge-first algorithm*. In [64] Laszlo et al. proved that it produces results that are generally better than those produced by the lightest-edge-first algorithm.

4.1.4 Our results

We show that a 4-approximation is possible for the CCPBL by applying the GW-algorithm. We also generalize the algorithm in [41] and show that it is a 2-approximation for the p-constrained Cycle Covering Problem.

The GW-algorithm is a generalized tool for approximating network design problems. It is a powerful tool as it can solve any problem that can be formulated as (IP) with downwards monotone functions, proper functions, or uncrossable functions, etc [42]. However, it is not difficult to see that the GW-algorithm cannot be applied directly for the p-constrained Path/Tree/Cycle Covering Problems. To use the GW-algorithm, firstly we need to define the function f to formulate the problem as (IP). When defining the function f for the p-constrained Path/Tree/Cycle Cover Problems, given a set S of vertices we need to know

the edges incident on the vertices in S. This is due to the fact that we have to count the number of connected components involving these vertices. However, the domain of f in (IP) is only the power set of V. Moreover the function f needs to be pre-specified in order to formulate the problem as (IP). Therefore the GW-algorithm cannot be applied directly to the p-constrained Path/Tree/Cycle Covering Problems as they cannot be formulated as (IP).

In this chapter we generalize the heaviest-edge-first algorithm and show that it is a 2-approximation for the p-constrained Tree/Cycle Covering Problems, and a 4-approximation for the p-constrained Path Covering Problems. In order to achieve this, we first present a different combinatorial analysis of the approximation ratio for the heaviest-edge-first algorithm. This analysis is very different from that in [42] which uses a primal-dual approximation framework. We are then able to tackle the p-constrained Path/Tree/Cycle Covering Problems, and show a performance bound of 2 for the p-constrained Tree/Cycle Covering Problems, and a performance bound of 4 for the p-constrained Path Covering Problem. We assume for the p-constrained Path/Cycle Covering Problems that the graph G satisfies the triangle inequality property.

4.2 4-Approximation algorithm for the Cycle Covering Problem with Bounded Length k

In this section we define a function f for the CPPBL and show that the GW-algorithm with this function is a 2-approximation for this problem in general graphs. For the completeness of this thesis, we first give a brief introduction to the GW-algorithm.

4.2.1 The GW-algorithm

Michel Goemans and David P. Williamson developed the GW-algorithm [40]. This technique can be used to solve a large class of graph problems. Many classic algorithms, such as Dijkstra's algorithm for the shortest path problem, Edmond's algorithm for the Minimum-Cost Arborescence Problem, and the algorithms for the Minimum Spanning Tree Problem, can be explained by this technique. The technique mainly solves network design problems, and it particularly applies to the covering and partitioning problems, e.g. partitioning the graph nodes into cycles, paths, or trees. It is a generalization of the idea which was firstly

introduced by Agrawal, Klein and Ravi [1]. Our explanation of the GW-algorithm follows closely to those in [40, 41, 42, 80].

The GW-algorithm is for network design problems that can be formulated as (IP). Let (LP) be the relaxation of (IP), where the constraints $x_e \in \{0, 1\}$ are replaced by $x_e \geq 0$. Since the GW-algorithm is a primal dual method applied to approximation algorithms, we also list the dual of (LP) below:

$$
\begin{aligned}
&(DLP): \quad Max \ \textstyle\sum_{S \subset V} f(S) \cdot y_S \\
&\text{subject to:} \\
&\qquad \textstyle\sum_{S: e \in \delta(S)} y_S \leq c_e && e \in E \\
&\qquad y_S \geq 0 && \emptyset \neq S \subset V
\end{aligned}
$$

The GW-algorithm [40] is designed originally for proper functions. A function is called proper if it satisfies the following two properties:

1. Symmetry: $f(S) = f(V - S)$ for all $S \subseteq V$;
2. Maximality: if A and B are disjoint, then $f(A) = f(B) = 0$ implies $f(A \cup B) = 0$.

The pseudo code of the GW-algorithm for proper functions is included in Figure 7.4 in the Appendix. The GW-algorithm has two phases, namely the increasing phase and deleting phase. The algorithm is in a sense similar to Kruskal's algorithm for the Minimum Spanning Tree Problem [62]. Recall that in Kruskal's algorithm, initially each vertex forms its own cluster. Then the shortest edge $e = (u, v)$ between these clusters is included in the minimum spanning tree, and the two clusters where u and v reside are merged into one cluster. The process is repeated until $n - 1$ edges have been selected. One difference of the GW-algorithm from Kruskal's algorithm is that the sets (clusters) are not treated equally. In the GW-algorithm, a set could be either *active* or *inactive*. Given a function f, a set S is called *active* if $f(S) = 1$; and S is *inactive* if $f(S) = 0$. Recall that in (DLP), each set S is associated with a dual variable y_S.

The selection of an edge in the GW-algorithm is driven by the complementary slackness conditions. The complementary slackness conditions of (LP) and (DLP) are as follows:

$$
\begin{aligned}
&[primal] \quad x_e > 0 \ \Rightarrow \ \textstyle\sum_{S: e \in \delta(S)} y_S = c_e, \ \forall e \in E \\
&[dual] \qquad y_S > 0 \ \Rightarrow \ \textstyle\sum_{e \in \delta(S)} x_e = f(S), \ \forall S \subset V
\end{aligned}
$$

In the GW-algorithm, an edge e would be included in the primal solution during the increasing phase ($x_e > 0$), only if $c_e = \sum_{S:e\in\delta(S)} y_S$. The algorithm starts with a primal solution of $x_e = 0$ for $\forall e \in E$, and a dual solution of $y_S = 0$ for $\forall S \subset V$. In the increasing phase, edges between different sets are chosen greedily based on the current values of the dual variables. No edges between two inactive sets will be considered, and the priority of the selection is given to edges incident on two active sets. In the deleting phase, edges would be removed from F (the forest produced by the algorithm in the increasing phase), as long as the resulting forest stays feasible.

We now examine the algorithm in more detail. In the increasing phase, the algorithm is iterative. During each iteration, it maintains a list Γ, which contains all active connected components of F in the current iteration, and possibly some inactive connected components of F. During an iteration, an inactive component of Γ might be merged with some active connected component of F. Initially each vertex of G forms a separate component in Γ, and its corresponding dual variable is set to be 0. In the next iteration, the algorithm computes the maximum ϵ such that all the dual variables of the active components can be simultaneously increased by ϵ without violating any dual constraints (called *"the minimum violation set rule"* in [40]). The dual variables of the active components are then increased by ϵ at the same time (*"the uniform increase rule"*). After this increase, some edge e gets tight in the sense that $c_e = \sum_{S:e\in\delta(S)} y_S$. Let e be an edge connecting two components C_i and C_j in Γ. e is then added to F and a new component $C = C_i \cup C_j$ is added to Γ replacing C_i and C_j. Since e is in F, both C_i and C_j satisfy their corresponding primal constraints ($|\delta(C_i)\cap F| > 1 \geq f(C_i)$ and $|\delta(C_j)\cap F| > 1 \geq f(C_j)$). However, the new component C may now be an active component. The algorithm repeats the above iteration until no active components exist in Γ.

A function is uncrossable if it satisfies the condition that if $f(A) = f(B) = 1$ for any sets A and B, then either $f(A \cup B)$=$f(A \cap B)$=1 or $f(A - B)$=$f(A - B)$=1. Let $A = \{v \in V : f(\{v\}) = 1\}$. The GW-algorithm [40] for 0-1 proper functions can be modified to obtain the same approximation ratio $(2 - \frac{1}{|A|})$ for uncrossable functions. In the modified algorithm, the edges are removed in the reverse order that they were added to F. More explanations and an example of a run of the GW-algorithm can be found in the Appendix.

4.2.2 Applying the GW-algorithm for the Cycle Covering Problem with Bounded Length k

The approximation ratio of the GW-algorithm for 0-1 proper functions and uncrossable functions is $2 - \frac{2}{|A|}$, where $A = \{v \in V : f(\{v\}) = 1\}$. For the CCPBL, we define the function f as: $f(S) = 1$ if $|S|$ cannot be expressed as a non-negative integer combination of the numbers in $\{3, 4, \cdots, k\}$; $f(S) = 0$ otherwise.

Recall that we say a natural number a is admissible to a set S if a can be expressed as a non-negative integer combination of the numbers in S. Let S_k denote the set $\{3, 4, \cdots, k\}$, then we have the following lemma.

Lemma 4.2.1. *The Frobenius number is 5 for S_4, and 2 for all S_k with $k \geq 5$.*

Proof. Firstly it is easy to see that 5 is not admissible to S_4. Given a set S, consider putting $|S|$ identical items into $\lceil |S|/k \rceil$ bins. We put k items into each of the first $\lfloor |S|/k \rfloor$ bins, and the rest ($|S|$ mod k) items into the last bin. If $|S|$ mod $k \geq 3$, then we are finished as every bin has at least three items. Therefore the only exception occurs when $|S|$ mod $k = 1$ or $|S|$ mod $k = 2$. Assume $|S|$ mod $k = 1$. It is easy to see that $f(S) = 1$ if and only if two more items cannot be borrowed from the first $\lfloor |S|/k \rfloor$ bins while keeping the number of items in these bins still at least three. This can be determined by checking whether $(k-3)(\lfloor |S|/k \rfloor)$ is less than $3 - (|S| \bmod k)$. Therefore the above definition of the function f is equivalent to, $f(S) = 1$ if $|S|$ mod $k < 3$ and $(k-3)(\lfloor |S|/k \rfloor) < 3 - (|S| \bmod k)$; $f(S) = 0$ otherwise.

The function $(k-3)(\lfloor |S|/k \rfloor)$ is increasing with respect to $|S|$, therefore when $k = 4$, $(k-3)(\lfloor |S|/k \rfloor) \geq |S|$ mod k holds for all $|S| > 5$. Thus the Frobenius number of S_4 is 5. It is also easy to verify that $(k-3)(\lfloor |S|/k \rfloor) \geq 2$ for all $k \geq 5$ when $|S| > k$, therefore the Frobenius number of S_k ($k \geq 5$) is 2. □

Note that the Frobenius number of S_3 can be arbitrarily large, as S_3 only contains a single number 3. In this case the size of each cycle in the optimal solution of the CCPBL must be exactly 3. This is a special case of the Exact Partitioning Problem in [40]. In the Exact Partitioning Problem, for a given k we must find a set of vertex-disjoint trees/cycles/paths of size k that cover all vertices. Goemans and Williamson et al. in [40] gave a $4(1-\frac{1}{k})(1-\frac{1}{n}))$-approximation for the Exact Tree/Cycle Partitioning Problems by using the GW-algorithm. Therefore in this chapter we only consider the case when $k \geq 4$.

When $k \geq 5$, it is easy to verify that the function f defined above is an uncrossable

function. Therefore in $2(1-\frac{1}{n})$-approximation we can compute a forest where the number of vertices in each connected component is admissible to S_k with $k \geq 5$. We can then obtain a $4(1-\frac{1}{n})$-approximation for the CCPBL with $k \geq 5$ by following the same strategy and analysis for the Exact Partitioning Problem in [40].

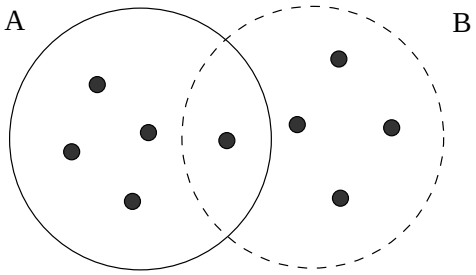

Figure 4.1: An example showing that f is not uncrossable.

In the following we focus on the CCPBL with $k = 4$. The function f defined above is not a downwards monotone function, as there exist two sets A and B, where $|A| = 4, |B| = 5$ and $A \subseteq B$, $f(A) = 0$ but $f(B) = 1$. It is also not a 0-1 proper function, as for two sets A and B, where $|A| = 2, |B| = 3$ and $A \cap B = \emptyset$, $f(A) \neq f(B)$ and $f(A \cup B) = 0$. Moreover, the example in Figure 4.1 shows that f is also not an uncrossable function. In this example, $|A| = |B| = 5$ and therefore $f(A) = f(B) = 1$, however, as $A \cap B = 1$, we have $f(A \cap B) = 1$ but $f(A \cup B) = 0$, and $f(A - B) = f(B - A) = 0$.

It follows that the CCPBL with $k = 4$ cannot be modeled by the three types of functions solvable by the GW-algorithm. However in Theorem 4.2.2 we show that the GW-algorithm is a 4-approximation for this problem.

Theorem 4.2.2. *The GW-algorithm can be used to give a 4-approximation for the Cycle Covering Problem with Bounded Length with $k \geq 4$.*

Proof. We have shown a 4-approximation for the CCPBL when $k \geq 5$. In the following we prove that the theorem holds when $k = 4$.

To prove this approximation ratio, we show that in 2-approximation, the GW-algorithm will produce a forest where the number of vertices in each connected component is admissible to S_4. We first examine the effect of increasing the dual variable y_S of a component S in Γ. Increasing y_S by ϵ would not affect any edges inside S, on the contrary, it contributes ϵ to

each edge of $F' \cap \delta(S)$ (recall that F' is the final solution after the deleting phase), because such edges are chosen by the GW-algorithm and they also belong to the cut of S to $V - S$. So although we only increase y_S by ϵ, the contribution of this increase to the final primal solution is $|F' \cap \delta(S)| \cdot \epsilon$.

According to the uniform increase rule, the dual variables of all active components in Γ are increased by the same amount in each iteration. We can therefore focus on a specific iteration, and investigate the impact of increasing the dual variables on our final primal solution. For a specific component S in Γ, y_S contributes to the edges in $F' \cap \delta(S)$. It follows that, in each iteration, the increase of the final primal solution is $\sum_{S\in\Gamma} \epsilon \cdot |F' \cap \delta(S)|$. Since the increase of the dual solution is $\sum_{S\in\Gamma:f(S)=1} \epsilon \cdot f(S)$, we need to prove that $\sum_{S\in\Gamma:f(S)=1} |F' \cap \delta(S)| \leq 2 \cdot \sum_{S\in\Gamma} f(S)$.

We define a hypergraph H for each iteration, where each component of Γ becomes a vertex of H, and for a component S in Γ, the edges of $F' \cap \delta(S)$ are also in H. So H can be obtained by contracting each component to a single vertex from F'. Because the components of Γ are connected and pairwise disjoint, H is a tree. In Figure 7.5 in the Appendix, the hypergraph H for each iteration can be obtained by contracting the red dashed circles. It is easy to see that the increase of our final primal solution in one iteration can be rewritten as $\sum_{v\in A} \epsilon \cdot d_v$, where A is the set of vertices of H which correspond to active components of Γ, and d_v is the degree of a vertex v in H. Since the dual solution increases $\epsilon \cdot |A|$ in this iteration, to establish the approximation ratio, all we need to show is that $\sum_{v\in A} d_v \leq 2|A|$.

The approximation ratio holds if H contains at most one inactive leaf. Intuitively the cost of active vertices with high degrees in H can be compensated by the active leaves, for they only have degree of 1. So now the problem is to prove that at most one leaf of H is inactive. Consider a connected component C in H. Assume the root of C and two leaves v_1 and v_2 of C are inactive. Assume v_1 is incident to edge e. Let C' be the resulting tree after removing e and v_1 from C. As each inactive component contains at least 3 vertices of G, C' must have at least 6 vertices and is therefore inactive. It follows that e will be removed in the deleting phase, which is contradictory to our assumption. Figure 4.2 shows that the ratio 2 is also tight, as it is possible for H to have one inactive leaf. The connected components in the current iteration are represented by large dashed circles in Figure 4.2.

Given the computed forest, we obtain a 4-approximation for the CCPBL with $k \geq 4$ by following the same strategy and analysis for the Exact Partitioning Problem in [40].

□

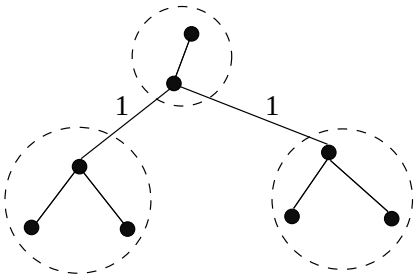

Figure 4.2: A tight example.

4.3 Approximation algorithms for the p-constrained Path/ Tree/Cycle Covering Problems

We have argued in the introduction section that the p-constrained Path/Tree/Cycle Covering Problems cannot be formulated by (IP). Therefore the GW-algorithm cannot be used directly to solve these problems. Recall that the p-constrained Path/Tree/Cycle Covering Problems are obtained by adding the cardinality constraint to network design problems with downwards monotone functions. In this section we generalize the heaviest-edge-first algorithm and show that a similar strategy yields a 2-approximation for the p-constrained Cycle/Tree Covering Problems, and a 4-approximation for the p-constrained Path Covering Problem.

4.3.1 Our proposed algorithm

The input to our algorithm is an undirected graph $G = (V, E)$ with a nonnegative edge cost $c(e)$ defined for every edge $e \in E$, and a constraint set C. The set C includes the constraints of a network design problem with downwards monotone functions, and also the constraint on the number of connected components in the optimal solution. The output of our algorithm is a forest F' which satisfies all the constraints in C, and the cost of F' is within twice the cost of the optimal solution. The algorithm has two stages, the same as in the GW-algorithm, the growing stage and the deleting stage. The pseudo code of the algorithm is listed in Figure 4.3.

This algorithm simply starts from a minimum spanning tree, examines all the edges in the minimum spanning tree in the non-increasing order of their costs, and an edge will be

Algorithm p**CCCP(**G**,** C**)**
Input: An undirected graph $G = (V, E)$, nonnegative edge costs, a constraint set C
Output: A forest F' on G, with a cost no more than twice the optimum

```
1  Comment: the growing stage starts from an MST
2  F ←any MST of G
3
4  Comment: the deleting stage
5  F' ← F
6  for i=1 to |V| − 1
7        e ←the edge with the ith largest cost in F
8        if F' − {e} still satisfies the constraints in C then
9              F' ← F' − {e}
10 endfor
```

Figure 4.3: The main algorithm for the p-constrained Path/Tree/Cycle Covering Problems.

removed if and only if its removal will not violate the constraints. The minimum spanning tree can be computed in time $O(|E|\alpha(|E|,|V|))$ [17] where $\alpha(|E|,|V|))$ is the inverse Ackermann function.

In the following we define P to be a network design problem with a downwards monotone function, and define P' to be the problem obtained by adding the cardinality constraint to P. P and P' are the default problems where our analysis is developed. We say P and P' are the path/tree/cycle version if the connected components in the optimal solutions are required to be paths/trees/cycles respectively. We define $F'(P)$ and $F'(P')$ to be the solutions produced by the algorithm in Figure 4.3 for P and P' (tree version) respectively. Similarly we define $F^*(P)$ and $F^*(P')$ to be the optimal solutions of P and P' respectively. We first present a combinatorial analysis for the performance guarantee of the above algorithm for P.

4.3.2 The structure of the solution $F'(P)$

We will now show that the above algorithm is a 2-approximation for the p-constrained Tree/Cycle Covering Problems and a 4-approximation for the p-constrained Path Covering Problem. We define the intersection $F'(P) \cap F^*(P)$ as follows. The intersection $F'(P) \cap F^*(P)$ includes only the edges that are present in both $F'(P)$ and $F^*(P)$. Note that each connected component of $F'(P) \cap F^*(P)$ is a minimum spanning tree of the vertices it spans.

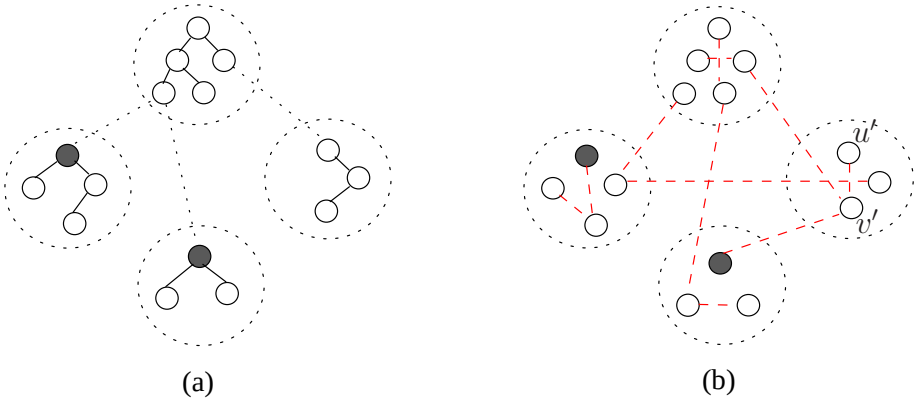

Figure 4.4: The structure of F'.

