University of Alberta

Improved Algorithms for Multicast Routing and Binary Fingerprint Vector Clustering

by

Zhipeng Cai 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2004

Your file Votre référence
ISBN: 0-612-95715-2
Our file Notre référence
ISBN: 0-612-95715-2

Giving one a fish is only for a meal, but teaching one to fish can benefit one for a life.
— Chinese Proverb

Acknowledgements

Computer science is where I will deeply dedicate myself, where my happiness lies, and where my life will scintillate. I would like to thank all the people who help me in such a spectacular field.

First and the foremost, I would like to thank my advisor, Guohui Lin, for his patient instruction. My discussion with him always leads to new insights in depth and width. He opens one black box after another for me and gives me theoretical and practical ability in problem formation and solving. I have learned from him not only the ability of doing research, but an attitude for doing research.

Next, I would like to thank my family for their support in all the phases of my life. Thank my lovely wife, Yingshu Li, for her love, her help in my research, her encouragement and her cares. Without her, this thesis would not have been possible.

Also, thanks to my colleagues Xiang Wan and Gang Wu for the creative discussions. I really enjoy the many chats with them that have enriched my view of algorithms.

Finally, I felicitate and cherish the opportunity to study in my department. Here, I find myself studying in a distinguished environment encompassed by an academic atmosphere where originality is promoted and individual potential is tapped.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and Organization

Computational Complexity is one part of the study in the *Theory of Computation* dealing with the resources required during the computation to solve a given problem. The most commonly examined resources are time and space [21]. While the given computational problem can be of arbitrary form, as long as it specifies the input and the desired output, there are two categories of problems which are of practical interest in Computer Science. One category is optimization problems, where the input includes an objective function and the output is the best of all possible solutions. The other category is decision problems, where the output is either "yes" or "no". Considering decision problems using two computational models, namely, the *deterministic Turing machine* and the *non-deterministic Turing machine*, two classical complexity classes can be defined. The class P consists of all the decision problems that can be solved on a deterministic Turing machine in an amount of time that is polynomial in the size of the input. The class NP consists of all the decision problems whose positive solutions can be verified, given the right information, on a deterministic Turing machine in polynomial time; or equivalently, it consists of all the decision problems whose solutions can be found in polynomial time on a non-deterministic Turing machine [21].

For a decision problem Π in class NP, if solving it in polynomial time means that every problem in class NP can be solved in polynomial time, then Π is a hardest problem in class NP. A hardest problem in class NP is also called an NP-complete problem. A well-known, NP-complete problem first proven by Cook is the *Satisfia-*

bility problem [2, 25].

In general, proof of NP-completeness can be divided into two phases, one of which is to show that the decision problem belongs to class NP and the other is to show that it is a hardest problem. Failing in the first phase gives another notion of "hardness" — NP-hard. A computational problem is *non-deterministic polynomial-time hard* (NP-hard) if an algorithm for solving it can be translated, for any decision problem Π in class NP, into an algorithm in polynomial time to solve Π. Clearly, an NP-hard problem is not easier than any NP-complete problem [12, 31, 21].

For an optimization problem, sometimes we are able to transform it into a decision problem via adding a parameter associated with the objective function. For this reason, an algorithm solving the optimization version implies an algorithm for solving the decision version, and vice versa. Therefore, if it happens that the decision version is NP-complete, then the optimization version is NP-hard, as an optimization problem doesn't belong to class NP. Our real computational problems in the world are usually this kind of optimization problems, and it occurs quite often that these problems are NP-hard. In this thesis, two such problems are considered, which are the *Multicast Routing* problem and the *Binary Fingerprint Clustering* problem, to be detailed.

The NP-hardness of an optimization problem Π implies that, if $P \neq NP$, an optimal solution of Π cannot be obtained in polynomial time. Depending on practical needs, usually there are two ways to approach the problem. If time is one of the key considerations while non-optimal solutions are allowed, then heuristics and approximation algorithms which run in polynomial time are suitable; on the other hand, if time is not really an issue but the quality of the solutions is the first consideration, then exact algorithms, which hopefully run fast although still in exponential time in the worst case, are desired.

We note that the difference between heuristics and approximation algorithms is usually characterized as follow: approximation algorithms provide a certain level of performance guarantee (in the worst case) while heuristics don't. In the next two sections, we will provide some basic notions used in the studies of approximation algorithms and exact algorithms, respectively.