If we represent each connected component of $F'(P) \cap F^*(P)$ as a super node, we obtain a new graph $G' = (V', E' \cup E'')$ as shown in Figure 4.4(a). Each vertex in V' is a super

node corresponding to a connected component of $F'(P) \cap F^*(P)$. The solid edges are from $F'(P) - F^*(P)$, and they form the set E'. The dotted edges represent the edges of $F - F'(P)$, and form the set E''. Recall that F is the starting point of the deleting phase of our algorithm. In Figure 4.4(a) a connected component T' of $F'(P)$ is surrounded by a dotted circle. We define a set as active if $f(S) = 1$, and as inactive otherwise. Clearly T' cannot have two or more disjoint inactive subtrees. Otherwise, an edge on the path connecting two disjoint inactive subtrees can be removed without affecting the feasibility of $F'(P)$. Define a subtree T_u of T' as minimally inactive, if T_u is inactive and every subtree of T_u is active. Since there can be no two disjoint inactive subtrees in T', a minimally inactive subtree of T' (rooted at, say, r'), if there is any, can be easily computed. If T' is inactive, we can re-root T' at r'. Therefore T' has the property that each of its subtrees is active. This property implies that T' has at most one inactive super node, and if one such node exists, it must be the root. We represent inactive super nodes by solid small circles in Figure 4.4(a). In Figure 4.4(b), the vertices are the super nodes in V', and the dashed lines represent the edges in $F^*(P) - F'(P)$.

4.3.3 The performance guarantee of the solution $F'(P)$

According to the structure of $F'(P)$ in Figure 4.4, all super nodes of a connected component T' in $F'(P)$, except possibly the root, are active. Each active super node must be connected to another super node by an edge of $F^*(P)$ (see Figure 4.4(b)). Let $H^*(P)$ be the set that contain two copies of each edge of $F^*(P)$ which connects a pair of super nodes of V'. These edges are the dashed edges in Figure 4.4(b). Note that all the missing edges of $F^*(P)$ not present in $F'(P)$ are duplicated in $H^*(P)$. Our method of proving the approximation bound is to find a distinct edge e^* in $H^*(P)$ for each active super node u', except the root super node, such that the cost of e^* is no smaller than the cost of the edge connecting u' to its parent in G'. If this is true, then we can claim that the approximation ratio of the algorithm in Figure 4.3 is 2.

Here we set some notations which will be used throughout this paper. We define T' to be a connected component of $F'(P)$ which will be the component of study in our analysis. For a super node u', we denote the subtree rooted at u' by $T'_{u'}$. We use $(u', p(u'))$ to denote the edge from a super node u' to its parent node $p(u')$ in G'. We call a super node u' compensated by an unused edge e^* of $H^*(P)$ if $c(e^*) \geq c((u', p(u'))$. We define an edge of $H^*(P)$ to be of type-1 if its endpoints u' and v' are both in T'. We also define an edge of

$H^*(P)$ to be of type-2 if its two end points are in different connected components of $F'(P)$. An example illustrating type-1 and type-2 edges is given in Figure 4.5(a).

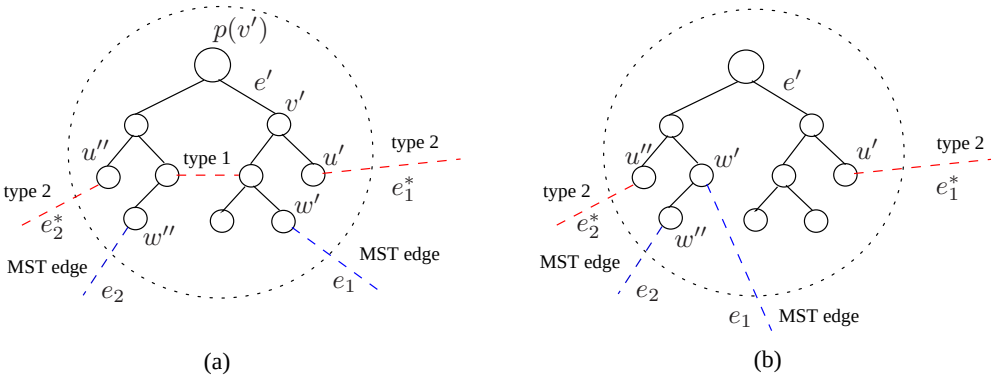

Figure 4.5: Two types of edges.

We first describe two lemmas characterizing some relationships of the edges in G' and H^*.

Lemma 4.3.1. *Let a type-2 edge e_1^* of H^* be incident on a super node u'. Let T_1 and T_2 be the two trees after removing an edge e' from T'. Assume T_2 contains u'. If T_1 is inactive, then $c(e_1^*) \geq c(e')$.*

Proof. Let $e' = (v', p(v'))$ (see Figure 4.5(a)). Assume that adding e_1^* to F creates a cycle. Let e_1 be an edge on the cycle that is deleted in the reverse deleting stage, and let e_1 be incident on w' in T'. If e' is on the path from u' to w', then we are finished since e' must be on the cycle, and according to the property of the minimum spanning tree, e_1^* has a cost no smaller than that of any edge on the cycle. Otherwise, consider the step just before e_1 to be deleted in the deleting phase. During that time e' and e_1 are both in F'. According to the property of downwards monotone functions, e' is also a candidate for deletion. However, the algorithm chooses to delete e_1, thus we have $c(e_1^*) \geq c(e_1) \geq c(e')$. Setting $e_1^* = e_1$ and $u' = w'$, the case when $e_1^* \in F - F'$ can be similarly argued. □

Lemma 4.3.2. *Let the set $S \subseteq E''$ contain the edges incident on some nodes of T' that were deleted from F in the deleting phase of our algorithm. For any two edges e_1 and e_2 in S, $max(c(e_1), c(e_2))$ is no smaller than the cost of any edge on the path that connects e_1 to e_2 in T'. Similarly, for two type-2 edges e_1^* and e_2^* of H^*, which are incident on u' and*

u'' of T' respectively, $max(c(e_1^*), c(e_2^*))$ is no smaller than the cost of any edge on the path which connects u' to u'' in T'.

Proof. Let e' be an edge in T' on the path connecting e_1 to e_2 (Figure 4.5). Without loss of generality, assume that $c(e_1) \leq c(e_2)$. Consider the step when e_2 is being considered for deletion from F. It is obvious that both e' and e_1 are present in F, thus deleting e' will also give us a feasible solution. In other words, e' is also a candidate edge for deletion. Since the algorithm deletes e_2 instead of e', according to our deleting phase, we must have $c(e_2) \geq c(e')$.

We now prove the second claim. Assume that e_1^* and e_2^* are not in F. Adding e_1^* and e_2^* will create two cycles to F. Consider the following cases:

Case 1: The two cycles created by adding e_1^* and e_2^* are disjoint (Figure 4.5(a)). Let them pass through two super nodes w' and w'' in T' respectively, and let w' and w'' be incident to two edges e_1 and e_2 in E'' respectively. In this case it is easy to see that e' must either be on the path from w' to w'', or on one of the cycles. The claim holds as we have shown above that $c(e') \leq max(c(e_1), c(e_2))$ if e' lies on the path from w' to w''.

Case 2: The two cycles pass through some common vertex w' in T' (Figure 4.5(b)). In this case it is easy to verify that e' must belong to one of the two cycles.

Therefore the claim holds if e_1^* and e_2^* are not in F. The other cases can be similarly argued. □

If we consider the edges of $H^*(P)$ incident on the super nodes of T' only, then we may get several connected subgraphs of $F^*(P)$ (see Figure 4.4(b)). Let T_j^* be one of them. Note that T_j^* is a connected subgraph of $F^*(P)$. Define a super node u' as extreme with respect to T_j^* (or a set S of super nodes) if no ancestors of u' in T' exist in T_j^* (or S); u' is non-extreme otherwise. An example is given in Figure 4.6(b). In Figure 4.6(b) both u' and v' are extreme super nodes, but w' is non-extreme. The following lemma shows that all super nodes, except one extreme super node of T_j^*, can be compensated by edges of T_j^* (i.e., edges of $H^*(P)$). Recall that a super node u' is compensated if an edge e^* of $H^*(P)$ is uniquely associated with u', and $c(e^*) \geq c((u', p(u'))$.

Lemma 4.3.3. *Let T_j^* be a connected subgraph of F^* where only the super nodes of T' are involved. Only one (arbitrary) extreme super node of T_j^* cannot be compensated by the edges of T_j^*.*

Proof. We prove this lemma by induction on the number of super nodes. The lemma is trivially true when T_j^* contains only one super node. Assume the lemma holds for all T_j^* with less than m super nodes. Given a T_j^* with m super nodes, we have two cases (Figure 4.6):

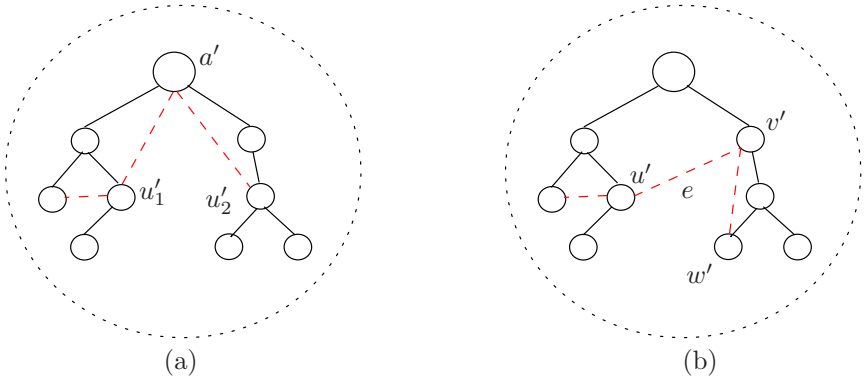

Figure 4.6: Only one extreme super node cannot be compensated by edges in $F^*(P)$. The dashed edges (elements of $H^*(P)$) are the only edges of T_j^*.

Case 1: There exists only one extreme super node a' in T_j^* (Figure 4.6(a)). We delete a' and all the edges of T_j^* incident on a'. Let $T_{j_1}^*$, $T_{j_2}^*$, $\cdots$, $T_{j_t}^*$ be the resulting connected components of T_j^* and u'_1, u'_2, $\cdots$, u'_t be the extreme super nodes of such connected components (with respect to each of them). We need to show that only a' is not compensated and all $u'_l, l = 1, 2, \cdots, t$ are compensated. According to our assumption, all super nodes except u'_1, u'_2, $\cdots$, u'_t can be compensated by the edges in $T_{j_1}^*$, $T_{j_2}^*$, $\cdots$, $T_{j_t}^*$. Since T_j^* is connected, each new connected component must have a different edge to a'. For a connected component $T_{j_l}^*$ $(1 \leq l \leq t)$, let $e'_l = (a', v'_l)$ be such an edge. As u'_l is an arbitrary extreme super node of $T_{j_l}^*$, we can assume that u'_l is v'_l or an ancestor of v'_l. Adding e'_l to $F'(P)$ will create a cycle that includes the edge $(u'_l, p(u'_l))$, therefore u'_l can be compensated by e'_l.

Case 2: There exist at least two extreme super nodes, say u' and v', in T_j^*. The two nodes u' and v' are connected in T_j^* (Figure 4.6(b)). Without loss of generality, we can assume that on the path $u' \sim v'$ there exists an edge e^* connecting u' or a descendant of u' to v' or to a descendant of v'. Therefore adding e^* to F creates a cycle that includes both $(u', p(u'))$ and $(v', p(v'))$. Deleting e^* from T_j^*, we get two connected components $T_{u'}^*$ and $T_{v'}^*$ each with fewer than m super nodes. According to our assumption, all super

nodes of T_j^* except u' and v' can be compensated by edges of $T_{u'}^*$ and $T_{v'}^*$. Furthermore, $c(e^*) \geq c(u', p(u'))$ and $c(e^*) \geq c(v', p(v'))$. So in this case, all super nodes of T_j^* except u' or v' can be compensated by edges of T_j^*.

□

Let $S_C = \{T_1^*, T_2^*, \cdots, T_l^*\}$ be any subset of the set of connected subgraphs of $F^*(P)$ where only the super nodes of T' are involved. We say a super node a in a component of S_C is an extreme super node of S_C, or a is extreme with respect to S_C, if no super node in the components of S_C is an ancestor of a in T'. Suppose that for each T_i^* $(1 \leq i \leq l)$ there exists a type-2 edge e_i^* attached to it. Let $E_C^* = \{e_1^*, e_2^*, \cdots, e_l^*\}$. The following lemma can be viewed as a generalization of Lemma 4.3.3.

Lemma 4.3.4. *Only one (arbitrary) extreme super node of S_C cannot be compensated by the edges in E_C^* and the edges in the components of S_C. Moreover, at least one edge of E_C^* is not used to compensate any super node involved in S_C.*

Proof. We prove this lemma by induction on the number of connected components in S_C. Due to Lemma 4.3.3, it is trivially true if S_C only contains one connected component. Assume it is true for all such sets with fewer than l connected components. Consider a set S_C consisting of exactly l connected components. Without loss of generality, assume T_1^* in S_C has an arbitrary super node a_1' which is extreme with respect to S_C. Let $S_C' = \{T_2^*, \cdots, T_l^*\}$. According to our assumption, there is only one arbitrary extreme super node a_2' of S_C' that cannot be compensated, and one edge, say e_2^* of E_C^*, unused. Let e_1^* and e_2^* be incident on u_1' and u_2', where u_1' is in T_1^* and u_2' is in T_2^*. Without loss of generality, assume that $c(e_1^*) \geq c(e_2^*)$. Consider the following cases:

Case 1: T_1^* is interleaving with a component T_j^* in S_C', in the sense that the path segment of T' connecting two end points of a type-1 edge e^* of T_1^* contains an extreme super node a_j' in T_j^* (Figure 4.7(a)). Without loss of generality we assume that a_j' is extreme with respect to S_C'. Since a_2' is assumed to be an arbitrary extreme super node, we can assume that $T_2^* = T_j^*$ and $a_2' = a_j'$. Therefore a_2' can be compensated by e^*, and for the components in S_C, only a_1' has not been compensated, and we have two type-2 edges e_1^* and e_2^* unused.

Case 2: T_1^* is not interleaving with any components in S_C'. As a_1' is extreme with respect to S_C, a_2' is either a descendant of a_1' or in a different branch from a_1'. In both cases the edge $(a_2', p(a_2'))$ must be on the path from e_1^* to e_2^*. By Lemma 4.3.2, the cost of

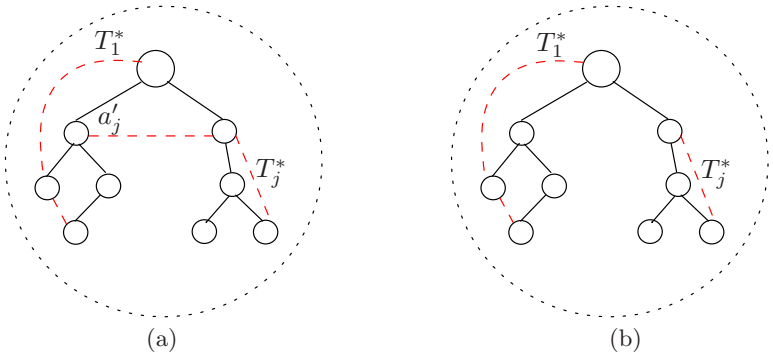

Figure 4.7: T_1^* is interleaving with T_j^* in (a), but not in (b).

$(a'_2, p(a'_2))$ is less than that of e_1^*. Therefore we can use e_2^* to compensate $(a'_2, p(a'_2))$, and for the components in S_C, only a'_1 has not been compensated, and we have a type-2 edge e_2^* unused. □

We are now ready to prove the performance guarantee of the algorithm in Figure 4.3 for network design problems with downwards monotone functions. The proof below is for the tree version. The cycle and path versions will be discussed later.

Theorem 4.3.5. *The cost of the forest F' found by the algorithm in Figure 4.3 is bounded by twice the cost of the optimal solution F^* of P (tree version).*

Proof. Let T_0^* be an arbitrary connected component of F^*. Assume the super nodes of T_0^* are all from a connected component T' of F'. Then as shown in Lemma 4.3.3, only one extreme super node a' cannot be compensated by the edges of T_0^*. If T_0^* has at least two extreme super nodes, then according to our proof of Case 2 for Lemma 4.3.3, we can pick an additional copy of an edge e^* of T_0^* to compensate a'. According to our way of rooting T', if T_0^* has only one extreme super node a', then a' must be the root of T'. In this case a' needs not to be compensated. Thus if T_0^* is contained entirely in T', then all the super nodes in T_0^* can be compensated by the edges in T_0^*.

Let $S_C = \{T_1^*, T_2^*, \cdots, T_l^*\}$ be a set of connected components of F^*, and $E_C^* = \{e_1^*, e_2^*, \cdots, e_l^*\}$ be a set of edges of F^*, where the super nodes of all the connected component of S_C are contained entirely in T'. And each edge e_i^*, $1 \leq i \leq l$, of E_C^* is of type-2, namely it has one end in T_i^* and the other end not in T'. According to Lemma 4.3.4, only one (arbitrary)

extreme super node a' (defined with respect to all the super nodes in the components of S_C) can not be compensated by the edges of these components, and one edge of E_C^* is unused. The theorem then holds if all the super nodes in T' are in the connected components of S_C. Therefore we assume that there exists a connected component T_0^* whose super nodes are all from T'. The theorem also holds if T_0^* does not contain the root r' of T', as all the super nodes in T_0^* can be compensated by the edges in T_0^*. In the following we assume that T_0^* contains r'.

Case 1: T_0^* is interleaving with a connected component of S_C. By an argument similar to Case 1 in the proof for Lemma 4.3.4, we can show that only one extreme super node can not be compensated by edges in F^*, and there is an edge of E_C^* remaining unused. The theorem then holds as r' is the only extreme super node with respect to all the super nodes in T' and r' needs not to be compensated.

Case 2: T_0^* is not interleaving with any connected components of S_C. Due to the intersection $F' \cap F^*$, in this case r' must be inactive and also the sole super node in T_0^*. The theorem then holds according to Lemma 4.3.2.

Thus for each super node u' in G', an edge in F^* can be found to have a larger cost than an edge in F' connecting u' to its parent in G'. In all cases, each edge in F^* is used at most twice, so by doubling the missing edges in F^*, clearly we have that $cost(F') \leq 2 * cost(F^*)$. □

The following theorem states that the approximation ratio is still 2 when P is of cycle version.

Theorem 4.3.6. *The cost of the forest F' found by the algorithm in Figure 4.3 is at most the cost of the optimal solution F_c^* of P (cycle version).*

Proof. For a cycle in the optimal solution, the number of edges and the number of super nodes in the cycle are the same. For each super node u', we try to find a distinct edge e^* in the optimal solution F_c^* to compensate u'. Consider a connected subgraph C^* of F_c^* whose super nodes are all from a connected component T' of F'; we have the following cases:

Case 1: C^* is a cycle. When P is of cycle version, a super node u' may have a self-loop edge e^* in F_c^*. For the k-Cycle Covering Problem, this happens when u' corresponds to a path with at least k vertices of G. In this case u' together with e^* represents a cycle in F_c^*. As u' is inactive, it must be the root r', but r' needs not to be compensated. Therefore in the following we assume that C^* does not contain any self-loop edge.

If C^* does not contain the root, then C^* must have two extreme super nodes u' and v' which are in different branches of r', as T' is rooted in such a way that each branch of T' is active. u' is connected to v' in C^*, so without loss of generality we can assume that there exists an edge e^* in C^* with one end in $T'_{u'}$ and the other end in $T'_{v'}$. If we remove e^* from C^*, then C^* becomes a path. According to Lemma 4.3.3, only one extreme super node, say u', cannot be compensated by the edges on the path. The theorem then holds, as e^* can be used to compensate u'.

Case 2: C^* is a path. In this case, we consider all the paths involving the super nodes of T'. Let $S_C = \{P_1^*, P_2^*, \cdots, P_l^*\}$ be the set of all such paths, and let $E_C^* = \{e_1^*, e_2^*, \cdots, e_l^*\}$ be a set of type-2 edges of F_c^*, where each edge e_i^* of E_C^*, $1 \leq i \leq l$, is attached to P_i^*. Note that for each path P_i^*, where $1 \leq i \leq l$, e_i^* is picked arbitrarily from the two type-2 edges connecting P_i^* to outside T'. Applying Lemma 4.3.4 on S and E_C, we have that only one of the involved extreme super nodes, say a', cannot be compensated, and we still have one edge e^* of E_C^* unused.

If the root is in one of the paths, then we are finished, since the root needs not to be compensated. Otherwise, the root is in a cycle C_r^* contained entirely in T'. The rest of the proof follows in the same way as that in the proof of Case 1 and Case 2 for Theorem 4.3.5.

Therefore we have shown that the cost of F' is at most the cost of F_c^*. Since we double the edges in F' to get our final cycle solution, the approximation ratio of our algorithm for P (cycle version) is still 2. □

Theorem 4.3.5 also holds when P is of path version. By doubling the edges in $F^*(P)$ we get a cycle cover C of the underlying graph G. A path cover can be obtained after removing an arbitrary edge from each cycle in C. Therefore we have the following theorem:

Theorem 4.3.7. *The algorithm in Figure 4.3 is a 4-approximation for P (path version).*

4.3.4 The performance guarantee for the p-constrained Path/Tree/Cycle Covering Problems

We claim that the algorithm in Figure 4.3 is a 2-approximation for P', a p-constrained Tree/Cycle Covering Problem. The algorithm in Figure 4.3 for P' will stop deleting edges from the minimum spanning tree if the number of components in the remaining forest involving the vertices in G is already p.

Assume c connected components exist in $F'(P)$. It is easy to see that $F'(P')$ can be obtained from $F'(P)$, by greedily adding $c-p$ minimum spanning tree edges between the connected components of $F'(P)$. Let these $c-p$ edges form the set S. The edges in S represent the optimal way to reduce the number of connected components in $F'(P)$ to p. However, to prove the performance guarantee for P', we need to locate $c-p$ edges in $F^*(P')$ which have a total cost no smaller than that of the edges in S. Also note that these $c-p$ edges in $F^*(P')$ can be used at most once for compensating the super nodes of G'.

Our analysis utilizes the fact that in the proof of the performance guarantee for P (tree/cycle version), if the root r' of a connected component T' of F' is active, then there must exist a type-2 edge e^* incident on a super node of T' that is not used to compensate any super node in T'. In order to prove the ratio for $F'(P')$, our analysis starts from the solution $F'(P)$, and focuses on locating a set S^* of $c-p$ edges in $F^*(P')$ between the connected components of $F'(P)$. In the next theorem, we locate such a set S^* and show that the edges in S^* are guaranteed to be used at most once to compensate some super nodes of G'. We give the formal proof in Theorem 4.3.8.

Theorem 4.3.8. *The cost of the forest $F'(P')$ found by the algorithm in Figure 4.3 for P' (a p-constrained Tree Covering Problem), is bounded by twice the optimum.*

Proof. First note that all the above analysis for problem P (tree/cycle version) can also be applied for $F^*(P')$, the optimal solution for P'. This is due to the fact that the constraints for P are also satisfied in $F^*(P')$. In the following we show the proof for P'.

Let T' be a connected component of $F'(P)$. If the root r' of T' is active, then as in the proof of Theorem 4.3.5, there must exist an unused type-2 edge e_1^* (with one end in T') in $H^*(P')$. Assume r' is inactive. Consider the case when there exists an edge e^* of $H^*(P')$ incident on r'. If e^* is of type-2, then a copy of such an edge has not been used to compensate any super node in T' in the analysis for $F'(P)$ (tree/cycle version). Otherwise e^* connects r' to another super node u' also in T'. But this corresponds to Case 1 in the proof of Theorem 4.3.5, where we can also get one unused type-2 edge e_1^* in $H^*(P')$ incident on a super node of T'.

Therefore whenever r' is incident to an edge e^* in $F^*(P')$, a type-2 edge of $H^*(P')$ can be used to connect T' to another connected component in $F'(P)$. Let S_1 denote the set of roots of the connected components of $F'(P)$ which are not incident to any edge in $F^*(P')$. Let S_2 denote the set of roots of the connected components of $F'(P)$ which are incident to

some edges in $F^*(P')$. Clearly in $F^*(P')$, the roots of S_2 are in at most $p - |S_1|$ different connected components. The roots of S_2 are in exactly $c - |S_1|$ connected components in $F'(P)$. Therefore there must exist at least $c - |S_1| - (p - |S_1|) = c - p$ type-2 edges in $F^*(P')$, to reduce the number of connected components containing the roots of $F'(P)$ to at most p. We show in the following that at least $c - p$ roots of S_2 can be allocated distinct type-2 edges in $F^*(P')$. These edges will form the aforementioned set S^*.

We define $V'(C)$ to be the set of super nodes in a component C of G'. Consider a connected component C of $F^*(P')$ that involves m roots of S_2. We claim that $m - 1$ or $m-2$ roots in $V'(C)$, can be assigned distinct type-2 edges in $F^*(P')$, that are incident only on the super nodes in $V'(C)$ and only used at most once to compensate some super nodes in G'. In the latter case, there must also exist two type-2 edges e_1^* and e_2^* appearing in the cut of $V'(C)$ to $V' - V'(C)$ in $F^*(P')$, that are only used once to compensate some super nodes in G'.

In the following we assume that C contains only two active roots r_1 and r_2 of S_2, which are connected directly by an edge e^* in $F^*(P')$. The general cases can be argued similarly. Assume r_1 and r_2 are in two different connected components C_1 and C_2 of $F'(P)$ respectively. According to our analysis, we have two unused type-2 edges e_1^* and e_2^*, from the cut of $V'(C_1)$ to $V' - V'(C_1)$ and the cut of $V'(C_2)$ to $V' - V'(C_2)$ in $H^*(P')$ respectively. The claim then holds, no matter e_1^* or e_2^* or both are incident on super nodes only in C_1 and C_2, or also on some super nodes not in C_1 and C_2. This completes the proof, as for each connected component C of $F^*(P')$ involving m roots of S_2, $m-1$ roots can be assigned distinct type-2 edges of $F^*(P')$, possibly after selecting one of two type-2 edges of $F^*(P')$. □

Similarly we can establish the following theorem.

Theorem 4.3.9. *The algorithm in Figure 4.3 is a 2-approximation for the p-constrained Cycle Covering Problem and a 4-approximation for the p-constrained Path Covering Problem.*

Chapter 5

Multi-Depot Capacitated Vehicle Routing

5.1 Introduction

In this chapter, we study approximation algorithms for two variants of the Vehicle Routing Problem involving multi-vehicles and multi-depots. More specifically, we investigate approximation algorithms for a model of the Multi-Depot Capacitated Vehicle Routing Problem (called MDCVRP in this thesis) [11], and a variant of the Vehicle Routing Problem with Time Windows (called MVSP in [52]).

5.1.1 Multi-Depot Capacitated Vehicle Routing Problem

In the MDCVRP we are given an undirected complete graph $G = (V \cup D, E)$, where V and D denote a set of customers and a set of depots respectively, and E denotes the set of weighted edges satisfying the triangle inequality. A vehicle with capacity k is located at each depot node and can be used to serve at most k customers. It is assumed here that the number of customer nodes $|V|$ is no more than $|D| \times k$. The MDCVRP is to find a minimum cost set of tours covering all the customer nodes of V such that each tour contains at most k customers, and a distinct depot.

A problem closely related to the MDCVRP, called the *Vehicle Dispatching Problem* (VDP), is studied by Krumke et al. in [60]. The VDP is defined similarly to the MDCVRP, with the only difference being that each vehicle will not return to its home base (depot).

In [60] the authors gave a $(2k-1)$-approximation for the VDP in general graphs, and a 2-approximation for a special case of the VDP when the capacity k is equal to the number of customers.

In the following we introduce some other approximation results related to the MDCVRP. The studies in [3, 10, 15, 16] consider the case in which a single vehicle is available to service the customers. Some other papers, e.g. [13, 49], work on the case when multiple vehicles can be used. However, it is assumed in these papers that all the vehicles must share a same depot. The existence of a central depot for all the vehicles is also the typical context for the VRP. Therefore MDCVRP can be viewed as a further generalization of the CVRP.

Not many approximation results are known for the Vehicle Routing Problem with Multi-Depots. To the best of our knowledge, the only known approximation results for problems in this category are by Simchi et al. in [67], and Chekuri et al. in [18]. In [67], Simchi et al. considered the MDCVRP in the following setting (called MCVRP in [67]): the underlying graph G is defined similarly to that in the MDCVRP. The vehicle also has a capacity of k; however, in the MCVRP when a vehicle returns to its depot after servicing some customers, the same vehicle can start another round of servicing immediately. In the MCVRP there are no constraints on the number of servicing rounds where a vehicle can participate. Therefore despite the existence of multi-depots and multi-vehicles, in the optimal solution of the MCVRP, it is possible that only one vehicle is used to service all the customers. This setting is also deployed in [18].

The main difference between the MDCVRP and the MCVRP is that, the MDCVRP has a "hard" capacity constraint for each depot. In fact the MDCVRP is a further generalization of the MCVRP. We transform an instance $G = (V \cup D, E)$ of the MCVRP to an instance $G' = (V \cup D', E \cup E')$ of the MDCVRP as follows. For each depot d of D, we consider copies of d denoted by $d_1, d_2, \cdots, d_{|V|}$). We create new zero cost edges $e' = (d_i, d_j)$ for any $1 \leq i < j \leq |V|$. All the new depots and new edges are added to D' and E' respectively. It is not difficult to verify that the optimal MDCVRP solution for G', is also the optimal MCVRP solution for G. The MCVRP is therefore a special case of the MDCVRP, and any algorithm that works for the MDCVRP can also be used for the MCVRP.

5.1.2 Multi-Vehicle Scheduling Problem

In the MVSP we are given an undirected complete graph $G = (V, E)$, where each vertex u_i of V is associated with a job j_u, and each edge e has a non-negative weight c_e. The edge

costs in G satisfy the triangle inequality. There are also m identical vehicles available to service the jobs. Each job j_u has its own release time $r(u)$ and handling time $h(u)$. A job j_u can only be serviced after its release time $r(u)$, and the handling time $h(u)$ represents the time needed to finish processing j_u. The objective is to find a schedule in which the maximum completion time of the jobs, i.e. the makespan, is minimized.

The current research status on the MVSP is as follows. Nagamochi et al. introduced the MVSP and gave a 2-approximation algorithm for the MVSP on paths in [52]. For the MVSP in trees, Nagamochi et al. in [51] and Augustine et al. in [5] separately gave two polynomial time approximation schemes (PTAS). In [52] it is assumed that the number of vehicles and the number of leaves of the underlying tree are some fixed constant. In [5] it is assumed that the number of distinct release times and the number of leaves of the underlying tree are constant. There are no constant factor approximation algorithms in the literature for the MVSP in trees. For general graphs, no approximation results are known.

5.1.3 Our results and solution techniques

We summarize our results as follows.

1. We design a 2-approximation algorithm for the MDCVRP in trees (Section 5.3). Using the results in [32], we obtain an $O(\log n)$-approximation for the MDCVRP in general graphs. However, when the underlying graph is a path, the MDCVRP is optimally solvable (Section 5.2).

   Our algorithm can be modified to get a 3-approximation for the VDP in trees [60], which results in a $min\{2k-1, O(\log n)\}$-approximation for the VDP in general graphs.

2. We give a 6-approximation algorithm for the MDCVRP in graphs with bounded tree-width.

3. For the MDCVRP in general graphs, we give another k-approximation algorithm, and thereby obtain a $min\{k, O(\log n)\}$-approximation for the MDCVRP in general graphs (Section 5.6).

4. For the MVSP in trees, we design a 3-approximation algorithm and therefore obtain $O(\log n)$-approximation for the MVSP in general graphs. No constant factor approximation algorithms are previously known for the MVSP in trees.

Almost all of our algorithms in this chapter are based on a different way of applying dynamic programming for approximation algorithms on the VRP in trees. The main idea of our approximation algorithms is to use dynamic programming to indirectly decompose the original problem P into a set of disjoint subproblems. Approximating these subproblems separately gives us the solution for P with the desired approximation ratio.

Dynamic programming is one of the most fundamental and powerful tools for designing efficient algorithms. However, its application in approximation algorithms for the VRP is relatively new. The existing way of using dynamic programming on approximation algorithms for the VRP, e.g. in [7, 52, 59], works as follows. Firstly a set of disjoint NP-hard subproblems is defined for the original problem P, and an approximation algorithm A with ratio α is designed for these subproblems. Typically these subproblems have good properties, and algorithm A can approximate them well. In this approach, dynamic programming is used as a master algorithm to locate a set of subproblems with cost bounded by $\beta \cdot OPT_P$, where OPT_P denotes the optimal solution of P. During this process all the configurations of the subproblems are tried, and algorithm A is applied to each of the configurations. The one with the smallest cost is chosen to be the final solution. It is easy to see that this solution is an $(\alpha\beta)$-approximation for the original problem P.

The above approach, used in [7, 52] relies on the fact that all the configurations of the subproblems of P can be examined in polynomial time by dynamic programming. Therefore it is only applicable when the underlying graph is a path (as in our algorithm for the MDCVRP on paths), or when some ordering can be found for the underlying graph (as in the $O(\log^2 n)$-approximation algorithm for the VRP with time windows [7], where the dynamic programming proceeds based on an ordering of the vertices). However, for the VRP in trees, we do not know how to deploy this method, since for a vertex u, the number of possible configurations of the subproblems containing u is exponential in the number of children of u.

We apply the dynamic programming technique to obtain constant factor approximation algorithms for the MDCVRP and the MVSP in trees in the following way. Our algorithms consist of two steps. In the first step, we use dynamic programming to decompose the original problem P into a set of disjoint subproblems. However, as it is not possible to try all the configurations of the subproblems by directly obtaining solutions for P, we instead find a relaxation problem P' of P and locate a set S of disjoint subproblems by using dynamic programming in the underlying graph. In the second step we work on approximation

algorithms for the subproblems in S. We show that a good approximation for P can be obtained by approximating these subproblems well.

Solving relaxation problems is a general principle when designing approximation algorithms. Finding appropriate relaxation problems, or equivalently computing good lower bounds for the original problems, is crucial to the performance guarantees of these algorithms. Our algorithms give two concrete ways to locate the relaxation problems for the VRP in trees.

We define an edge e to be a gap if in the optimal solution no vehicle passes through e to service customers. Define a gapless subproblem to be a subproblem whose underlying subgraph contains no gaps. In the algorithm for the MDCVRP in trees, we choose P' in a way such that P' satisfies some property inherent in the gapless subproblems of P. P' is much simpler and can be solved well by using dynamic programming in the underlying tree. More importantly, as P' captures the major properties or structures of P, after solving P' we not only find a lower bound on OPT_P, but also obtain a set of subproblems with good properties which can be used to approximate P well.

The two steps in our algorithms are highly connected. For the MVSP in trees, the problem P' solved in the first step in fact comes from the approximation algorithm used in the second step. We locate P' for the MVSP in the following way. For a gapless subproblem, we design an approximation algorithm A which produces a solution with cost function C. We treat C as a lower bound, and define P' to be the problem of locating a set S of subproblems with the smallest possible bound C. It is guaranteed that when applying algorithm A on the subproblems in S, the solution will be upper bounded by the smallest lower bound C.

The major difference of our algorithm from the existing algorithms [7, 52] is as follows. In the existing method, the algorithm A is explicitly executed for each configuration tried by dynamic programming. However evaluating each configuration by A in trees cannot be done in polynomial time by using dynamic programming. We instead focus on the effect of evaluating each gapless subproblem by A. We first derive a theoretical lower bound LB for the solution produced by algorithm A. P' is then defined to be locating a set S of subproblems with the minimum possible bound LB. P' therefore has the property that the cost of the solution obtained after running algorithm A on the subproblems in S, is no more than that of the solution after applying A on the gapless subproblems. If we assume that A is an α-approximation for the gapless subproblems, then solving the located subproblems by A will yield a solution with cost bounded by $\alpha \cdot OPT_P$. Therefore our algorithm in

fact indirectly turns an α-approximation algorithm for the gapless subproblems into an α-approximation for the original MVSP instance P. We will give more details of the algorithm in Section 5.7 of this chapter.

5.2 An optimal algorithm for the MDCVRP on paths

Figure 5.1 shows the forbidden subgraphs in the optimal solution of an instance of the MDCVRP on paths.

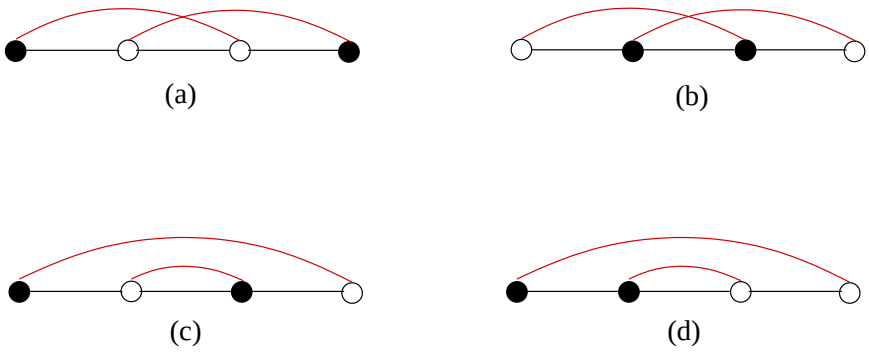

Figure 5.1: Forbidden subgraphs in the optimal solution for the MDCVRP on a path. Depots and customers are represented by circles with black and white fills respectively. An arc connecting two different dots in a solution indicates that the customer is assigned to the depot in that solution.

We have the following lemma:

Lemma 5.2.1. *The subgraphs shown in Figure 5.1 are forbidden in the optimal solution of the MDCVRP on a path.*

Proof. Let the left and right depots in Figure 5.1 be d_1 and d_2 respectively, where d_1 lies to the left of d_2. Let the customers assigned to the two depots be $v_1, v_2, v_3, \cdots, v_t$ (ordered from left to right according to their locations on the path) where $1 \leq t \leq 2k$. We assign $v_1, v_2, \cdots, v_k$ to d_1, and the remaining customers to d_2. It is not difficult to verify that the new solution would not have a larger cost than that of the original solution.

□

As a consequence of this lemma, the only allowed subgraphs in the optimal solution are:

Figure 5.2: Allowed subgraphs in the optimal solution for the MDCVRP on a path.

Define an edge e to be a gap if in the optimal solution no vehicle crosses e to serve customers. Define a gapless subproblem to be a subproblem whose underlying subgraph contains no gaps. We have the following lemma for a gapless subproblem of the MDCVRP on paths.

Lemma 5.2.2. *In the optimal solution of a gapless subproblem, at most one vehicle u can be assigned fewer than k customers. All the customers assigned to a depot other than u must lie on one side of the depot.*

Proof. In Figure 5.2(e), let the left and the right depots be d_1 and d_2, and the left and the right customers be v_1 and v_2 respectively. If d_2 is assigned fewer than k customers, then it is easy to see that assigning v_1 and v_2 to d_2 has a smaller cost than that of the original solution. Therefore either the current solution is not optimal or not gapless. This contradicts the assumption. □

Given a gapless subproblem, we can figure out the optimal schedule as follows. We scan the path segment from left to right, and put the encountered customers in a queue. When a depot d is met, we remove k customers from the queue and assign them to d, if the queue has k or more customers. If the number of customers in the queue is less than k, we assign the customers to d and turn to scan the path segment from right to left. The process is terminated when d is visited again. Depot d gets the remaining customers in the queue. Note that due to the gapless property, only d may be assigned $< k$ customers.