1.1 Approximation Algorithms

As stated above, for those NP-hard optimization problems in which fast near-optimal solutions are required, design and analysis of approximation algorithms come to play. Obviously, two key ingredients of an approximation algorithm are polynomial running time and guaranteed performance in the worst case. One additional advantage of designing approximation algorithms is that, often we do not need extra assumptions about inputs. Nonetheless, we should note that not every NP-hard optimization problem admits good approximation algorithms. In fact, for some NP-hard optimization problems, designing a "good" approximation algorithm is itself a hard problem. But that falls into the study of *inapproximability* which we will not get into in this thesis. In the literature, there are various existing general techniques for designing approximation algorithms, such as linear programming and rounding, the primal-dual method, and greedy method.

Definition 1.1.1 [2] *An algorithm $\mathcal{A}$ is an α-approximation algorithm for an optimization problem Π containing a minimization (maximization) objective function, if $\mathcal{A}$ runs in polynomial time and always produces a solution that is within a factor of $\alpha > 1$ (< 1, respectively) to the optimal solution.*

In the first part of this thesis, we will concentrate on *Multicast Routing* problems. One version of the routing problem is the so-called *Multicast k-Path Routing* (kMPR), to be detailed. In kMPR, the underlying communication network is modeled as an edge-weighted complete graph $G(s, V, D)$, where s is the source node, D is the destination node set and V is the set of all the nodes in the network including those nodes that can only be used as intermediate nodes (called *Steiner* nodes). During the routing, a message is sent out from the source node s along a routing path meaning that only those destination nodes on the path can receive the message. Furthermore, the number of such receiving destination nodes is not greater than k, a prespecified routing parameter. The goal of the routing is to partition the destination node set D into a collection of subsets such that each subset of destination nodes can be arranged on a routing path, and such that the total routing cost is minimized, which is measured as the sum of the weights of the edges on the routing paths.

For every instance of the kMPR problem, supposing its optimal solution is W^*, an approximation algorithm proposed in [18] guarantees to return a routing scheme with cost $W \leq 4W^*$. By Definition 1.1.1, this algorithm is a 4-approximation, which in fact is the previous best. We design later a 3-approximation for the kMPR problem.

Another version of the routing problem is the so-called *Multicast k-Tree Routing* (kMTR), again to be detailed. In the kMTR problem, the routing is along a tree rooted at the source node and transmission nodes have broadcasting capability. The previous best approximation algorithm for the kMTR problem has a performance guarantee of $2.4 + \rho$ [22]. We introduce some new design techniques to achieve a $(2+\rho)$-approximation, where ρ is the best approximation ratio for the *Metric Steiner Tree Problem* [27] (which was about 1.55 at the writing of this thesis).

1.2 Exact Algorithms

The Multicast Routing problem is modeled from real world applications, such as streaming continuous media, where a fast solution is required as usually there are a huge number of such routing requests in a short amount of time. There are other problems where better solutions are preferred instead of running time, and sometimes the best solutions are desired. One such problem is the *Assignment Clustering Problem* abstracted out of DNA micro-array analysis, to cluster a set of binarized oligonucleotide fingerprint vectors. The input to the problem is a set of n vectors of dimension m, and each vector entry takes a value of 1, 0 or unknown value N. The goal is to assign every N either a 1 or a 0 such that in the set of resolved vectors which contain no N's the number of distinct vectors is minimized. When vectors contain more than two N-entries, the clustering problem is hard. Nonetheless, optimal solutions implying the least clusters would help save a lot of experimental cost and thus are desired. Previous work was mostly involved in the design of better approximations and heuristics [18, 22]. In the second part of the thesis, we apply the A* search algorithm [21], borrowed from the Artificial Intelligence community, to the clustering problem. We take advantage of the previously designed approximations and heuristics and design some new heuristics to speed up the search.

The A* search algorithm guarantees an optimal solution to every input and thus its effectiveness is measured by its running time. We test the A* search algorithm on existing datasets and also generate a number of random datasets for testing. The experimental results, as well as our discussions, are detailed in the second part of the thesis.