In general cases the optimal solution might contain gaps. However, we can try all the configurations of the gaps by using dynamic programming. The method is similar to that in [52]. Let l be the leftmost vertex of the path. In the following we assume that a and b are two vertices on the path and a is to the left of b. Denote $P(a, b)$ to be the path segment from a to b. We define $L^*(P(a, b))$ to be the optimal cost of the subproblem defined on $P(a, b)$ (assume this subproblem is gapless). Let $e = (u, v)$ be an edge on the path where u

is to the left of v. We define $L(e)$ to be the cost of the optimal solution of the subproblem defined on $P(l, u)$. We can then write the recursion as:

$$L(e) = min_{e'}\{L(e') + L^*(P(v', u))\}$$

where the edge $e = (u, v)$ lies to the right of $e' = (u', v')$ and u' is to the left of v'.

In this recursion, we proceed from left to right on the path. When an edge $e = (u, v)$ is met, we try to form the last gapless subproblem by locating another edge $e' = (u', v')$ which is to the left of e. We assume the subproblem on $P(v', u)$ is gapless and apply the above algorithm to this subproblem. It is not difficult to see that the time complexity of this algorithm is $O(|V|^3)$.

5.3 2-Approximation algorithm for the MDCVRP in trees

In the sequel we assume that all the customers and depots are located at the leaves of the tree. The input tree can be transformed by creating a new node u' for each depot/customer u and adding a zero cost edge between u' and u.

The following notations are used in describing the algorithm. For a vertex u, we denote T_u to be the subtree rooted at u, and $p(u)$ to be the parent of u. $p(u)$ is null if u is the root. We associate with each subtree T_u a number $f(T_u)$ which equals k times the number of depots minus the number of customers in T_u. We call a subtree T_u positive (negative) if $f(T_u)$ is positive (negative). Similarly, an edge $e = (p(u), u)$ or a vertex u is positive (negative) if the subtree T_u is positive (negative). Finally we denote $c(e)$ to be the weight of an edge e, and denote OPT_P to be the optimal cost of solving a problem P.

Consider an instance of the MDCVRP where the underlying tree is of height 2 and the number of customers is exactly k times the number of depots. We assume that the leaves are all located at the second level of the tree (see Figure 5.3). In the figure a triangle represents a subtree of height one, and inside a triangle, say, for subtree T_u, we show the number $f(T_u)$.

It is NP-hard to find the number of times each edge incident on r is traversed in the optimal solution. The proof is similar to that in the appendix of [81] for the k-delivery TSP in trees. The proof reduces the 3-partition problem to the MDCVRP.

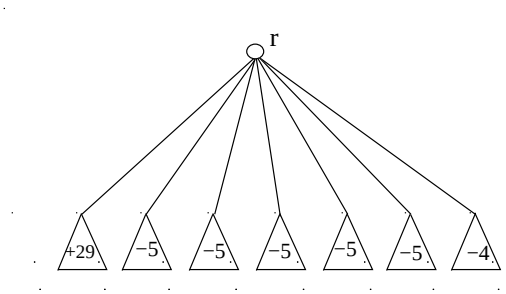

Figure 5.3: An instance of the MDCVRP on a tree of height 2. $k = 8$.

5.3.1 A new flow bound for the MDCVRP in trees

Assume that in an instance of the MDCVRP in trees, the number of customers is exactly k times the number of depots. It is easy to see that $FB_e = 2\lceil |\frac{f(T_u)}{k}| \rceil$ is a lower bound on the number of times the edge $e = (p(u), u)$ needs to be traversed in the optimal solution. This bound is called the *flow* bound. Similar flow bounds were used for the k-delivery TSP [15] and the Dial-a-Ride problem [16].

One difficulty of designing approximation algorithms for the MDCVRP in trees is that, the above *flow* bound is not meaningful for designing approximation algorithms for the MDCVRP in trees when the number of customers is less than k times the number of depots. We give an example in Figure 5.4.

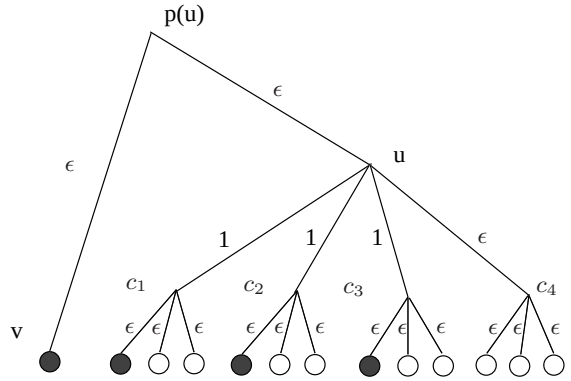

Figure 5.4: An instance of the MDCVRP. k=3.

In Figure 5.4, depots are represented by circles with black fills and customers are represented by circles with white fills. The edges (u, c_1), (u, c_2) and (u, c_3) have cost 1, and other edges have a cost of ϵ, for arbitrarily small $\epsilon > 0$. Firstly, it is easy to see that $FB_{e'} = 1$ for the positive edge $e' = (u, c_1)$, but e' is not traversed in the optimal solution. Secondly, according to the *flow* bound, the edge $e = (p(u), u)$ in Figure 5.4 should never be traversed, as $f(T_u) = 0$. However, in the optimal solution we need the vehicle at depot v to service the customers in T_{c_4}. Following the same structure, it is possible to create instances where an edge $e' = (p(u'), u')$ (with $f(T_{u'})$=0) has to be traversed an unbounded number of times in the optimal solution.

Another difficulty of designing approximation algorithms for the MDCVRP in trees, if we want to use the existing dynamic programming method for the VRP as discussed in Section 5.2, is to define the disjoint subproblems. We can define gapless subproblems similarly for the MDCVRP on paths (recall that an edge e is a gap if in the optimal solution no vehicle passes through e to service customers). However, as we discussed above, it is not possible to try all the configurations of the gapless subproblems in trees. Moreover, the optimal solution for a gapless subproblem can be quite complicated. We give an example in Figure 5.5.

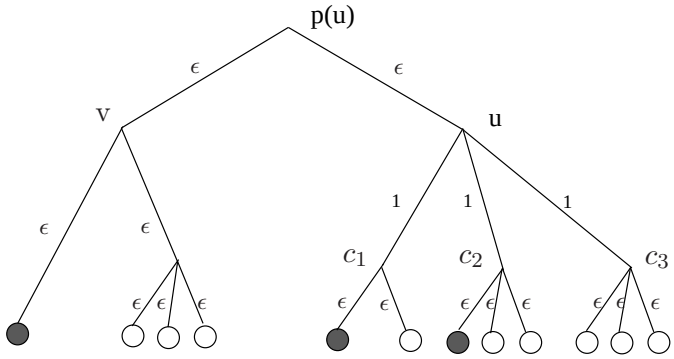

Figure 5.5: An example of a gapless subproblem for the MDCVRP in trees. k=3.

In this example, only the edges (u, c_1), (u, c_2) and (u, c_3) have a cost 1, and all other edges have cost ϵ. It is easy to see that every edge needs to be traversed in the optimal solution of this instance. To avoid traversing (u, c_1), (u, c_2) and (u, c_3) twice, the depot inside T_v needs to service the three customers in T_{c_3}, and the three customers inside T_v will

be serviced by the two depots in T_{c_1} and T_{c_2}. Therefore the edge $(p(u), u)$ is traversed due to two reasons, one is to service some customers outside T_u by depots inside T_u, and the other is to service some customers inside T_u by depots outside T_u. This makes it difficult to design approximation algorithms for the MDCVRP in trees based on the *flow* bound.

Despite this difficulty, the optimal solution for the MDCVRP in trees still has a nice property. Given an MDCVRP feasible solution, let $g(e, x)$ be the number of customers assigned to a depot x where the depot and the customers are separated by e. $g(e, x)$ is considered to be a flow passing through e. We say a flow $g(e = (p(u), u), x)$ is positive, if depot x is in T_u; otherwise we say $g(e, x)$ is negative. We define *net flow* on an edge e to be the summation of all the flows passing through e. The optimal solution of the MDCVRP in trees satisfies the property stated in Lemma 5.3.1. In the literature of network flow algorithms, this is called the *flow conservation property*.

Lemma 5.3.1 (Flow Conservation Property)**.** *In a feasible solution of an MDCVRP instance in a tree, for a non-leaf vertex u, the total flow passing through the edges connecting u to its children equals the total flow through the edge* $(p(u), u)$.

The *flow conservation property* is the basis of our 2-approximation algorithm for the MDCVRP in trees. Before sketching our algorithm, we first define a new flow bound, called the *general flow* bound, as follows. This bound will be used to locate the appropriate subproblems. Each edge $e = (p(u), u)$ of the tree is associated with a number $g(e)$, which represents that either $g(e)$ net customers outside T_u need to be serviced by depots inside T_u, or $|g(e)|$ net customers inside T_u need to be serviced by depots outside T_u. For each customer leaf node u, $g(e = (p(u), u))$ is -1. For each depot leaf node u, $g(e = (p(u), u))$ is $\leq k$. Note that some of the depots may not be used in the solution. We call such an assignment a flow configuration. Let $GFB = 2\sum_e \lceil |\frac{g(e)}{k}| \rceil \cdot c(e)$. We call GFB a *general flow* bound for an MDCVRP instance P, if $GFB \leq OPT_P$ and the corresponding flow configuration is feasible, in the sense that under this configuration the *flow conservation property* is satisfied. Note that the *general flow* bound is a lower bound on the optimal cost only with respect to the whole corresponding flow configuration. Assume that for a *general flow* bound GFB, an edge e is assigned net flow $g(e)$. It is possible that $2\lceil |\frac{g(e)}{k}| \rceil$ is not a lower bound on the number of traversals the vehicles would make on e in the optimal solution. However, if the number of customers is exactly k times the number of depots, then the *flow* bound and the *general flow* bound are equivalent. In the following we denote

$F(GFB)$ to be the flow configuration corresponding to a *general flow* bound GFB, and denote $GFB(P)$ to be the smallest *general flow* bound of an MDCVRP instance P.

Algorithm MDCVRP_trees(T)
Input: an MDCVRP instance P defined on a tree T rooted at r.
Output: a solution bounded by $2 \cdot OPT_P$.

1 $GFB(P) \leftarrow$ finds the smallest possible *general flow* bound

2 $S \leftarrow$ subproblems after removing such edges e in T that $g(e) = 0$ in $F(GFB(P))$

3 Solve the subproblems separately in S

Figure 5.6: Solving the MDCVRP in trees.

In our algorithm (called MDCVRP-trees) for the MDCVRP in trees, we define the problem P' to be finding $GFB(P)$ for an MDCVRP instance P. To avoid the difficulty of figuring out the solution of P directly, our algorithm has two steps. In the first step, we solve P' optimally by dynamic programming. It is easy to see that $GFB(P)$ is a lower bound on OPT_P, as its corresponding flow configuration $F(GFB(P))$ satisfies the *flow conservation property*. In the second step, we form a forest T' by removing the tree edges whose flow is 0 in $F(GFB(P))$. We then formulate the subproblems on the connected components of T'. In the third step, we solve these subproblems separately and obtain the final schedule. Implied by the NP-hardness of the MDCVRP in trees, these subproblems are also NP-hard. A 2-approximation algorithm is designed for each of the subproblems. Therefore our two-phase algorithm is a 2-approximation for the MDCVRP in trees.

In general the subproblems generated in our algorithm can be quite different from the gapless subproblems. For example, for the gapless subproblem in Figure 5.5, our algorithm removes the edges $(p(u), u)$ and $(p(u), v)$ to form two subproblems based on the *general flow* bound. One of them is defined on T_u and the other is defined on T_v. We will prove later in Lemma 5.3.3 that the cost of solving these subproblems is bounded by at most twice the cost of solving the gapless subproblems.

We transform a *general flow* bound of an MDCVRP instance P defined on a graph G to an equivalent *flow* bound on another graph G' as follows. For an edge $e = (p(u), u)$, where u is a depot, we create $k - g(e)$ pseudo customers and connect each of these customers to $p(u)$ with a zero cost edge. Here $g(e)$ is the net flow of e. Let the resulting graph be G'.

It is easy to see that $GFB(P)$ equals the *flow* bound cost in G'. We in fact transform the subproblems in G satisfying the *general flow* bound to subproblems in G' satisfying the *flow* bound. In this way the difficulty mentioned in the beginning of this section is avoided. We can assume now that the subproblems being solved satisfy the *flow* bound.

5.3.2 Locating the subproblems for the MDCVRP in trees

The main feature of our algorithm for locating the subproblems is a table, $table(u)$, for each vertex u. An entry $table(u)[i]$ $(-|V| \leq i \leq |V|$, where V is the set of customers in G, represents the smallest possible *general flow* bound of the MDCVRP instance defined on T_u, given that T_u provides services to i customers outside T_u when i is positive, or $|i|$ customers in T_u need services from outside T_u when i is negative. For each vertex u of G, we initialize $table(u)[0]$ to 0 and all other entries of $table(u)$ to $+\infty$. The pseudo code of generating the tables is shown in Figure 5.7.

Algorithm MDCVRP_GFB(u)
Input: an MDCVRP instance P defined on a tree T_u.
Output: a table showing lower bounds of the optimum.

```
1  if u is a depot then
2       for i = 1 to k do
3            table(u)[i]=2c((p(u), u))
4       endfor
5       return table(u)
6  else if u is a customer then
7       table(u)[−1]=2c((p(u), u))
8       return table(u)
9  endif
10
11 for each child v of u do
```

```
12      table(v) = MDCVRP_GFB(v)
13      tmp = a new table with each entry equal to +∞;
14      for i = −|V| to |V| do
15          for j = max(−|V|, −|V| − i) to min(|V|, |V| − i) do
16              if ((table(v)[i] + table(u)[j]) < tmp[i + j]) and
17                (table(v)[i], table(u)[j] ≠ +∞) then
18                  tmp[i + j] = table(v)[i] + table(u)[j]
19              endif
20          endfor
21      endfor
22      table(u) = tmp
23  endfor
24
25  for i = −|V| to |V| do
26      table(u)[i] = table(u)[i] + 2⌈|i/k|⌉· c((p(u), u))
27  endfor
28  return table(u)
```

Figure 5.7: Locating the subproblems for the MDCVRP in trees.

The table entries are generated as follows. Firstly note that for a given vertex u and a number i, $table(u)[i]$ includes the cost of traversing the edge $e = (p(u), u)$ exactly $2\lceil|\frac{i}{k}|\rceil$ times. This is also shown in the lines 25 to 27 in Figure 5.7. If u is a depot which will be assigned some customers, then the edge from u to its parent has to be traversed exactly

once, so we set the entry $table(u)[i]$ $(1 \leq i \leq k)$ to $2c(e)$. Similarly if u is a customer, we set the entry $table(u)[-1]$ to $2c(e)$, as only u needs service from outside T_u.

For a non-leaf node u, we assume that all the tables belonging to its children have already been computed. Let its children be $c_1, c_2, \cdots, c_t$ (in an arbitrary order). The algorithm just scans this list of children from left to right, and updates the entries of $table(u)$ incrementally. When a child v of u is encountered, we incorporate $table(v)$ into $table(u)$ as follows. We first create a new table tmp with each entry setting to $+\infty$. For two numbers i and j where $-|V| \leq i+j \leq |V|$, we update $tmp[i+j]$ to be $table(v)[i]+table(u)[j]$ if the latter is smaller. By this update, we not only find a feasible solution that satisfies the flow conservation property, but also compute the minimum general flow bound with respect to the subtree of T_u that have already been considered. After processing v, the table tmp becomes the new table $table(u)$ for u.

The subproblems are determined as follows. Let the root of the tree be r. When the algorithm MDCVRP_GFB terminates, we find the entry $table(r)[0]$. We then trace by reversing the process and find the flow configuration A corresponding to $table(r)[0]$. The subproblems are defined on the connected components after removing edges with 0 net flow in A.

5.3.3 Solving the subproblems for the MDCVRP in trees

Given the subproblems satisfying the *flow* bound in G', we design a 2-approximation algorithm for these subproblems. The pseudo code of the main algorithm for solving the subproblems is listed in Figure 5.8.

Algorithm MDCVRP_solve_subproblem(*repository*, T_u**)**
Input: an MDCVRP instance P defined on a tree T_u.
Output: an assignment of the customers to depots.

1 *Comment: the top of repository has already been assigned* k_1 *customers*

3 *top_depot* $\leftarrow$ *repository.top*()

4 if u is a depot then

5 if $k_1 > 0$ then

6 *Comment: keep top_depot the top of repository*

```
7             top_depot ← repository.pop()
8             repository.push(u)
9             repository.push(top_depot)
10        else
11            repository.push(u)
12        endif
13        return
14  else if u is a customer then
15        assign u to top_depot
16        if top_depot has already been assigned k customers then
17            repository.pop()
18        endif
19        return
20  endif
21
22  Comment: u is an internal node
23  for each positive children c of u
24        MDCVRP_solve_subproblem(repository, T_c)
25  endfor
26
27  Comment: the repository has all the depots needed for the rest of T_u
28  for each negative children c of u
```

29 MDCVRP_solve_subproblem(*repository*, T_c)

30 endfor

31

32 return

Figure 5.8: Solving the subproblems satisfying the flow bound for the MDCVRP in trees.

The algorithm (called MDCVRP_solve_subproblem in Figure 5.8) runs recursively. The main component of the algorithm is a stack called *repository* which contains a set of depots available to service the customers. The algorithm traverses the tree in a depth first order and the stack *repository* is updated when a leaf (a depot or a customer) is met. When a depot d_1 is encountered, it is pushed into *repository*. However, if the current top depot d_2 of *repository* has already been assigned some customers, then d_2 will remain in the top and we insert d_1 just below d_2 in *repository*. When a customer is met, we assign this customer to the top depot of *repository*, and we remove the top element from *repository* if it has already been assigned k customers. Therefore, in the algorithm it is always maintained that, customers can only be assigned to the top depot in *repository*. We state this fact in Lemma 5.3.2. This lemma is crucial for our final analysis of the performance guarantee.

Lemma 5.3.2. *At any time during a run of the algorithm MDCVRP_solve_subproblem in Figure 5.8, only the top depot in the stack repository may have been assigned some ($< k$) customers. All the other depots in the stack represent empty vehicles with capacity k.*

When a non-leaf vertex u is being visited, the algorithm descends one level to process its children. Since the algorithm processes all the positive children before the negative children, during the processing of a negative child c, the repository has enough depots to serve the customers in T_c. During the processing of a positive child, the rest of the available depots are pushed into *repository*.

For a positive subtree T_u, we can interpret the number $f(T_u)$ as the remaining capacity of the depots in T_u after servicing the customers (including the pseudo customers) inside T_u. Assume, when visiting a positive branch T_u with $f(T_u) = t_2$, that the top depot in *repository* has a remaining capacity of t_1. When the algorithm leaves T_u, we should be able to collect the remaining capacity t_2 of T_u, and the top depot in *repository* should have a

remaining capacity of $(t_1 + t_2)$ mod k. Note that in order to guarantee the approximation ratio, the top depots in *repository* when the vehicle enters and leaves T_u may not be the same.

Consider the example in Figure 5.5. Our algorithm locates two subproblems, one defined on T_u and the other on T_v. Assume the algorithm visits the positive children of u in the order of c_1 and c_2. Let d_1 and d_2 be the depots in T_{c_1} and T_{c_2} respectively. It is not difficult to see that just before entering T_{c_2}, d_1 is the top depot in the stack *repository* and is assigned the only customer in T_{c_1}. Then d_2 is pushed into *repository*, however, d_1 would remain at the top of *repository* as it has a remaining capacity $< k$. So in this example, instead of d_1, d_2 would be the depot with a remaining capacity of 3 after collecting the remaining capacity of T_{c_2}. It is proved in Lemma 5.3.3, that a positive edge e (the edge (u, c_2) in this example) is traversed at most $FB_e + 2$ times under this strategy.

We present our final analysis for the algorithm MDCVRP_solve_subproblem in Lemma 5.3.3.

Lemma 5.3.3. *In the solution produced by the MDCVRP_solve_subproblem algorithm, each edge $e = (p(u), u)$ is traversed at most $FB_e + 2$ times.*

Proof. We can assume u is not a leaf, as otherwise e would be traversed exactly twice. We assume the algorithm finds the routes $r_1, r_2, \cdots, r_t$ that cross e (listed in increasing order of the time when they were created). We claim that only during the first and last routes the vehicle may cross e to service fewer than k customers. Consider the following cases:

Case 1: e is positive. In the algorithm, the first time that e is traversed is to process T_u (let this time be t_1). When the algorithm returns to u and traverses e from u to $p(u)$ for the first time (let this time be t_2), all the customers inside T_u should have been assigned to some depots, and all the remaining available depots should have been pushed into the stack *repository*. From then on, the algorithm will not traverse e explicitly, but implicitly when using the depots of T_u in *repository* to service customers outside T_u. It is easy to see that during any assignment of customers to a depot in *repository*, the route already established for this depot will not be affected. Therefore, the number of times e is traversed is determined by the number of depots of T_u in *repository* at time t_2. According to Lemma 5.3.2, only the top depot in *repository* at any time may be assigned customers. Assume at time t_1 the top depot d_1 in *repository* is already assigned some, say k_1, customers. According to the algorithm, d_1 will consume $k - k_1$ customers in T_u. Therefore if $k_1 \geq f(T_u)$ mod k,

edge e would be traversed FB_e times, and otherwise, FB_e+2 times in the solution produced by our algorithm.

Case 2: e is negative. In the algorithm, e is traversed for two purposes. Firstly, we may use some depots outside T_u to service customers inside T_u. Secondly, there may be one depot inside T_u which will be used to service some customers outside T_u. The last possibility is due to the fact that our algorithm always assigns customers to the top depot of the stack *repository*. Let t be the time just before T_u is entered for the first time. Then the number of times e is traversed, is determined by the capacities of the depots in *repository* at time t. According to Lemma 5.3.2, only the top depot in *repository* may have been assigned customers, therefore e will be traversed exactly FB_e times if the top depot in *repository* has a remaining capacity of at least $f(T_u)$ mod k and no depot in T_u is assigned customers outside T_u, and $FB_e + 2$ times otherwise.

□

Therefore we establish the following theorem:

Theorem 5.3.4. *The algorithm MDCVRP_trees is a 2-approximation for the MDCVRP in trees. The running time of the algorithm is $O(|V|^3)$.*

Proof. Given an instance P of the MDCVRP in trees, in the first step we compute the $GFB(P)$. $GFB(P)$ is a lower bound on the optimum OPT_P. For each subproblem induced by $GFB(P)$, the algorithm MDCVRP_solve_subproblem will produce a solution where each edge is traversed at most $FB_e + 2$ times. As each edge will be traversed at least 2 times in the subproblems, summing up the costs on each edge, the cost of the final solution is bounded by $2OPT_P$. □

Recall that in the vehicle dispatching problem (VDP), the vehicles service the customers as in the MDCVRP, but the vehicles do not return to their depots after the completion of the routes. We have

Lemma 5.3.5. *There is a solution for the VDP in trees where each edge $e = (p(u), u)$ is traversed at most $\frac{1}{2}FB_e + 2$ times.*

Proof. We apply the strategy similar to that in the algorithm in Figure 5.8 for the VDP, except that each vehicle does not return to its depot. When a negative edge $e = (p(u), u)$ is traversed, if a depot does not accrue sufficient vertices after traversing $e = (p(u), u)$ from

$p(u)$ to u, e has to be traversed once more from u to $p(u)$. For each edge, this case occurs at most once. Therefore e will be traversed at most $\frac{1}{2}FB_e + 2$ times under this strategy. □

Theorem 5.3.6. *There is a 3-approximation algorithm for the VDP in trees. The running time of the algorithm is $O(|V|^3)$.*

Proof. Note that $\frac{1}{2}FB_e$ is a lower bound on the number of traversals the vehicle will make on e in the optimal solution of the transformed subproblems. □

5.4 Transforming the subproblems for the MDCVRP in general graphs

It is not difficult to see that $GFB(P)$ of an MDCVRP instance P is equal to the optimal solution of the following nonlinear program:

$$(NLP)\ \ Min\ \ \sum_{i,j} 2 \cdot c(ij)\lceil \tfrac{x(ij)}{k} \rceil$$

subject to:

$$\sum_j x(ij) - \sum_j x(ji) = b_i \qquad \forall i \in V \cup D \cup \{s\}$$

$$0 \le x(ij) \le u(ij) \qquad \forall i, j \in V \cup D \cup \{s\}$$

We add a new vertex s to the original graph G, and connect s to each depot by an edge with zero cost and capacity k. All other edges have infinite capacities. An integer b_i is associated with each vertex $i \in V$. For a customer vertex i, b_i is set to -1. For a depot vertex i, b_i is equal to 0. For the new vertex s, we set b_i to $|V|$.

For each edge $e = (i, j)$ where $i, j \in V$, we denote $c(ij)$ to be its edge cost, $u(ij)$ to be its capacity, and we associate it with a flow $x(ij)$ or $x(ji)$. Here both $x(ij)$ and $x(ji)$ are positive, and $x(ij)$ represents the flow from i to j, and $x(ji)$ represents the flow from j to i.

(NLP) falls into the category of minimum cost network flow problems with concave fixed charge functions. Consider the following nonlinear program:

$$(NLP')\ \ Min\ \ \sum_{i,j} c(ij)h(x(ij))$$

subject to:

$$\sum_j x(ij) - \sum_j x(ji) = b_i \qquad \forall i \in V \cup D \cup \{s\}$$

$$0 \le x(ij) \le u(ij) \qquad \forall i, j \in V \cup D \cup \{s\}$$

where every edge in the graph has a capacity of k, and the function h is defined as

$$h(x(ij)) = \begin{cases} 0 & : \quad x(ij) = 0 \\ 1 & : \quad x(ij) > 0 \end{cases}$$

It is proved in [37] that (NLP') (even the uncapacitated case) is NP-hard. Computing the minimum *general flow* bound is therefore NP-hard for general graphs. In this section we prove that given an α-approximation algorithm for computing (NLP), there is a $(6 \cdot \alpha)$-approximation algorithm (please see Figure 5.9) for the MDCVRP in general graphs.

Algorithm $MDCVRP\_general(G, A)$
Input: An MDCVRP instance P defined on a general graph $G = (V, E)$. An α-approximation algorithm A for computing $GFB(P)$.
Output: A feasible MDCVRP solution with cost bounded by $3\alpha \cdot GFB(P)$.

1 $S \leftarrow$ subproblems obtained by applying A on G
2 for each subproblem P_1 in S
3 obtain P_2 by applying the transformation algorithm in Figure 5.12 on P_1
4 solve P_2 by the algorithm in Figure 5.8
5 end for

Figure 5.9: An approximation algorithm for the MDCVRP in general graphs.

In the following we present the details of the transformation algorithm. Let P_1 be a subproblem obtained after computing $GFB(P)$. Let G_1 be the subgraph of the original graph G consisting of the edges with positive flows in P_1. In addition we assume that the edges in G_1 are directed, e.g. for an edge $e = (i, j)$ in G, if $x(ij) > 0$ in P_1, then a directed edge will be created in G_1 with head i and tail j. It is easy to see that directed cycles in G_1, if there is any, can be removed without breaking the feasibility or degrading the quality of the solution. Let $i_1, i_2, \cdots, i_l, i_1$ be such a cycle. Without loss of generality, assume the flow x is minimum on $e = (i_l, i_1)$ among the flows on the edges of the cycle. We then decrease the flow by x along the edges in the cycle. It is easy to see that the resulting solution still satisfies the flow conservation property. As e is no longer in the solution, the cost of the new solution is strictly less than that of the original solution.

Therefore G_1 can be assumed to be a *directed acyclic graph* (DAG). We show in Figure 5.10 that G_1 may not be a tree.

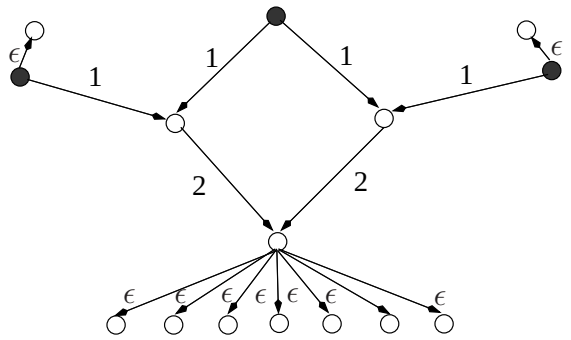

Figure 5.10: An example showing that each subproblem may not induce a tree. $k = 4$.

In this example the edges in the subproblem have costs of ϵ (a very small positive number), 1, or 2. It is assumed here that all other edges in Figure 5.10 have costs as large as possible without breaking the triangle inequality. It is easy to verify that the solution in Figure 5.10 is optimal.

For a graph G', we define $GFB'(G')$ to be $\sum_e 2c(e)\lceil\frac{g(e)}{k}\rceil$, where each edge in G' is associated with a flow $g(e)$ in G', for a given integer k. In the following we show that G_1 can be transformed into an arborescence G_2, and $GFB'(G_2) \leq 2 \cdot GFB'(G_1)$. An arborescence is a directed rooted tree where each edge points away from the root. We define a *pseudo cycle* to be two directed simple paths $P_1 = u_1, u_2, \cdots, u_{l-1}, u_l$ and $P_2 = u_1, u'_2, \cdots, u'_m, u_l$, where $m \leq l$. These two paths only share their two endpoints u_1 and u_l. An example of a pseudo cycle is given in Figure 5.11(a).

An operation called *canceling a pseudo cycle* is used in the transformation algorithm. Assume the pseudo cycle consists of two paths P_1 and P_2. Without loss of generality, assume P_1 has a total edge cost no greater than that of P_2. We first locate the edges with the minimal flow x in P_2. We then cancel flow x along the edges in P_2, and re-push this flow through the edges in P_1. Those edges with zero flow after this operation are removed from the current graph.

An example of canceling a pseudo cycle is illustrated in Figure 5.11. Assume that in this example the total cost of the edges of P_1 is no greater than that of P_2. In Figure 5.11(a), edge $e = (u'_3, u_4)$ has the minimum flow 3 in P_2. We cancel flow x along the edges in P_2, and

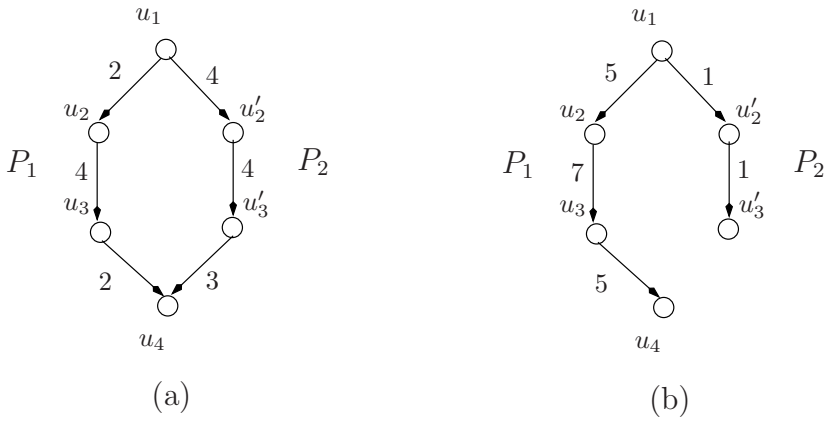

Figure 5.11: A pseudo cycle example.

re-push this flow through the edges in P_1. The resulting graph is shown in Figure 5.11(b).

The psuedo code of the transformation algorithm is given in Figure 5.12.

The algorithm simply detects whether a pseudo cycle exists in the current graph G_2. Such a pseudo cycle, if it exists, is then canceled by removing one or more edges from G_2. Since in each iteration at least one edge will be removed from G_2, the algorithm runs in time $O((|D| + |V|)^3)$.

The ceiling function has the property stated in Lemma 5.4.1.

Lemma 5.4.1. *Given a positive integer k, for two non-negative numbers A and B where $A \leq B$, $\lceil \frac{A}{k} \rceil + \lceil \frac{B-A}{k} \rceil \geq \lceil \frac{B}{k} \rceil \geq \lceil \frac{A}{k} \rceil + \lceil \frac{B-A}{k} \rceil - 1$.*

We show that $GFB'(G_2) \leq 2 \cdot GFB'(G_1)$ in Lemma 5.4.2.

Lemma 5.4.2. *Let G_2 be the resulting graph after applying the algorithm in Figure 5.12 on a graph G_1 where each edge is associated with a flow satisfying the constraints in (NLP). Then $GFB'(G_2) \leq 2 \cdot GFB'(G_1)$.*

Proof. Let an edge e get a nonzero flow $f_1(e)$ and a flow $f_2(e)$ in G_1 and G_2 respectively. It is easy to see that $\sum_{e \in G_1} c(e) \cdot \frac{f_1(e)}{k} \geq \sum_{e \in G_1} c(e) \cdot \frac{f_2(e)}{k}$. This is due to the fact that when canceling a pseudo cycle C, we re-push a flow x to the path with a smaller total edge cost. Therefore this flow incurs a smaller cost after canceling C.

Let S_+ consist of all such edges e that $f_2(e) \geq f_1(e)$, and let S_- consist of all such edges e that $f_2(e) < f_1(e)$. Since $\sum_{e \in G_1} c(e) \cdot \frac{f_1(e)}{k} \geq \sum_{e \in G_1} c(e) \cdot \frac{f_2(e)}{k}$, we have that

Algorithm $Transform(G_1)$
Input: A connected graph G_1 where each edge is associated with a flow satisfying the constraints in (NLP).
Output: An arborescence G_2 where each edge is associated with a flow satisfying the constraints in (NLP), $GFB'(G_2) \leq 2 \cdot GFB'(G_1)$.

1 $G_2 \leftarrow G_1$

2 while exists a pseudo cycle C

3 $\quad G_2 \leftarrow$ the resulting graph after canceling C

4 end while

5 return G_2

Figure 5.12: Transform an MDCVRP subproblem to a new subproblem defined on a tree.

$$\sum_{e\in S_+} c(e) \cdot \tfrac{f_2(e)-f_1(e)}{k} \leq \sum_{e\in S_-} c(e) \cdot \tfrac{f_1(e)-f_2(e)}{k} \qquad (1)$$

The total cost of the transformed solution is $\sum_{e\in S_+} c(e)\cdot(\lceil\frac{f_2(e)}{k}\rceil)+\sum_{e\in S_-} c(e)\cdot(\lceil\frac{f_2(e)}{k}\rceil)$. According to Lemma 5.4.1, this cost is at most

$$\begin{aligned}
& \sum_{e\in S_+} c(e)\cdot(\lceil\frac{f_1(e)}{k}\rceil+\lceil\frac{f_2(e)-f_1(e)}{k}\rceil)+\sum_{e\in S_-} c(e)\cdot(\lceil\frac{f_2(e)}{k}\rceil)\\
\leq & \sum_{e\in S_+} c(e)\cdot(\lceil\frac{f_1(e)}{k}\rceil+\frac{f_2(e)-f_1(e)}{k}+1)+\sum_{e\in S_-} c(e)\cdot(\lceil\frac{f_2(e)}{k}\rceil)\\
\leq & \sum_{e\in S_+} c(e)\cdot(\lceil\frac{f_1(e)}{k}\rceil+1)+\sum_{e\in S_-} c(e)\cdot\frac{f_1(e)-f_2(e)}{k}+\sum_{e\in S_-} c(e)\cdot(\lceil\frac{f_2(e)}{k}\rceil)\\
\leq & \sum_{e\in S_+} c(e)\cdot(\lceil\frac{f_1(e)}{k}\rceil+1)+\sum_{e\in S_-} c(e)\cdot\frac{f_1(e)-f_2(e)}{k}+\sum_{e\in S_-} c(e)\cdot(\frac{f_2(e)}{k}+1)\\
\leq & \sum_{e\in S_+} c(e)\cdot\lceil\frac{f_1(e)}{k}\rceil+\sum_{e\in S_-} c(e)\cdot\frac{f_1(e)}{k}+\sum_{e\in G_1} c(e)\\
\leq & \frac{1}{2}GFB'(G_1)+\sum_{e\in G_1} c(e),
\end{aligned}$$

As every edge in G_1 is counted at least twice in $GFB'(G_1)$, we establish that $GFB'(G_2) \leq 2 \cdot GFB'(G_1)$.

□

Due to Lemma 5.4.2 and Theorem 5.3.4, we have

Theorem 5.4.3. *If there is an α-approximation to compute (NLP), then there exists an 6α-approximation for the MDCVRP in general graphs.*

Proof. Note that for each edge e, the approximation ratios are additive when transforming the subproblems and establishing the actual schedule of the subproblems. And for the MDCVRP, GFB is bounded by twice of the MDCVRP optimum. □

5.5 6-Approximation for the MDCVRP in graphs with bounded branch/tree-width

The algorithm in Figure 5.7 can be extended easily to compute the minimum *general flow* bound in graphs with bounded branch/tree-width. In the following we first give a brief introduction to branch decomposition and tree decomposition.

The branch decomposition of a graph $G = (V, E)$ is a ternary *tree* T and a bijection from the set of leaves of T to E. Removing an edge, say e, from T will produce two subtrees T_1 and T_2 of T. T_1 and T_2 induce a partition of E into E_1 and E_2. The *middle set* of e, denoted by $mid(e)$, consists of the vertices of V which are incident to the edges in E_1 and E_2. The width of an edge $e \in T$, denoted by $|mid(e)|$, is defined to be the number of vertices in the middle set corresponding to e. The width of a branch decomposition T is the maximum width of all the edges in T. The *branch-width* of a graph G, denoted by $\beta(G)$, is the maximum width of all the branch decompositions of G. An example branch decomposition is given in Figure 5.14. This example is from [45]. The original example graph is given in Figure 5.13.

It is NP-hard to compute the optimal branch-width and find an optimal branch decomposition of a graph. However, as shown in [83], a polynomial time algorithm exists to approximate the branch-width of a graph within a factor of 3. Therefore in the following, we assume that a branch decomposition is already given and the branch-width is some constant.

Similar to other branch decomposition based algorithms, for a given branch decomposition T with branch-width $\beta(G)$, our algorithm for computing GFB of G consists of two steps. In the first step, T is transformed to a rooted binary tree T'. This can be done by

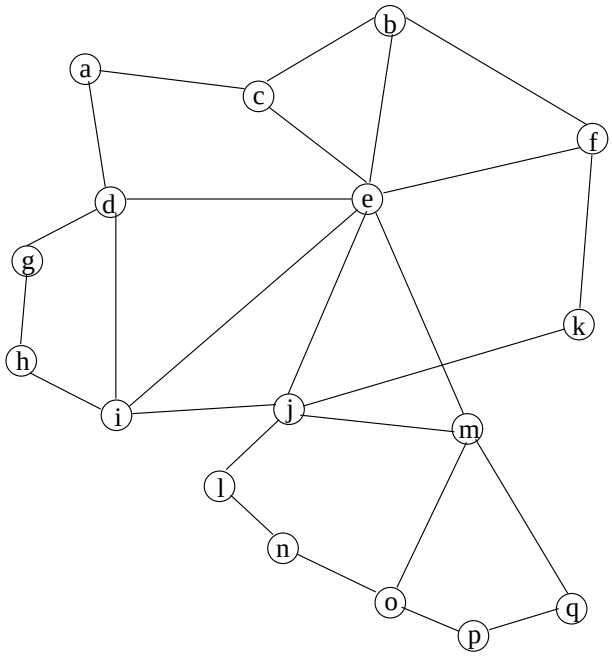

Figure 5.13: An example graph for branch decomposition.