1.3 Organization

The remainder of the thesis is organized as follows. In the first part, two versions of the Multicast Routing problems are introduced and studied. Specifically, Chapter 2 introduces the Multicast Routing problem and its various versions in detail. Chapter 3 reviews the previous best 4-approximation algorithm for kMPR and its key design ideas, explores our new techniques, and describes our 3-approximation algorithm. The existing approximation algorithms for the kMTR problem are reviewed in Chapter 4, where we explore some new techniques to design the $(2+\rho)$-approximation algorithm. Chapter 5 summarizes our contributions in this study and points out some promising future work.

The second part of the thesis deals with the Assignment Clustering problem, which is introduced in detail in Chapter 6. Chapter 7 reviews existing approximation algorithms and heuristics and describes our A* search algorithm. We also describe in detail the heuristic evaluation functions used in the search. The experimental results are included and discussed in Chapter 8. Chapter 9 concludes the study with some future work.

Part I

Multicast Routing

Chapter 2

Introduction to Multicast Routing

There are three communication methods in networks: unicast, broadcast and multicast as shown in Figure 2.1 [29]. In unicast, a source node sends one copy of a message packet to a specified destination. In multicast is the one where a source node sends one copy of message packet to all members of a multicast group. If a source node sends one copy of message packet to all the other nodes in the network, then it is called broadcast. We focus on multicast routing in this dissertation.

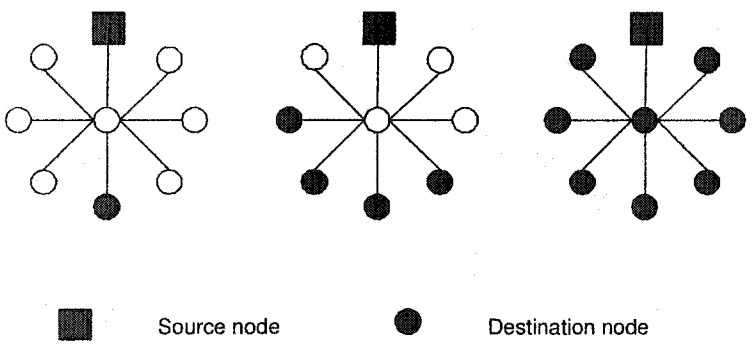

Figure 2.1: Unicast, Multicast and Broadcast

Multicast is a one-to-many communication method where data can be sent from a source node to multiple destination nodes. We usually consider multicast routing problems in Local Area Networks (LANs) and Wide Area Networks (WANs). The LANs span a small geographical area but the WANs span a larger area as shown in Figure 2.2. Usually the nodes connected to LANs communicate over a broadcast network, while nodes connected to WANs talk to each other via a switched or router

network [18, 34, 15]. It is easy to implement a multicast communication in LANs, but difficult in switched or router networks. In this study, we focus on multicast in WANs, where packet distribution trees need to be built for multicast routing. These trees help a source node to send packets to all the receivers. Minimizing the amount of network resources employed by multicasting is one of the challenges in network communications.

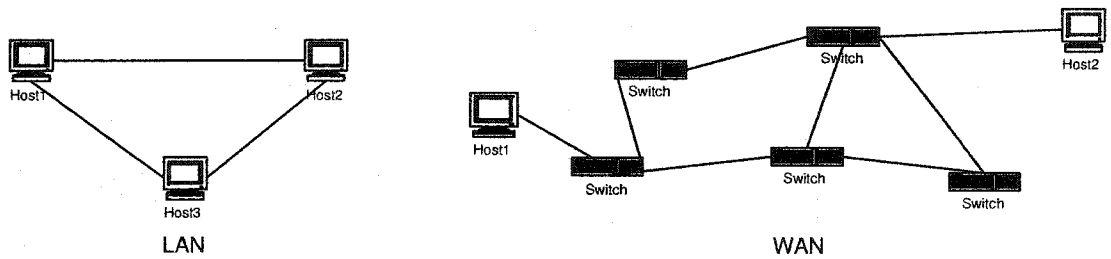

Figure 2.2: LANs and WANs.

There exist many multicast applications in WANs, such as file distribution, interactive games, news feeds and video conferences. The implementations of most of these applications are not efficient because most of them only support point-to-point (unicast) communications. Efficient multicasting support in WANs is necessary in order to make these applications more popular and less bandwidth-intensive.