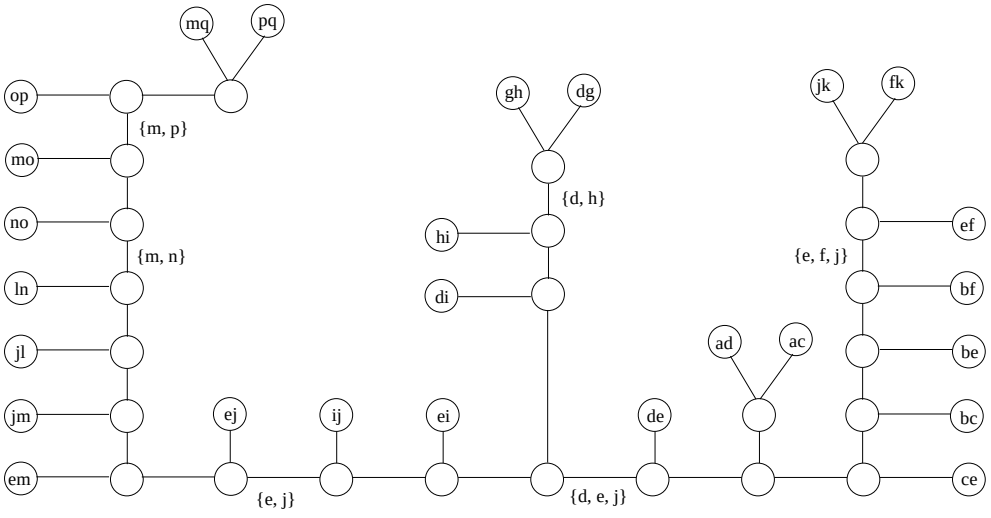

Figure 5.14: A branch decomposition of width 3 for the graph in Figure 5.13.

selecting an edge (u, v) from T and replacing e by a root r, and two new edges (r, u) and (r, v). In the second step, we modify the algorithm in Figure 5.7 with input T' to compute GFB of G as follows. We still associate each vertex u of T with a table $table(u)$. However, the dimension of $table(u)$ is equal to the width $|mid(e)|$ of the edge $e = (p(u), u)$. Given an arbitrary order of the vertices in $mid(e)$, the ith dimension of $table(u)$ denotes the flow balance on the ith vertex in $mid(e)$. For each node v of T', let G_v denote the subgraph of G induced by the edges corresponding to the leaves in T'_v. Let the ith dimension of $table(u)$ take a value b_i in a particular entry of $table(u)$. This entry of $table(u)$ then represents the smallest *general flow* bound defined on G_v, given that the ith vertex in $mid(e)$ will receive a flow b_i from, or send a flow b_i to, the vertices in $G - G_v$. As $|V|$ customers exist in the graph, the number of entries in $table(u)$ is bounded by $|V|^{\beta(G)}$.

For a vertex u in T', its table will be computed after the tables of its two children have already been built. Let its children be c_1 and c_2, and let $e_1 = (u, c_1)$ and $e_2 = (u, c_2)$. Let n_u be the target entry in $table(u)$, when merging two entries n_1 and n_2 which belong to $table(c_1)$ and $table(c_2)$ respectively. Consider a vertex v in $mid(e_1)$ and $mid(e_2)$. If v only appears in one of the two sets, say $mid(e_1)$, then the value of the dimension in n_u corresponding to v will be equal to that in n_1. If v is in both sets, two cases need to be examined. When v is also in $mid(e)$, the value of the dimension in n_u corresponding to v will be equal to the sum of that in n_1 and n_2. When v is absent in $mid(e)$, the sum of the values of the dimensions in n_1 and n_2 corresponding to v should be zero. Otherwise the flow conservation property will be violated and there is no need to combine these two entries.

In the following we briefly introduce tree decomposition and tree-width. A tree decomposition of a graph $G = (V, E)$ is denoted by (T, I), where T is a tree and $I = \{i : X_i \subset V\}$. A tree decomposition satisfies:

(1) $\cup_{i \in I} X_i = V$,

(2) for each edge $e = (u, v) \in E$, $\exists i \in I$, s.t. $u, v \in X_i$,

(3) for all $u \in V$, $\{i \in I : u \in X_i\}$ induces a connected subtree of T.

The *width* of a tree decomposition (T, I) is defined to be $\max_{i \in I} |X_i| - 1$. The *tree-width* of a graph G, denoted by $\tau(G)$, is the minimum width over all tree decompositions of G. A graph is a *partial k-tree* if and only if it has a tree-width k. The above dynamic program

to compute GFB assumes constant branch-width. It is shown in Theorem 5.5.1 [82] that branch-width and tree-width are closely related to each other.

Theorem 5.5.1 (Robertson and Seymour)**.** *Let $G = (V, E)$ be a graph, with $E \neq \emptyset$. Then $max(\beta(G), 2) \leq \tau(G) + 1 \leq max(\lfloor \frac{3}{2}\beta(G) \rfloor, 2)$.*

As the minimum *general flow* bound can be optimally computed for graphs with bounded branch/tree-width, further due to Theorem 5.4.3, we establish

Lemma 5.5.2. *There is a 6-approximation for the MDCVRP in graphs with bounded branch/tree-width.*

Similarly we have the following results for the VDP:

Theorem 5.5.3. *If there is an α-approximation to compute (NLP), then there exists an 4α-approximation for the VDP in general graphs.*

Proof. This is due to Lemma 5.4.2 and Lemma 5.3.5. □

Lemma 5.5.4. *There is a 4-approximation for the VDP in graphs with bounded branch/tree-width.*

5.6 k-Approximation for the MDCVRP in general graphs

In this section we show a k-approximation algorithm for the MDCVRP in general graphs. We define a k-factor of G as a set of edges $E_k \subseteq E$, such that for each depot $v \in D$, $\sigma(v) \leq k$, and for each customer, $v \in V$, $\sigma(v) = 1$, where $\sigma(v)$ is the number of edges of E_k incident on v. In the following we denote the cost of k-factor E_k by $\|E_k\|$.

This algorithm is similar to that in [60] in the sense that the final solution is obtained from a minimum cost k-factor of G. For the sake of completeness, we describe the algorithm for the MDCVRP below.

Algorithm $MDCVRP(k)$

Step 1 Construct a minimum cost flow instance H as follows.

We first construct a complete bipartite graph H where one part contains the depots, and the other part contains the customers. We then add a vertex s to H, and connect

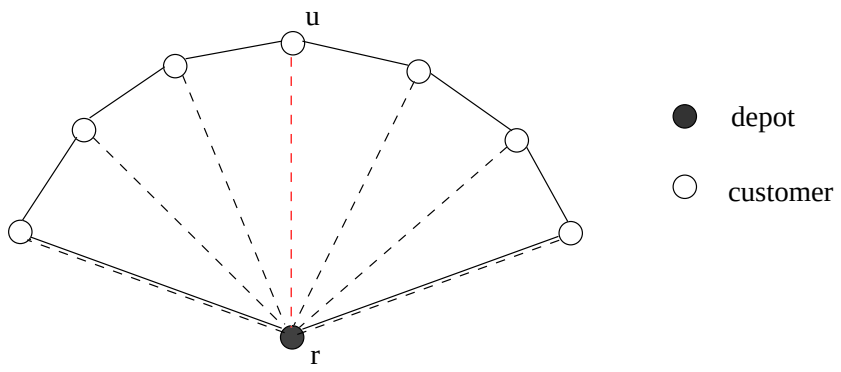

Figure 5.15: A k-factor obtained from an optimal solution. Solid lines represent the edges in an optimal solution and the dashed lines represent the edges in the corresponding k-factor.

s to each depot by an edge with cost of 0 and capacity of k. We further associate a positive integer $|V|$ (supply) with s, and -1 (demand) with each customer in H.

Step 2 Find the minimum cost k-factor E_k.

This can be done by running minimum cost flow algorithms on H.

Step 3 For each depot r, find any tour involving r and the customers connected to r in E_k.

In the following we bound the cost of the minimum k-factor to the optimal solution OPT. As shown in Figure 5.15, a k-factor can be obtained from the optimal solution by connecting each depot to the vertices in the same tour with the depot. Moreover, the cost of such a k-factor is bounded by $\frac{k}{2}\cdot$ OPT where OPT is the cost of the optimal solution. We show this bound as follows. Consider a tour τ in an optimal solution involving a depot r (as in Figure 5.15). Let S be the set of edges from r to the customers in τ. We arbitrarily pair the edges in S. According to the triangle inequality, the cost of each pair of edges is bounded by OPT. If k is odd, we are left with an edge, e.g. the edge (u, r) in Figure 5.15. Adding (u, r) to τ introduces two cycles. Therefore the cost of (u, r) is at most $\frac{1}{2}$OPT. This proves that there is a k-factor E_k whose cost is bounded by $\frac{k}{2}\cdot$ OPT. Due to the triangle inequality, the cost of the final solution produced by the algorithm (Step 3) is bounded by twice of $\|E_k\|$. Therefore this algorithm gives a k-approximation for the MDCVRP in general graphs.

5.7 3-Approximation algorithm for the MVSP in trees

In this section we show that our way of applying dynamic programming for approximation algorithms on the VRP can be used to design a 3-approximation algorithm for the MVSP [52] in trees. In the MVSP we are given an undirected complete graph $G = (V, E)$, where each vertex u_i of V is associated with a job j_u, and each edge e has a non-negative weight $c(e)$. The edge costs in G satisfy the triangle inequality. There are also m identical vehicles available to service the jobs. Each job j_u has its own release time $r(u)$ and handling time $h(u)$. A job j_u can only be serviced after its release time $r(u)$, and the handling time $h(u)$ represents the time needed to finish processing j_u. The objective is to find a schedule in which the maximum completion time of the jobs, i.e. the makespan, is minimized.

5.7.1 Defining the problem P' for the MVSP in trees

We first introduce several lower bounds for a gapless subproblem of an instance of the MVSP in trees. These lower bounds are used to design approximation algorithms for the MVSP on paths [52]. For a gapless subproblem P_1, we define two lower bounds for the makespan:

$LB_1(P_1) = \max_{u \in V(P_1)}\{r(u) + h(u)\}, \quad LB_2(P_1, m') = \frac{W(P_1)+H(P_1)}{m'}$

where $V(P_1)$ and $E(P_1)$ are the vertex set and edge set involved in P_1 respectively, m' represents the number of vehicles used to service the jobs in P_1, and $W(P_1)$ and $H(P_1)$ are the total edge weights of $E(P_1)$ and the total handling times of the jobs associated with the vertices in $V(P_1)$ respectively.

We define a vehicle configuration to be any partition of the tree where each connected component C_i in the partition is associated with a positive integer m_i and $\sum_i m_i = m$. Given a vehicle configuration VC, let $C_{VC}(u)$ to be the set of connected components inside T_u under VC. The connected components in $C_{VC}(u)$ contain only the vertices in T_u. Let $S_{VC}(u)$ be the set of subproblems defined on the corresponding connected components in $C_{VC}(u)$. We define a new bound $LB(G, m)$ to be $\min_{VC} \max_{P_2 \in S_{VC}(r)}(LB(P_2, m') = LB_1(P_2) + (2W(P_2) + H(P_2))/m')$, where r is the root of the tree and m' is the number of vehicles allocated for P_2 under VC. Our relaxation problem P' for the MVSP is just to find the set of subproblems with the smallest possible bound $LB(G, m)$. We first solve P' optimally by dynamic programming, and by doing so the original problem is decomposed to a set S of subproblems. Since in the computation of $LB(G, m)$ the gapless subproblems will be considered, $LB(G, m)$ is bounded by $3 \cdot OPT$. In the second step we find a feasible

schedule for the subproblems in S with a makespan of at most $LB(G, m)$. We first justify the second step of our algorithm in Lemma 5.7.1. Its proof is constructive.

Lemma 5.7.1. *There is a feasible solution with cost bounded by $3 \cdot OPT$ for the connected components corresponding to the smallest possible bound $LB(G, m)$.*

Proof. Consider a gapless subproblem P_1 in the optimal solution of an instance of the MVSP in trees. Assume m' vehicles are involved in P_1. In the optimal solution for P_1, an edge might be traversed by more than one vehicle. We double the tree edges and obtain a Hamiltonian path containing all the jobs (by a depth first traversal of the tree). We traverse the path from the root, and associate the handling times with the jobs when they are visited for the first time. Therefore we transform P_1 for the MVSP in trees to another gapless subproblem P_1' of the MVSP on paths with $W(P_1') = 2W(P_1)$, $H(P_1') = H(P_1)$ and $LB_1(P_1') = LB_1(P_1)$. Using the algorithm in [52], we obtain a schedule for P_1' with cost bounded by $LB_1(P_1') + LB_2(P_1', m')$. This schedule is also feasible for P_1 and has a cost of $LB_1(P_1) + (2W(P_1) + H(P_1))/m' = LB(P_1, m')$.

The above argument can be applied on the subproblems defined on the connected components obtained after computing $LB(G, m)$. Since $LB(G, m)$ is the smallest possible, the cost of this schedule is no larger than that of after applying the same operation on the gapless subproblems. Therefore the cost of the schedule is bounded by $3 \cdot OPT$. □

The above proof also shows how to solve the subproblems once they are located, therefore in the following we only focus on how to locate the appropriate subproblems.

5.7.2 Locating the subproblems for the MVSP in trees

The core of our solution is an algorithm to solve the following simple feasibility decision problem D: Given a real number λ, is $LB(G, m) \leq \lambda$?

Solving the decision problem

We solve the feasibility problem also by dynamic programming. Given a vehicle configuration VC, we denote $P_{VC}(u)$ to be the subproblem defined on the connected component containing u in $C_{VC}(u)$, and $P'_{VC}(u)$ to be the set of subproblems defined on the connected components not containing u in $C_{VC}(u)$. Recall that $C_{VC}(u)$ consists of all the connected components in T_u under VC.

To solve the decision problem, we maintain two tables, $table(u)$ and $obj(u)$, for an arbitrary vertex u. The table $obj(u)$ has one dimension, and an entry $obj(u)[i](1 \leq i \leq m)$ indicates that whether the bound $LB(T_u, i)$ is at most λ for the subproblem defined on T_u under all possible vehicle configurations. Given that the edge $(p(u), u)$ is not traversed by any vehicle and that in total i vehicles are used to service the jobs in T_u, we denote the bound $LB(T_u, i)$ to be $\min_{VC} \max_{P_2 \in S_{VC}(u)}(LB(P_2, m') = LB_1(P_2) + (2W(P_2) + H(P_2))/m')$, where m' is the number of vehicles allocated for P_2 under VC.

The other table $table(u)$ for u has 3 dimensions. Assume the minimum value of an entry $table(u)[i_1][i_2][i_3]$ of $table(u)$ is obtained under the vehicle configuration VC. Then this value represents $2W(P_{VC}(u)) + H(P_{VC}(u))$, given that i_1 vehicles have been used to service the jobs in $P_{VC}(u)$, i_2 equals $LB_1(P_{VC}(u))$, and i_3 vehicles have been used to service the jobs in the subproblems of $P'_{VC}(u)$. In other words, given the constraints of i_1, i_2, and i_3, the value of the entry $table(u)[i_1][i_2][i_3]$ equals the minimum of $2W(P_{VC}(u)) + H(P_{VC}(u))$, for any vehicle configuration VC in T_u.

For each vertex u, we initialize the table entries $table(u)[i_1][r(u) + h(u)][0]$, where $0 \leq i_1 \leq m$, to $h(u)$. Other entries of $table(u)$ are set to $+\infty$. Given an input of λ, our algorithm maintains that an entry $table(u)[i_1][i_2][i_3]$ is not $+\infty$ if and only if there exists a vehicle configuration VC under which the lower bound LB of each subproblem in $P'_{VC}(u)$ is less than or equal to λ. In the following we define $LB_1(G)$ to be $\max_{u \in V}\{r(u) + h(u)\}$.

Algorithm MVSP_decision(T_u, m, λ)
Input: an MVSP instance P defined on a tree T_u, an integer m and a real number λ.
Output: a table showing a lower bound $LB(T_u, m)$ (at least λ).

1 if u is a leaf then

2 $obj(u)[1] = r(u) + 2h(u)$

3 $table(u)[1][r(u) + h(u)][0] = h(u)$

4 return $table(u)$

5 endif

6

7 for each child v of u do

8 $table(v)$ = MVSP_decision(T_v, m, λ)

9 tmp = a new table with each entry equal to $+\infty$

10 for $i_1 = 0$ to m do

11 for $i_2 = 0$ to $LB_1(G)$ do

12 for $i_3 = 0$ to $m - i_1$ do

13 *Comment: when $e = (u, v)$ is a cut edge*

14 for $j = 0$ to $m - i_1 - i_3$ do

15 if $table(u)[i_1][i_2][i_3] < tmp[i_1][i_2][i_3 + j]$ and $obj(v)[j] \leq \lambda$ then

16 $tmp[i_1][i_2][i_3 + j] = table(u)[i_1][i_2][i_3]$

17 endif

18 endfor

10 *Comment: when $e = (u, v)$ is not a cut edge*

20 for $j_1 = 0$ to $m - i_1 - i_3$ do

21 for $j_2 = 0$ to $LB_1(G)$ do

22 for $j_3 = 0$ to $m - i_1 - i_3 - j_1$ do

23 $t_2 = \max\{i_2, j_2\}$

24 if $(table(u)[i_1][i_2][i_3] + table(v)[j_1][j_2][j_3] + 2c((u, c))) < tmp[i_1 + j_1][t_2][i_3 + j_3]$ then

25 $tmp[i_1 + j_1][t_2][i_3 + j_3] = table(u)[i_1][i_2][i_3] + table(v)[j_1][j_2][j_3] + 2c((u, v))$

26 endif

27 endfor

```
28              endfor
29            endfor
30          endfor
31        endfor
32    endfor
33    table(u) = tmp
34  endfor
35
36  Comment: consider the case when e = (p(u), u) is a cut edge
37  for i_1 = 0 to m do
38    for i_2 = 0 to LB_1(G) do
39      for i_3 = 0 to m − i_1 do
40        if (i_2 + table(u)[i_1][i_2][i_3]/i_1 < obj(u)[i_1 + i_3]) then
41          obj(u)[i_1 + i_3]=i_2 + table(u)[i_1][i_2][i_3]/i_1
42        endif
43      endfor
44    endfor
45  endfor
46  return table(u)
```

Figure 5.16: Solving the decision problem for the MVSP in trees.

The pseudo code of generating the tables is given in Figure 5.7. For a leaf u, since there must be a vehicle to service u, and since there are no other jobs in T_u, only the entry

$table(u)[1][r(u)+h(u)][0]$ needs to be set to $h(u)$ for the case when the edge $(p(u),u)$ is not a cut edge. The edge cost of $(p(u),u)$ will be considered later when merging $table(u)$ to $table(p(u))$. When $(p(u),u)$ is a cut edge, we are ready to compute the objective for T_u, and we accordingly set the entry $obj(u)[1]$ to $r(u)+2h(u)$.

When u is a non-leaf node, we proceed as in the dynamic programming algorithm for locating subproblems for the MDCVRP in trees. We assume that all the tables belonging to its children have already been computed. Let its children be $c_1, c_2, \cdots, c_t$ (in an arbitrary order). The algorithm scans this list of children from left to right, and incorporates the tables of the children into $table(u)$ (one at a time). The table $obj(u)$ is updated after all its children are considered. An entry of $obj(u)$ is updated as follows.

$$obj(u)[i_1+i_3]=\min_{i_2}(i_2+table(u)[i_1][i_2][i_3]/i_1)$$

where $0 \le i_1 \le m$, $1 \le i_2 \le LB_1(G)$, $0 \le i_3 \le m$. Here $LB_1(G)$ represents the first lower bound LB_1 for all the jobs in the graph.

For an entry $table(u)[i_1][i_2][i_3]$, assume its value is obtained under the vehicle configuration VC, then it represents $2W(P_{VC}(u))+H(P_{VC}(u))$. Recall that i_1 denotes the number of vehicles used in $P_{VC}(u)$, and i_3 records the number of vehicles used in the subproblems of $P'_{VC}(u)$. Therefore the total number of vehicles used in T_u corresponding to the entry $table(u)[i_1][i_2][i_3]$ is i_1+i_3. As we assume the edge $e=(p(u),u)$ is a cut edge, and i_2 represents $LB_1(P_{VC}(u))$, we can then find the minimum for $obj(u)[i_i+i_3]$ by computing the lower bound $LB(T_u, i_1+i_3)$ after trying all possible values of i_2.

The updating of $table(u)$ for a vertex u is crucial for solving the decision problem D. Assume $table(v)$ has already been computed before we start to incorporate $table(v)$ into $table(u)$. Let $table_v(u)$ be the new table for u after incorporating $table(v)$. Every entry of $table_v(u)$ is initialized to $+\infty$, and $table(u)$ will be set to $table_v(u)$ after processing v. We have the following cases:

Case 1: (u,v) is a cut edge. In this case an entry of $table_v(u)$ is updated as follows.

$$table_v(u)[i_1][i_2][i_3+j] = \min_{i_3} table(u)[i_1][i_2][i_3], \text{ given that } obj(v)[j] \le \lambda.$$

Here $0 \le i_1 \le m$, $1 \le i_2 \le LB_1(G)$, $0 \le i_3 \le m$, $0 \le j \le m-i_1-i_2$, and λ is the input of the feasiblility decision problem. The entry $table_v(u)[i_1][i_2][i_3+j]$ will be updated if $table(u)[i_1][i_2][i_3]$ is smaller for all possible i_3 and j.

Note that an entry $table(u)[i_1][i_2][i_3]$ has a meaningful value (not $+\infty$), if and only if under the corresponding vehicle configuration VC, $LB(P_2, m') \leq \lambda$ holds for every subproblem P_2 in $P'_{VC}(u)$. Here m' denotes the number of vehicles allocated for P_2 under VC. This is implemented by line 15 in Figure 5.16. Recall that we are solving the feasibility decision problem. Therefore if $LB(P_2, m') > \lambda$ for some subproblem P_2 in $P'_{VC}(u)$, then we already know that VC is not a feasible vehicle configuration and there is no need to complete the rest of the computation for VC.

Under this case we assume that (u, v) is a cut edge and j vehicles are used to service the jobs in T_v. Therefore for all such j that $obj(v)[j] \leq \lambda$, we update $table_v(u)[i_1][i_2][i_3 + j]$ to $table(u)[i_1][i_2][i_3]$ if the latter is smaller, as the third dimension of $table(u)$ represents the number of vehicles used in the subproblems of $P'_{VC}(u)$.

Case 2: (u, v) is not a cut edge. Under this case an entry of $table_v(u)$ is updated as follows.

$$table_v(u)[i_1 + j_1][t_2][i_3 + j_3] = table(u)[i_1][i_2][i_3] + table(c)[j_1][j_2][j_3] + 2c((u, v))$$

where $0 \leq i_1 \leq m$, $1 \leq i_2, j_2 \leq LB_1(G)$, $0 \leq i_3 \leq m - i_1$, $0 \leq j_1 \leq m - i_1 - i_3$ and $0 \leq j_3 \leq m - i_1 - i_2 - j_1$. Here $t_2 = \max\{i_2, j_2\}$.

Under this case we need to merge the two components C_u and C_v containing u and v respectively under the current settings for the two entries of $table(u)$ and $table(v)$. We assume that $i_1 + j_1$ vehicles will be used to service the jobs in the new component. The new component has LB_1 equal to the maximum of that of C_u and C_v, so we update the second dimension of the new entry to be $t_2 = \max\{i_2, j_2\}$. After the merging, the connected components not containing u and v remain unchanged, therefore we set the third dimension to $i_3 + j_3$. Also we add $2 * c(u, v)$ to this entry, since (u, v) now becomes part of the new connected component containing both u and v.

Note that for both cases, we do not check whether an entry $table(u)[i_1][i_2][i_3]$ is feasible or not when updating this entry. This implies that $i_2 + 2 * table(u)[i_1][i_2][i_3]/i_1$ might be larger than λ when processing u. This is because the corresponding connected component containing u might be merged later with a connected component containing $p(u)$. In this case u will be in a larger connected component associated with more, e.g. m', vehicles. Thus the contribution of $table(u)[i_1][i_2][i_3]$ to the lower bound LB of the new component becomes $table(u)[i_1][i_2][i_3]/m'$ which is strictly less than $table(u)[i_1][i_2][i_3]/i_1$. This entry might lead to a feasible final solution, therefore we still need to keep it for later computation.

The above algorithm runs in strongly polynomial time. Recall that $LB_1(P_1) = \max_{u \in V(P_1)} \{r(u) + h(u)\}$, therefore for a vertex u the second dimension of $table(u)$ takes at most $|V|$ distinct values. A particular entry of $table(u)$ can be located by a binary search over these $|V|$ distinct values. The time complexity of this algorithm is dominated by Case 2 when updating $table(u)$ for each vertex u. The algorithm runs in time $O(m^4|V|^3 \log |V|)$.

It is not difficult to see that this algorithm can be used to solve a disguised problem D' of D: Given a real number λ, compute the smallest possible bound $LB(G, m)$ (at least λ) subject to the constraint that, under the corresponding vehicle configuration VC the bound $LB(C, m')$ of each connected component C, not including the root r in $C_{VC}(r)$, is at most λ. Here m' is the number of vehicles used for the customers in C. We will show in the next subsection that a strongly polynomial time algorithm for the optimization problem P' can be obtained by solving D'.

Solving the optimization problem

In this subsection we present a parametric searching algorithm (called MVSP_optimization in the following) for solving our optimization problem by taking advantage of the decision problem D'. The parametric search technique was developed by Megiddo in [72] and [73], and it works as follows. Let λ^* be the optimal solution of an optimization problem P_1. Assume that for P_1 we have a decision problem $D_1(\lambda)$ which is monotone in λ, in the sense that we can decide whether $\lambda < \lambda^*$, $\lambda = \lambda^*$ or $\lambda > \lambda^*$. Assume that we have an algorithm A for the decision problem $D_1(\lambda)$. To solve the optimization problem P_1, Megiddo's idea is to run A generically. It seems somewhat strange to run an algorithm when the input is still unknown. The core in Megiddo's method is to maintain an open interval I where λ^* lies throughout the execution of the algorithm A. More specifically, I is initialized to $(-\infty, +\infty)$, and at each step of running A with unknown input, a critical value t_1 is computed and the concrete version of A is executed with parameter t_1. Therefore after the first step, the interval I is shrunk to either $(-\infty, t_1]$ or $[t_1, +\infty)$. When the generic version of A is terminated, we either find λ^* or an interval with its lower end equal to λ^*.

It is crucial to generate the critical values in the parametric search framework. These critical values in fact discretize the problem P_1 and make it possible to compute P_1 optimally in polynomial time. Intuitively the critical values, or the steps of A, represent all the tests which the optimal solution must pass. On the other hand, if a solution passes all such tests, then it is a candidate of the optimal solution. Note that as long as all the critical values are

tested, the master optimization algorithm needs not necessarily to be the same as A, e.g., in many cases sorting all the critical values suffices for solving the optimization problem.

In the following, we describe a parametric searching algorithm for solving P'. Recall that P' is to compute the smallest possible bound $LB(T, m)$ for a tree T and a given integer m. We first define the discrete events, or the critical values, needed by the parametric search method. Given a subtree T_u of T and a number $0 < m' \leq m$, we define the critical value at u to be the bound $LB(T_u, m')$ under the constraint that m' vehicles are assigned to service the jobs in T_u. Corresponding to this definition, our algorithm runs in the bottom up fashion: the computation of the lower bound $LB(T, m)$ starts from the leaves of the tree, and the optimal value of LB for a subtree T_u will be available after all the vertices in T_u have been processed.

The reason we choose $LB(T_u, m')$ $(0 < m' \leq m)$ to be the critical value at a vertex u is as follows. Let F^* be the optimal forest corresponding to the bound $LB(T, m)$. Then $LB(T, m)$ is determined by a particular connected component C of F^*. Assume u is the root of C and T_u is allocated m' vehicles in total in the optimal solution. Then we can find the optimal solution by applying the algorithm MVSP-decision in Figure 5.16 with the parameter $LB(T_u, m')$. Given the values $LB(T_u, m')$ for every subtree T_u of T and every integer $0 < m' \leq m$, it is easy to see that we can compute $LB(T, m)$ by $m \cdot |V|$ applications of the algorithm MVSP-decision in Figure 5.16.

For a leaf u of T, it is trivial to compute $LB(T_u, m')$ $(0 < m' \leq m)$. In our algorithm, whenever a critical value $LB(T_u, m')$ $(u \in T, 0 < m' \leq m)$ is known, we propagate the test on this value along the path from u to the root r of T. In other words, for every vertex u' on the path from u to r, we apply algorithm MVSP-decision on $T_{u'}$ with m vehicles and the feasibility parameter $LB(T_u, m')$. For an arbitrary vertex u, a critical value $LB(T_u, m')$ $(u \in T, 0 < m' \leq m)$ is computed if and only if for every vertex u' (other than u) in T_u, the test of $LB(T_{u'}, m')$ $(0 < m' \leq m)$ has been taken on T_u. It is easy to see that the optimal solution can be computed after $m|V|^2$ calls to the algorithm MVSP-decision. Therefore the time complexity of the algorithm is $O(m^5|V|^5 \log |V|)$.

We omit the proof for the following Lemma.

Lemma 5.7.2. *The algorithm MVSP_optimization computes the minimum possible bound $LB(T, m)$. The running time of the algorithm is $O(m^5|V|^5 \log |V|)$.*

By Lemma 5.7.1 and 5.7.2 we establish the following theorem.

Theorem 5.7.3. *The algorithm MVSP_optimization can be used to obtain a 3-approximation for the MVSP in trees.*

Using the results in [32], we have

Theorem 5.7.4. *There is an $O(\log n)$-approximation for the MVSP in general graphs.*

Chapter 6

Conclusion

Approximation algorithms are very useful when dealing with NP-hard problems. For some problems even in the simplest form, e.g. the MDCVRP in trees, the optimal solutions demonstrate no orderliness. However many such problems still have some nice hidden structures or properties. These properties can be utilized to design solutions which might not be optimal but have provably bounded costs. An example is the flow conservation property implied by the optimal solutions for the MDCVRP. This property allows us to efficiently compute nice lower bounds for the MDCVRP in tree-like graphs. Solutions with bounded cost can then be constructed based on these lower bounds.

In this thesis we have investigated approximation algorithms for various variants of the CVRP. Various approximation techniques are also examined or provided for these problems. We have proposed a come-back rule for two variants of the CVRPPD, namely the k-delivery TSP and the Capacitated Dial-a-Ride Problem. This rule leads to the half-load strategy which is shown to be useful for the k-delivery TSP in trees. For the BWTSP, we have examined the application of matching and König's theorem on approximation algorithms. For the p-constrained Cycle Covering Problems, we have presented a combinatorial analysis of a minimum spanning tree-based algorithm and have shown that it is effective for dealing with the cardinality constraint imposed on network design problems with downwards monotone functions. For the MDCVRP and the MVSP, we have proposed a different way of using dynamic programming in designing approximation algorithms for the VRP. The results obtained in this thesis are summarized in Table 6.1.

Table 6.1: Results obtained.

Problem	Previous	Our results	Techniques used
k-delivery TSP on paths	$O(n^2/k)$ [81]	optimal	come-back rule
k-delivery TSP in trees	2 [68]	$\frac{5}{3}$	come-back rule
Dial-a-Ride on paths	3 [61]	2.5	come-back rule
BWTSP		$(4-\frac{3}{2k})$	matching, König's theorem
BWTSP(special case)		$(4-\frac{15}{8k})$	matching, GW-algorithm
CCPBL	$4(p_L+1)$ [70]	4	GW-algorithm
pCCCP		2	combinatorial analysis
MDCVRP on paths		$O(n^3)$ time, exact	dynamic programming
MDCVRP in trees		2	the general flow bound
MDCVRP partial k-trees		6	the general flow bound
MDCVRP general		$\min\{k, O(\log n)\}$,	metric embedding
VDP general	$2k-1$ [60]	$\min\{2k-1, O(\log n)\}$	metric embedding
MVSP in trees		3	dynamic programming
MVSP general		$O(\log n)$	metric embedding

There is still interesting research work remaining. One direction for future research is to find approximation algorithms for (NLP) (finding the smallest *general flow* bound) with bounds better than $O(\log n)$. The known approximation techniques, e.g. the GW-algorithm [40] and the iterative rounding algorithm [48], do not seem to apply.

Another direction for future research is to investigate the possibility of extending our way of using dynamic programming in designing approximation algorithms for the VRP, to improve the approximation ratios for the MVSP and the VRP with time windows (VRPTW) in general graphs. In the VRPTW, each customer u is associated with a full time window $[r_u, d_u]$, and the vehicle must arrive after time r_u and before time d_u to service the customer. An $O(\log^2 n)$-approximation is given in [7] for the VRPTW in undirected complete graphs. It would be interesting to improve this approximation ratio or reduce the time complexity of the algorithm in [7].

Chapter 7

Appendix

7.1 Pseudo code of the full-load algorithm

The following figure shows the full-load algorithm in [81] for the k-delivery TSP in trees.

There is no planning phase in the full-load algorithm, instead the lists L_+ and L_- contain all real tree edges. Given the lists L_+ and L_-, in the full_load_deliver procedure in Figure 7.3, the vehicle picks up vertices consecutively from the positive edges of L_+ until its load is exactly k or all the pickup vertices of the positive edges are served. Then the vehicle delivers its load to the negative edges also consecutively, until all of its load is consumed.

The pickup and delivery functions of the full-load algorithm follow much in the same way as in the half-load algorithm, with the major difference that there's no need to push down the L_+ list when delivering. Recall that under the full-load strategy, the vehicle continues to pick up or deliver vertices if possible. Therefore when the vehicle decides to pick up (deliver) some vertices from (to) an edge $e = (p(u), u)$, we know immediately the amount of product the vehicle should pick up (deliver) from e. This makes the full-load algorithm much simpler to implement than the half-load algorithm.

Recall that the half-load algorithm contains two phases. In the first planning phase, the algorithm works on the original graph and prepares pseudo edges for the second phase, where the actual route is generated. As in the full-load algorithm, the actual route generating phase of the half-load algorithm works recursively in a top-down fashion. However, unlike the full-load algorithm, in the half-load algorithm positive pseudo edges for a particular positive tree edge $e = (p(u), u)$ can not be determined when e is firstly visited in this phase. All the information we have about the traversals on e is only that the vehicle takes more than $\frac{k}{2}$

Procedure full_load_deliver(α, L_+, L_-)
Input: two lists L_+ and L_- containing the tree edges with their loads; α denotes the current load of the vehicle
Output: a tour satisfies the capacity constraint

```
1  while L+ ≠ ∅ and (L− ≠ ∅ or α < k) do
2        Comment: pick up the vehicle load consecutively from edges in L+
3        if L+ ≠ ∅ then
4              pickup(α, L+)
5        endif
6
7        Comment: deliver the vehicle load consecutively to edges in L−
8        if L− ≠ ∅ then
9              deliver(α, L−)
10       endif
11 endwhile
12
13 return
```

Figure 7.1: A procedure for the full-load algorithm.

vertices during most of the traversals. Thus unless the positive pseudo edges from u to its positive children have been built, we have no knowledge about how many traversals would the vehicle make on this edge, and we also don't know how much load would the vehicle take during each traversal. Building pseudo edges and further handling of the pseudo edges to generate the actual route are the major difference in the implementation aspect between the full-load and the half-load algorithms.

The pseudo codes of the pickup and deliver procedures for the full-load algorithm are listed in Figures 7.2 and 7.3 respectively.

7.2 More Explanations of the GW-algorithm

The pseudo code of the GW algorithm for proper functions is listed in Figure 7.4.

In the following, we give some examples of proper functions and the problems they solve:

1. Steiner Tree Problem: Given a set $N \subseteq V$ of terminal nodes, find a minimum-cost tree that connects all nodes in N.

   $f(S) = 1$ if and only if $\emptyset \neq S \cap N \neq N$, and 0 otherwise.

2. Point-to-point Connection Problem: Given a set $C = \{c_1, \cdots, c_p\}$ of source nodes, and a set $D = \{d_1, \cdots, d_p\}$ of destination nodes, find a minimum cost forest such that each connected component of the forest contains the same number of source nodes and destination nodes.

   $f(S) = 1$ if and only if $|S \cap C| \neq |S \cap D|$, and 0 otherwise.

3. T-join Problem: Given an even subset T of vertices, find a minimum-cost set of edges that has odd degree at vertices in T and even degree at vertices not in T.

   $f(S) = 1$ if and only if $|S \cap T|$ is odd, and 0 otherwise.

An example illustrating a run of the GW algorithm is given in Figure 7.5. This example is from [19].

In this example, a minimum cost Steiner tree is needed to connect three terminal vertices a, b and f. The underlying graph and three iterations of the GW algorithm are illustrated in Figure 7.5. The final solution F' of the GW algorithm is indicated in thick lines. The active and inactive components of Γ are represented by red and black dashed circles respectively.

Procedure pickup(α, L'_+)
Input: a list L'_+ of positive tree edges; α denotes the amount of product in the vehicle
Output: a route of picking up the vertices

```
1  e = (p(u), u) ← the head of L'+
2  if u is a leaf then
3        pick up u and update α
4        return
5  endif
6
7  if α + n(e) ≤ k then
8        L'+ ← L'+ - {e}
9  else
10       n(e) ← n(e) − k + α
11 endif
12
13 Comment: L+ and L− contain edges from u to its children
14 full_load_deliver(α, L+, L−)
15
16 return
```

Figure 7.2: The pickup procedure for the full-load algorithm.

Procedure deliver(α, L'_-)
Input: a list L'_- of negative tree edges; α denotes the amount of product in the vehicle
Output: a route of delivering the vertices

```
1  e = (p(u), u) ← the head of L'_-
2  if u is a leaf then
3        service u and update α
4        return
5  endif
6
7  if α ≥ n(e) then
8        L'_- ← L'_- - {e}
9  else
10       n(e) ← n(e) − α
11 endif
12
13 Comment: L_+ and L_- contain positive and negative edges from u to its children
14 full_load_deliver(α, L_+, L_-)
15
16 return
```

Figure 7.3: The deliver procedure for the full-load algorithm.

Algorithm GW(G, f)
Input:An undirected graph $G = (V, E)$, edge costs $c_e \geq 0$, and a proper function f
Output:A forest F' and a value LB

1 F=$\emptyset$

2 *Comment: Implicitly set* $y_S \leftarrow 0$ *for all* $S \subset V$

3 $LB \leftarrow 0$

4 $\Gamma = \{\{v\} | v \in V\}$

5 For each $v \in V$

6 $\quad d(v) \leftarrow 0$

7 While $\exists C \in \Gamma : f(C) = 1$

8 $\quad$ Find edge $e = (i, j)$, $i \in C_p \in \Gamma, j \in C_q \in \Gamma, C_p \neq C_q$ that minimizes

9 $\quad \epsilon = \frac{c_e - d(i) - d(j)}{f(C_p) + f(C_q)}$

10 $\quad F = F \cup \{e\}$

11 $\quad$ *Comment: Implicitly set* $y_C \leftarrow y_C + \epsilon \cdot f(C_r)$

12 $\quad$ For all $v \in C_r \in \Gamma$ do $d(v) = d(v) + \epsilon \cdot f(C_r)$

13 $\quad LB \leftarrow LB + \epsilon \sum_{C \in \Gamma} f(C)$

14 $\quad \Gamma = (\Gamma - \{C_p, C_q\}) \cup \{C_p \cup C_q\}$

15 endwhile

16

17 *Comment: the deleting phase*

18 $F' \leftarrow \{e \in F : f(N) = 1$ *for some connected component* N *of* $(V, F - \{e\})\}$

Figure 7.4: The GW algorithm.