2.1 Problems and Motivation

The underlying communication network is modeled as an edge-weighted complete graph $G(s, V, D)$, where s is the source node, D is the destination node set and V is the set of all the nodes in the network including those nodes that can be used only as intermediate nodes (called Steiner nodes) [19]. In general the edge weight function is additive. Since every node can act as an intermediate medium for forwarding data, we may assume without loss of generality that the edge weight is equal to the cost of the shortest path connecting the two ending nodes in G. Therefore, the edge weight function naturally satisfies the triangle inequality.

In order to perform multicast communications in WANs, the source node and all

the destination nodes must be interconnected by a tree. The problem of multicast routing in WANs is thus treated as finding a multicast tree in a network that spans the source and all the destination nodes. The goal is to minimize the total cost of the multicast tree, which is defined as the sum of the costs of all the edges in the tree.

We focus on the Capacitated Multicast Routing problem where messages will be sent out one at a time. Each time, only the nodes on one path (tree) can receive the message because not all of the switches or routers in the network have the broadcasting ability and this path (tree) can contain a limited number of nodes. There are two such multicast routing models. The first one is the Multicast k-Path Model which can be regarded as a generalization of the one-to-one connection. The purpose is to find a set of paths where the nodes on the path can receive data and the total cost, which is measured as the sum of the weights of the edges on the routing paths, is minimized. When the number of destination nodes in a path is limited to k, we call it Multicast k-Path Routing (kMPR) problem. The second model is the Multicast k-Tree Model which can also be regarded as a generalization of the one-to-one connection. Under this model, multicast routing is to find a set of trees such that each tree includes only a limited number of destination nodes which are supposed to receive data and every destination node must be designated to receive the data in one of the trees. The goal is to partition the destination node set D into a collection of subsets such that each subset of destination nodes can be arranged into a routing tree and the total routing cost is minimized. When the number of destination nodes in a tree is limited to k, we call it the Multicast k-Tree Routing (kMTR) problem.

2.2 Contributions

For the kMPR problem, we propose a 3-approximation algorithm which improves on the previous best approximation with performance ratio 4 [18]. Another $(2+\rho)$-approximation algorithm for the kMTR problem is also presented. The previous best approximation algorithm for kMTR is given in [22] with performance ratio $(2.4+\rho)$. In the following chapters, these two approximation algorithms will be described in detail.

Chapter 3

Multicast k-Path Routing

The Multicast k-Path Routing (kMPR) problem has mentioned above. Unidirectional transmissions are used here. Data needs to be transmitted from the source to all destination nodes. It is assumed that if there is a link between two nodes (switches or routers) u and v in the network, then there are two paths between them, one carrying the transmission from u to v and the other from v to u. Data can be transmitted on these two paths simultaneously.

The kMPR problem, for $k = 1$ or $k = 2$, is not NP-hard. There exist polynomial time algorithms for 1MPR and 2MPR [14, 18].

3.1 Polynomial Time Algorithms for 1MPR and 2MPR

For the kMPR problem, the solution to the case $k = 1$ is just a star centered at the source s. The optimal solution to the 1MPR problem thus consists of $|D|$ shortest paths [1] from the source node s to each of the $|D|$ destination nodes. It can be solved in polynomial time. The following theorem is on 2MPR.

Theorem 3.1.1 [18] *The 2MPR problem is polynomial time solvable.*

To prove the theorem, we present a polynomial time algorithm for 2MPR in the following part. In fact, the 2MPR problem can be reduced to a graph matching problem and thus can be solved in polynomial time [18]. Firstly, a multicast connection

(s, D) is obtained in the network $G(s, V, D)$, where $D = \{d_1, d_2, ...d_{|D|}\}$ (Figure 3.1 contains four destination nodes in black). The reduction can be done as follows.

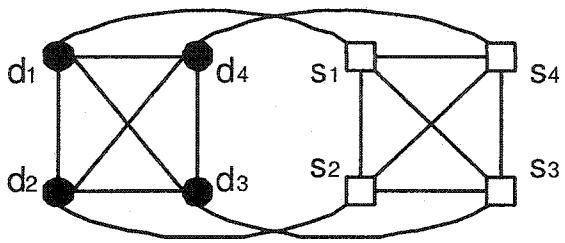

Figure 3.1: Reduction from the 2MPR problem to a minimum weight matching problem.

For $d_i, d_j \in D$ and $i \neq j$, denote the shortest path between these two nodes as $p_G(d_i, d_j)$ with the weight $c(p_G(d_i, d_j))$ and the shortest path between source s and node d_i as $p_G(s, d_i)$ with the weight $c(p_G(s, d_i))$. Then construct an auxiliary graph $G'(D \cup \{s_1, s_2, ...s_{|D|}\}, E')$ (Figure 3.1). For $d_i \neq d_j$ there is an edge between d_i and d_j with given weight $w(d_i, d_j)$, where

$$w(d_i, d_j) = \min \{c(p_G(s, d_i)) + c(p_G(d_i, d_j)), c(p_G(s, d_j)) + c(p_G(d_j, d_i))\}.$$

For $i \neq j$ there is an edge with given weight $w(s_i, s_j) = 0$ between s_i and s_j. There is an edge between s_i and d_i for each i with given weight $w(s_i, d_i) = c(p_G(s, d_i))$. The minimum weight matching M of G', which is a perfect matching of G', can be found efficiently. From M we can obtain an optimal 2-routing of (s, D) on G in the following way. For each edge $(d_i, d_j) \in M$, if $w(d_i, d_j) = c(p_G(s, d_i)) + c(p_G(d_i, d_j))$, we produce a 2-path from s to d_j via d_i that consists of $p_G(s, d_i)$ and $p_G(d_i, d_j)$, otherwise we produce 2-path from s to d_i via d_j that consists of $p_G(s, d_j)$ and $p_G(d_j, d_i)$ [18]. For each edge $(s_i, d_i) \in M$, produce a 1-path from s to d_i [18]. Since each of the possible shortest 2-path or 1-path is associated with exactly one edge in G', the total cost is the total weight of the edges, and each of the destination nodes is incident to exactly one edge in M [18], we can obtain an optimal solution to the 2MPR problem. It is obvious that this optimal solution of 2MPR can be obtained in polynomial time [18].

3.2 kMPR is NP-hard

The general case of the kMPR problem is more difficult. According to the specification of our problem, we model the network as an arc-weighted digraph $G(V, A, c)$, where the vertex-set V is the set of nodes in the network representing switches/routers and the arc-set A is the set of links between nodes representing wires. For arc $(u, v) \in A$, a cost function $c : A \rightarrow R^+$ measures the desirability of using a particular arc. We also assume that $G(V, A, c)$ is totally symmetric [18]. We will consider the decision version of the kMPR problem. Given a multicast connection (s, D) in a network G, an integer $k > 2$ and a bound $B \geq 0$, the problem asks if there exists a k-routing for (s, D) whose cost is at most B.

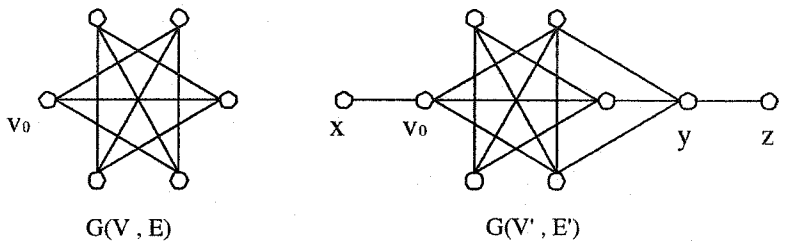

Figure 3.2: Constructing a new graph $G(V', E')$ from a given 3-regular graph $G(V, E)$.

It was proven in [25, 12] that the Hamiltonian Circuit Problem for the 3-regular graphs in which all nodes are of degree three (the left graph in Figure 3.2) is NP-complete. It was also proven in [25] that Hamilton Path Problem is NP-complete through a simple reduction [18] as follows. Given a 3-regular graph $G(V, E)$, construct a new graph $G(V', E')$ where

$$V' = V \cup \{x, y, z\} \quad \text{and} \quad E' = E \cup \{(y, z), (x, v_0)\} \cup \{(y, v) | (v, v_0) \in E\},$$

for some fixed $v_0 \in V$. It can be verified that $G(V, E)$ has a Hamilton circuit if and only if $G(V', E')$ has a Hamilton path (see Figure 3.2) [18].

The Hamilton Path Problem for the above defined graph $G(V', E')$ can be reduced in polynomial time to the kMPR problem. First, construct $G(V', A, c)$ by substituting each edge $(u, v) \in E'$ with a pair of arcs (u, v) and (v, u) whose costs are equal to 1. Then, set $s = x$, $D = V' \backslash \{x\}$, $B = |V'|$ and $k = |V'|$. It is easy to verify that

$G(V', E')$ has a Hamilton path if and only if $G(V', A, c)$ has a k-routing for (s, D) whose cost is at most B [18].