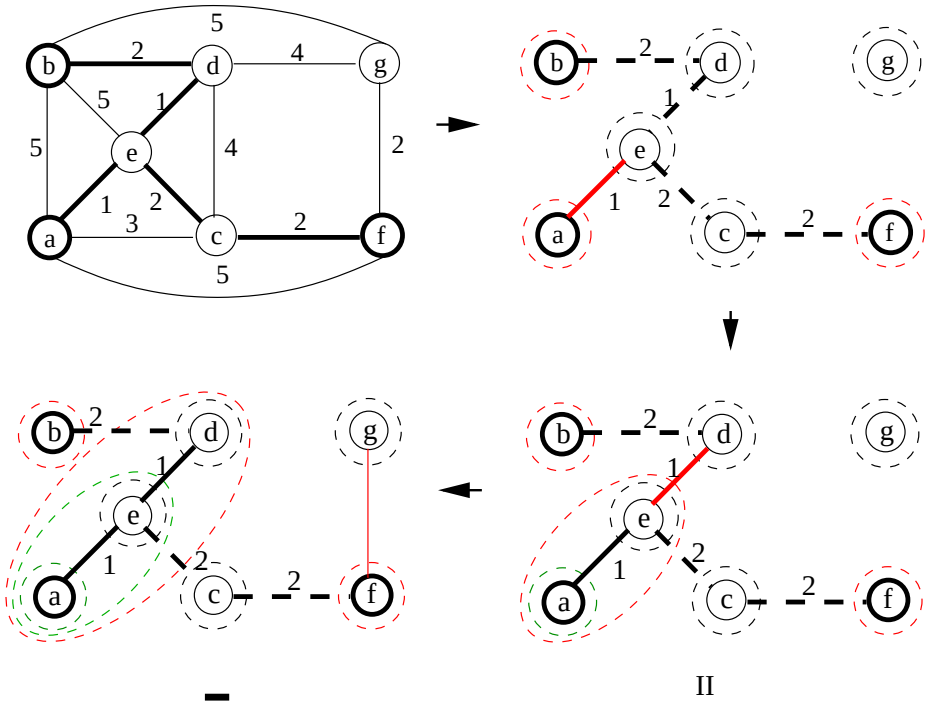

Figure 7.5: A run of the GW algorithm for an instance of the Steiner tree problem.

The edge selected by the GW algorithm in each iteration is colored red in Figure 7.5. We explain the second iteration of the example. At the beginning of the second iteration, the list Γ contains the components $\{a, e\}, \{b\}, \{c\}, \{d\}, \{f\}, \{g\}$. Among all the components, $\{a\}$, $\{a, e\}$, $\{b\}$, $\{f\}$ are active and their dual variables have values 1, 0, 1, 1 respectively. Other components are inactive and the values of their dual variables remain 0. Note that although $\{a\}$ is not in Γ, its dual variable would still contribute to the computation of ϵ for this iteration. This is because $y_{\{a\}}$ has value 1 (set during the first iteration), and the edge (a, b) is on the cut of $\{a\}$ to $V - \{a\}$. In order to get the edge (a, b) tight, we need to increase $y_{\{b\}}$ and $y_{\{a,e\}}$ by $\frac{c_e - y_{\{a\}} - y_{\{b\}} - y_{\{a,e\}}}{2} = \frac{5-1-1-0}{2} = 1.5$. This edge is not chosen in this iteration, since the edge (d, e) is already tight when $y_{\{a,e\}}$ is increased by 1. Also note that once an active component C is removed from the list Γ, its dual variable y_C would remain unchanged starting from the next iteration. Thus the *minimum violation set rule* and the *uniform increase rule*, together with the merging operation, ensure that the dual solution is feasible.

Since there are exponential number of subsets, it is not feasible for the algorithm to explicitly maintain the dual variables. Instead, the values of the dual variables are remembered implicitly by the variable $d(v)$ for each vertex v. Recall that in the algorithm, the dual variables are referenced only when computing the minimum ϵ for the current iteration. Consider an edge $e = (u, v)$, where u and v are in two different components C_u and C_v of Γ respectively. The algorithm needs to compute $c_e - \sum_{S:e\in\delta(S)} y_S$ to obtain the minimum ϵ for the current iteration. Instead of increasing the dual variable y_S by ϵ, the algorithm chooses to increment the d variables of the vertices in S by ϵ. It is easy to see that $c_e - d(u) - d(v)$ is equal to $c_e - \sum_{S:e\in\delta(S)} y_S$, since for a set S such that $e \in \delta(S)$, either u or v must be in S. An example is that in Figure 7.5, in the second iteration, when $y_{\{b\}}$ and $y_{\{a,e\}}$ are increased by 1, d_a would be $y_{\{a\}} + y_{\{a,e\}} = 2$ since vertex a is inside $\{a\}$ and $\{a, e\}$. Similarly d_b equals to $y_{\{b\}} = 2$. In order to get the edge (a, b) tight during the third iteration, $y_{\{b\}}$ and $y_{\{a,d,e\}}$ must be increased simultaneously by $\frac{c_{ab} - y_{\{b\}} - y_{\{a\}} - y_{\{a,e\}} - y_{\{a,d,e\}}}{2} = \frac{c_{ab} - d_a - d_b}{2} = 0.5$ (note that $y_{\{a,d,e\}} = 0$). The difficulty of storing the dual variables is therefore circumvented.

In the reverse deleting phase, an edge is not deleted only if its deletion will cause a connected component of the forest to be inactive. This also means that we can delete an edge from F as long as its deletion still makes the forest feasible. In [40], the deletion phase is implemented as follows. A component C of F is rooted at an arbitrary vertex, and in a bottom up manner we compute the f value for each vertex; an edge joining a vertex to

its parent is discarded if the f value of the set of vertices in its subtree is 0. However, for some problems, the order of deleting these edges is crucial for the approximation bound. For example, for the uncrossable functions, the edges must be deleted in the reverse order of their selection to F.

Bibliography

[1] A. Agrawal, P. Klein, R. Ravi, When trees collide: An approximation algorithm for the generalized Steiner problem on networks. *Proceedings of the 23rd annual ACM-SIAM symposium on Theory of Computing*, 134-144, 1991.

[2] S.Anily, J.Bramel, The swapping problem on a line. *SIAM Journal on Computing.*, 29(1):327-335, 1999.

[3] S.Anily, J.Bramel, Approximation algorithms for the capacitated traveling salesman problem with pickups and pelieries. *Naval Research Logistics*, 46:654-670, 1999.

[4] M.J.Atallah, S.R.Kosaraju, Efficient solutions to some transportation problems with applications to minimizing robot arm travel. *SIAM Journal on Computing*, 17(5):849-869, 1988.

[5] John E. Augustine, Steven Seiden, Linear time approximation schemes for vehicle scheduling problems. *Theoretical Computer Science*, 324(2-3):147-160, 2004.

[6] Y. Bartal, Probablistic approximations of metric space and its algorithmic applications. *Proceedings of the 37th Foundation of Computer Science*, 184-193, 1996.

[7] Nikhil Bansal, Avrim Blum, Shuchi Chawla, Adam Meyerson, Approximation algorithms for deadline-TSP and vehicle routing with time-windows. *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, 166-174, 2004.

[8] C. Basnet, L. R. Foulds, J. M. Wilson, Heuristics for vehicle routing on tree-like networks. *Journal of the Operational Research Society,* 50:627-635, 1999.

[9] B. K. Bhattacharya, D. Gaur, Q. Shi, Y. Hu, A Rule for the capacitated vehicle routing problem with pickups and deliveries in trees. *manuscript*, 2008.

[10] B. K. Bhattacharya, Y. Hu, A. Kononov, Approximation algorithms for the black and white traveling salesman problem. *The 13th Annual International Computing and Combinatorics Conference*, 559-567, 2007.

[11] B. K. Bhattacharya, Y. Hu, Dynamic programming based approximation algorithms for vehicle routing problems. *manuscript*, 2008.

[12] M. Bourgeois, G. Laporte, F. Samet, Heuristics for the black and white traveling salesman problem. *Computers and Operations Research*, 30:75-85, 2003.

[13] Z. Cai, G. Lin, G. Xue, Improved approximation algorithms for the capacitated multicast routing problem. *The 11th Annual International Computing and Combinatorics Conference*, 136-145, 2005.

[14] P. Chalasani, R. Motwani, Approximating capacitated routing and delivery problems. *SIAM Journal on Computing*, 28(6):2133-2149, 1999.

[15] M. Charikar, S. Khuller, B. Raghavachari, Algorithms for capacitated vehicle routing. *SIAM Journal on Computing*, 31(3):665-682, 2002.

[16] M. Charikar, B. Raghavachari, The finite capacity Dial-a-Ride problem. *Proceedings of the 39th Foundation of Computer Science*, 458-467, 1998.

[17] B. Chazelle, A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47, 2000, 1028-1047.

[18] C. Chekuri, A. Kumar, Maximum coverage problem with group budget consrtaints and applications. *The 7th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems,* LNCS 3122:72-83, 2004.

[19] J. Cheriyan, R. Ravi, Approximation algorithms for network problems. *Lecture Notes*, http://www.math.uwaterloo.ca/ jcheriya/lecnotes.html

[20] N. Christofides, The traveling salesman problem. *Combinatorial Optimization, (eds) N. Christofides, A. Mingozzi, P. Toth and C. Sandi*, 315-318, 1979.

[21] J. Chuzhoy, A. Gupta, J. S. Naor, A. Sinha, On the approximability of some network design problems. *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, 943-951, 2005.

[22] R. Cole, J. Hopcroft, On edge coloring bipartite graphs. *SIAM Journal on Computing,* 11:540-546, 1982.

[23] R. Cole, K. Ost, S. Schirra, Edge-coloring bipartite multigraphs in $O(E \log D)$ time. *Combinatorica,* 21:5-12, 2001.

[24] S. Cosares, D.N. Deutsch, I. Saniee, O.J. Wasem, SONET Toolkit: A decision support system for designing robust and cost effective fibre-optic networks. *Interfaces* 25:20-40, 1995.

[25] G. Cornuejols, W. Pulleyblank, A matching problem with side constraints, *Discrete Mathematics*, 29:135-159, 1980.

[26] G. B. Dantzig, R. H. Ramser, The truck dispatching problem. *Management Science*, 6:80-91, 1959.

[27] G. B. Dantzig, D. R. Fulkerson, S. Johnson, Solution of a large scale traveling salesman problem. *Operations Research*, 2:393-410, 1954.

[28] D. Dinitz, The solution of two assignment problems. *In Russian; Studies in Discrete Optimization* (A.A. Fridman, ed.), Nauka, Moscow, 333-348, 1976.

[29] J. Dunavy, A. Akuoko-Asibey, R. Masse, D. Pilon, An analysis of the transportation industry in 2005. *Statistics Canada*, Catalogue NO. 11-621-MIE-NO. 044,2005.

[30] J. Edmonds, Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449-467, 1965.

[31] J. Edmonds, E. L. Johnson, Matching: a well-solved class of integer linear programs. *in Combinatorial Structures and Their Applications*, R. Guy, H. Hanini, N. Sauer, and J. Schonheim, eds., Gorden and Breach, New York, pp:89-92, 1970.

[32] J. Fakcharoenphol, S. Rao, K. Talwar, A tight bound on approximating arbitrary metrics by tree metics. *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 448-455, 2003.

[33] G. N. Frederickson, D. J. Guan, Nonpreemptive ensemble motion planning on a tree. *Journal of Algorithms*, 15:29-60, 1993.

[34] G. N. Frederickson, M. S. Hecht, C. E. Kim, Approximation algorithms for some routing problems. *SIAM Journal on Computing*, 31(3):178-193, 1978.

[35] H. N. Gabow, S. Pettie, The dynamic vertex minimum problem and its application to clustering-type approximation algorithms. *The 8th Scandinavian Workshop on Algorithm Theory*, 190-199, 2002.

[36] D.R. Gaur, A. Gupta, R. Krishnamurti, A 5/3-approximation algorithm for scheduling vehicles on a path with release and handling times, *Information Processing Letters*, 86(2):87-91, 2003.

[37] G. M. Guisewite, P. M. Pardalos, Minimum concave-cost network flow problems: applications, complexity, and algorithms, *Annals of Operations Research*, 25(1-4):75-100, 1990.

[38] G. Ghiani, G. Laporte, F. Semet, The black and white traveling salesman problem. *Operations Research*, 54:366-378, 2006.

[39] P. C. Gilmore, E. L. Lawler, D. B. Shmoys, Well-solved special cases. In Eugene L. Lawler, Jan Karel Lenstra, Alexander H. G. Rinnooy, Kan and David B. Shmoys, editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, John Wiley & Sons, 1985, pp:87-143.

[40] M. X. Goemans, D. P. Williamson, A general approximation technique for constrained forest problems. *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, 307-316, 1992.

[41] M. X. Goemans, D. P. Williamson, Approximating minimum-cost graph problems with spanning tree edges. *Operations Research Letters*, 16:183-194, 1994.

[42] M. X. Goemans, D. P. Williamson, The primal-dual method for approximation algorithms and its application to network design problems. *Approximation Algorithms, D. Hochbaum, ed.*

[43] R P Graves, Life of Sir William Rowan Hamilton. , 1975.

[44] P. Hell, D. Kirkpatrick, J. Kratochvil, I. Kritz, On restricted two-factors. *SIAM Journal on Discrete Mathematics*, 1(4):472-484, 1988.

[45] I. V. Hicks, A. M. C. A. Koster, E. Kolotoglu Branch and tree decomposition techniques for discrete optimization. *Tutorials in Operations Research*, INFORMS, New Orleans, 2005.

[46] C. Imielinska, B. Kalantari, L. Khachiyan, A greedy heuristic for a minimum-weight forest problem. *Operations Research Letters*, 14:65-71, 1993.

[47] C. Imielinska, B. Kalantari, L. Khachiyan, A greedy heuristic for a minimum-weight forest problem. *Operations Research Letters*, 14:65-71, 1993.

[48] K. Jain, A factor 2 Approximation Algorithm for the Generalized Steiner Network Problem. *Combinatorica*, 21(1):39-60, 2001.

[49] R. Jothi, B. Raghavachari, Approximation algorithms for the capacitated minimum spanning tree problem and its variants in network design. *ACM Transactions on Algorithms (TALG)*, 1(2):265-282, 2005.

[50] R. M. Karp, Complexity of computation. *Proceedings AMS-SIAM Symposia in Applied Mathematics*, Vol. VII, 1974.

[51] Y. Karuno, H. Nagamochi, A polynomial time approximation scheme for the multi-vehicle scheduling problem on a path with release and handling times. *Proceedings of the 12th International Symposium on Algorithms and Computation*, 36-47, 2001.

[52] Y. Karuno, H. Nagamochi, A 2-approximation algorithm for the multi-vehicle scheduling problem on a path with release and handling times. *Proceedings of the 9th Annual European Symposium on Algorithms*, 218-229, 2001.

[53] Y. Karuno, H. Nagamochi, T. Ibaraki, Better approximation ratios for the single-vehicle scheduling problems on line-shaped networks, *Networks*, 39(4):203-209, 2002.

[54] Y. Karuno, H. Nagamochi, T. Ibaraki, Vehicle scheduling on a tree with release and handling time, *Annals of Operations Research*, 69:193-207, 1997.

[55] Y. Karuno, H. Nagamochi, An approximability result of the multi-vehicle scheduling problem on a path with release and handling times, *Theoretical Computer Science*, 312:267-280, 2004.

[56] Y. Karuno, H. Nagamochi, 2-Approximation algorithms for the multi-vehicle scheduling problem on a path with release and handling times, *Discrete Applied Mathematics*, 129:433-447, 2003.

[57] King, G. F., C. F. Mast, Excess travel: causes, extent, and consequences. *Transportation Research Record*, 1111:126-134, 1997.

[58] D. König, Über Graphen und ihre Anwendungen. *Math. Annalen*, 77:453-465, 1916.

[59] N. Korula, C. Chekuri, Approximation algorithms for orienteering with timewindows. *manuscript*, Sept., 2007.

[60] S. O. Krumke, S. Saliba, T. Vredeveld, S. Westphal, Approximation algorithms for a vehicle routing problem. *Mathematical Methods of Operations Research*, 68(2):333-359, 2008.

[61] S. Krumke, J. Rambau, S. Weider, An approximation algorithm for the nonpreemptive capacitated dial-a-ride problem. *Preprint 00-53, Konrad-Zuse-Zentrum fr Informationstechnik Berlin*, 2000.

[62] J. B. Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, Vol 7, No. 1, pp. 48C50, Feb 1956.

[63] M. Labbé, G. Laporte, H. Mercure, Capacitated vehicle routing on trees. *SIAM Operations research*, 39:616-622, 1991.

[64] M. Laszlo, S. Mukherjee, An approximation algorithm for network design problems with downwards-monotone demand functions. *Optimization letters*, 2:171-175, 2008.

[65] G. Laporte, Location-routing problems. In B.L. Golden and A. A. Assad, editors, *Vehicle routing: methods and studies*, 163-197, Amsterdam, 1988.

[66] E. L. Lawler, Combinatorial optimization: networks and matroids. Holt, Rinehart and Winston, New York, 1976.

[67] Chung-lun Li, David Simchi-levi, Worst-case analysis of heuristics for multidepot capacitated vehicle routing problems. *ORSA Journal on Computing* 2(1):64-73, Winter 1990.

[68] A. Lim, F. Wang, Z. Xu, The capacitated traveling salesman problem with pick-ups and deliveries on a tree. *The 16th International Symposium on Algorithms and Computation*, LNCS, 3827:1061-1070, 2005.

[69] V. Mak, N. Boland, Heuristic approaches to the asymmetric traveling salesman problem with replenishment arcs. *International Transactions in Operations Research*, 7:431-437, 2000.

[70] B. Manthey, Minimum weight cycle covers and their approximability. *The Proceedings of the 33rd Workshop on Graph-Theoretic Concepts in Computer Science*, 178-189, 2007.

[71] B. Manthey, On approximating retricted cycle covers. *SIAM Journal on Computing,*, 38(1):181-206, 2008.

[72] N. Megiddo Combinatorial optimization with rational objective functions. *Proceedings of the tenth annual ACM symposium on Theory of computing*, 1-12, 1978.

[73] N. Megiddo Applying parallel computation algorithms in the design of serial algorithms. *Journal of ACM*, 30(4):852-865, 1983.

[74] K. Menger, Reminiscences of the vienna circle and the mathematical colloquium. *Dortmund*, 1994.

[75] H. Nagamochi, K. Mochizuki, T. Ibaraki, Complexity of the single vehicle scheduling problem on graphs, *Information Systems and Operations Research*, 35(4):256-276, 1997.

[76] H. Psaraftis, M. Solomon, T. Magnanti, T.-U. Kim, Routing and scheduling on a shoreline with release times, *Management Science*, 36:212-223, 1990.

[77] A. I. Serdyukov, An algorithm with an estimate for the traveling salesman problem of the maximum. *Upravlyaemye Sistemy*, 25: 80C86, 1984 (in Russian).

[78] K. T. Talluri, The four-day aircraft maintenance routing problem. *Transportation Science*, 32:43-53, 1998.

[79] J. Tsitsiklis, Special cases of traveling salesman and repairman problems with time windows, *Networks*, 22:263-282, 1992.

[80] D. P. Williamson, M. X. Goemans, M. Mihail, V. V. Vazirani, A primal-dual approximation algorithm for generalized Steiner network problems. *Proceedings of the 25th annual ACM-SIAM symposium on Discrete algorithms*, 708-717, 1993.

[81] F. Wang, A. Lim, Z. Xu, The one-commodity pickup and delivery traveling salesman problem on a path or a tree. *Networks*, 48(1):24-35, 2006.

[82] N. Robertson, P. D. Seymour, Graph Minors. XIII. The disjoint paths problem, *Journal of Combinatorial Theory, Series B*, 52:153-190, 1990.

[83] N. Robertson, P. D. Seymour, Graph Minors. X. Obstructions to tree-decomposition, *Journal of Combinatorial Theory, Series B*, 63:65-110, 1995.

[84] P. Toth, D. Vigo, editors, The vehicle routing problem. *SIAM Monographs on Discrete Mathematics and Applications*, 9, SIAM, Phaladelphia, PA, 2002.

[85] V. V. Vazirani, Approximation algorithms. *Berlin: Springer*, ISBN 3540653678, 2003.

[86] O. Vornberger, Complexity of path problems in graphs, PhD thesis, *Universitat-GH-Paderborn*, 1979.

[87] O.J. Wasem, An algorithm for designing rings in survivable fibre networks. *IEEE Transactions on Reliability*, 40:428-432, 1991.