In the following section we will review a 4-approximation algorithm for the kMPR problem which was the previous best approximation algorithm [18].

3.3 Previous Best Approximation Algorithm for the kMPR Problem

Before we describe the algorithm, the Minimum Spanning Tree problem needs to be introduced first.

Definition 3.3.1 [6] *A minimum spanning tree of an edge-weighted graph is a tree which connects all the vertices and has minimal total weight.*

A minimum spanning tree can be found in polynomial time. The most famous algorithm is described in Prim [26]. This algorithm can compute a minimum spanning tree for a graph with n vertices and m edges in time $O(m + nlogn)$. In the previous best approximation algorithm for the kMPR problem, the minimum spanning tree is used. In the following part, we introduce that algorithm.

Given a network $G = (s, D)$, let $P_1^*, P_2^*, \cdots, P_m^*$ be the set of paths in an optimal k-path routing. Let $c(P_i^*)$ denote the cost of path P_i^*, which is the sum of the costs of the edges on the path. Let $R^* = \sum_{i=1}^{m} c(P_i^*)$ be the cost of the path routing. The 4-approximation algorithm proposed in [18] constructs a minimum spanning tree T on $s \cup D$, duplicates the edges in T to produce a Hamiltonian cycle [6] C via suitable short-cutting, and then partitions the cycle C into segments each containing exactly k distinct destinations (the last segment might contain less than k distinct destinations). Every segment is connected to the source s via a shortest path from s. Since the cost of a minimum spanning tree T is at most R^* (Note that $P_1^*, P_2^*, \cdots, P_{m-1}^*$ and P_m^* themselves form a spanning tree), the cost of the cycle C is no more than $2R^*$. It is obvious that the total cost of the shortest-paths added in order to connect segments to the source s is at most R^*. However, since for every segment the shortest path connecting from the source s to it could designate at an internal node on the path,

the algorithm uses two copies of the added path to generate two paths in order to produce feasible routings. Therefore, the cost of the resultant k-path routing could be as large as $4R^*$. In fact, the following example shows that the ratio 4 is asymptotically tight. In this example, the optimal k-path routing is $P_1^*, P_2^*, \cdots, P_m^*$, where $P_1^* = s - d_{mk-1} - d_{mk} - d_1 - \cdots - d_{k-2}$, $P_2^* = s - d_{k-1} - d_k - d_{k+1} - \cdots - d_{2k-2}, \cdots$, $P_m^* = s - d_{(m-1)k-1} - d_{(m-1))k} - d_{(m-1)k+1} - \cdots - d_{mk-2}$. The weights are $w(s, d_{ik-1}) = M$ for $i = 1, 2, \cdots, m$, and $w(d_j, d_{j+1}) = 1$ when $j \neq ik-2$. We can see the tree in Figure 3.3. Note that the cost of the optimal k-path routing is $R^* = m(M + k - 1)$. The minimum spanning tree has a cost which is the same as the one for the optimal routing, and the cost of the Hamiltonian cycle is exactly twice of R^*. According to the partitioning, $d_1, d_2, \cdots, d_k$ are on the same segment and d_{k-1} is the closest to the source s. Therefore, the final k-path routing has a cost of $m(4M + 2k - 3)$, which is asymptotically 4 times R^*.

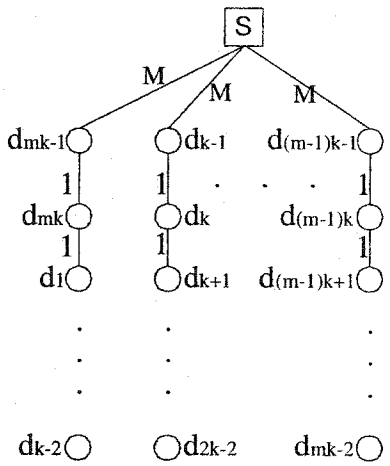

Figure 3.3: Optimal k-path routing

3.4 A 3-Approximation Algorithm for the kMPR Problem

We propose another way to partition the obtained Hamiltonian cycle into segments. Each segment contains exactly k distinct destinations (again, the last segment might

contain less than k distinct destinations), by which the added paths connecting them to the source can be made to be from one of the end destination nodes. The total length of these added paths is no more than R^*.

Observe that in $P_1^*, P_2^*, \cdots, P_m^*$, the distance from every destination node d_i to the source s is an upper bound on the actual distance $d_G(d_i, s)$ calculated in the underlying network G. Suppose the destination nodes are $d_1, d_2, \cdots, d_n$. It follows that

$$\sum_{i=1}^{n} d_G(d_i, s) \leq k \times R^*,$$

since there are at most k destination nodes on every path P_j^* for $j = 1, 2, \cdots, m$. Suppose the destination nodes on the obtained Hamiltonian cycle are indexed consecutively from 1 to n (with source s lying between d_1 and d_n), partition the term $\sum_{i=1}^{n} d_G(d_i, s)$ into k subterms:

$$\sum_{i=0}^{\lfloor \frac{n}{k} \rfloor} d_G(d_{ik+j}, s), \qquad j = 1, 2, \cdots, k.$$

(Note: when the index is out of range, there is no such destination node.) It follows that there exists at least one index j^* such that

$$\sum_{i=0}^{\lfloor \frac{n}{k} \rfloor} d_G(d_{ik+j^*}, s) \leq R^*.$$

Now partition the Hamiltonian cycle into segments. The first one contains the destination nodes $d_{j^*}, d_{j^*+1}, d_{j^*+2}, \cdots, d_{j^*+k-1}$, the second one contains the destination nodes $d_{j^*+k}, d_{j^*+k+1}, d_{j^*+k+2}, \cdots, d_{j^*+2k-1}, \cdots$, and so on. For the ith segment, the path used to connect it to the source s is the edge $d_{j^*+(i-1)k}$. It is clear that every segment appended with the connecting path is still a path and thus they form a feasible routing. Note that the cost of the segments is no more than $2R^*$ and the cost of the added edges/paths is no more than R^*. Therefore, the cost of this routing has cost no more than $3R^*$.

Theorem 3.4.1 *The kMPR ($k \geq 3$) problem admits a 3-approximation algorithm which runs in $O(|V|^3)$ time.*

PROOF. Note that completing the graph might take $O(|V|^3)$ time. After that, computing a minimum spanning tree can be done in $O(|D|^2)$ time and forming the Hamiltonian cycle in $O(|D|^2)$ time. It takes $O(|D|)$ time to compute the partition which is the optimal index j^*. Therefore, the overall running time is $O(|V|^3)$. The performance ratio follows from the the above discussion. ■

Chapter 4

Multicast k-Tree Routing

In this chapter, we introduce another problem (kMTR) in Multicast Routing. In the kMTR problem, the underlying communication network is an edge-weighted complete graph $G(s, V, D)$ where s is a source node, $D = \{d_1, d_2, \ldots, d_n\}$ is a destination node set, and V is a superset of D containing Steiner nodes which can be used as intermediate nodes to reduce the routing cost. The edge weight function satisfies the triangle inequality. The goal is to find a least cost k-tree routing, which contains a set of Steiner trees rooting at s and spanning all destination nodes. Every tree contains at most k destination nodes. Note that in a feasible k-tree routing, one destination node assigned in some trees can be used as a Steiner node in others.

4.1 Polynomial Time Algorithms for 1MTR and 2MTR

For the kMTR problem, 1MTP and 2MTP can be solved in polynomial time. When $k = 1$, the graph is just a star centering at the source, which is the same as 1MPR. When $k = 2$, the problem can be reduced to a graph matching problem and can be solved in polynomial time which is also the same as 2MPR. We have given the algorithms for 1MPR and 2MPR in the previous chapter 3. In the following section we will prove that kMTR is NP-hard when $k \geq 3$ and give a $(2 + \rho)$-approximation algorithm for the kMTR problem.

4.2 3MTR is NP-hard

Given a collection of 3-sets $\mathcal{C} = C_1, C_2, \cdots, C_m$, each of them is a set containing 3 elements from a base set S. The Exact 3-Set Cover problem asks for a subcollection of disjoint 3-sets whose union is S. The Exact 3-Set Cover problem is NP-hard [6].

Theorem 4.2.1 *The 3MTR problem is NP-hard.*

PROOF. Suppose S contains $n = 3q$ elements which are denoted as $s_1, s_2, \cdots, s_n$. Create one destination node for every element s_i, $1 \leq i \leq n$, one Steiner point for

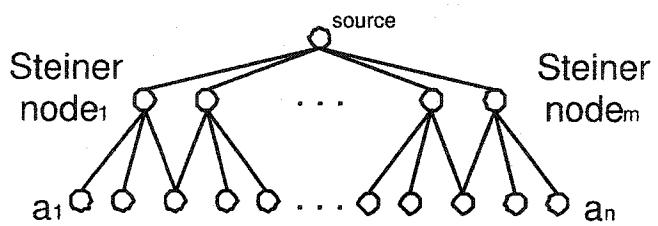

Figure 4.1: Transforming the 3-set cover to 3MTR.

every 3-set C_i and an edge connecting this Steiner point to every destination node inside the set. Create a source node s which is adjacent to every Steiner point. It is clear that so far the source node has degree m and every Steiner point has degree 4. The edges constructed at this point all have cost 1. Finally, complete the graph to obtain G (for example, every pair of destination nodes are connected via an edge with cost 2). It is easy to check that for the instance constructed above, there is a 3-tree routing of cost $4q$ if and only if there is a subcollection $\mathcal{C}'_0 = C_{i_1}, C_{i_2}, \cdots, C_{i_q}$ such that $S = \cup_{j=1}^{q} C_{i_j}$. Therefore, 3MTR is NP-hard. ■

4.3 Currently Best Approximation Algorithm for Steiner Tree

Definition 4.3.1 [13] *Given a graph $G = (V, E)$, a set $R \subseteq V$ of terminals and a weight function of the edges, a Steiner tree is a connected subgraph of G that spans*

all the nodes in R. We call this problem the Steiner tree problem. The Minimum Steiner Tree problem (SMT) in a graph is to find a tree, whose total edge length is the minimum. We denote the total edge length of SMT by smt and extend the definition of the length function $|\cdot|$ from a single edge to arbitrary sets of edges by defining $|X| := \sum_{x \in X} |x|$ for $X \subseteq E$. Similarly we define $|G|$ for a graph $G = (V, E)$ as the total length of all of its edges. In this way we have $smt = |SMT|$.

The first step in our algorithm and the previous best algorithm [22] is to apply the currently best approximation algorithm, the Loss Contracting Algorithm (LCA), for the metric Steiner tree problem. This algorithm will be described below.

The solution of the minimum spanning tree will be used to obtain the approximation result of the minimum Steiner tree. We denote the minimum spanning tree by MST with total edge length mst. The minimum Spanning tree can be found in polynomial time [6].

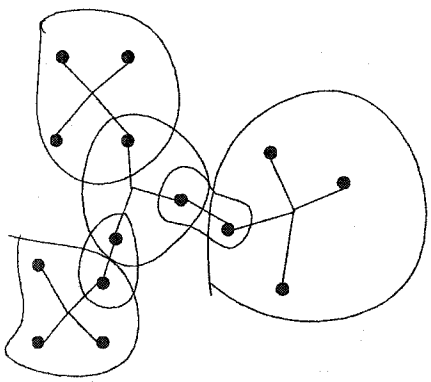

Figure 4.2: The full components of a Steiner tree

It is well known that the Steiner tree problem is NP-hard. The best approximation algorithm, the Loss Contracting Algorithm (LCA), was proposed by Karpinski and Zelikovsky in [27]. They used the k-Steiner trees to obtain an approximation algorithm for the Steiner tree. A full Steiner tree is a Steiner tree where all the terminals(destination nodes) are leaves of the tree. A Steiner tree can be decomposed into the so-called full components by splitting terminals that are interior vertices. A k-Steiner tree is a collection of full components each with at most k terminals Figure

4.2. An optimal k-Steiner tree is denoted by SMT_k and its total edge length is smt_k. It has been proven that when $k \leq 3$, an optimal solution to SMT_k can be found in polynomial time.

Since it is hard to design a good approximation algorithm of SMT, the currently best approximation algorithm for Steiner tree tries to obtain a good approximation algorithm for k-Steiner tree. Borchers and Du [5] have proved that for the Steiner ratio ρ_k which defined as $\rho_k := \frac{smt_k(G)}{smt(G)}$, when $k \to \infty$, $\rho_k \to 1$. So the performance ratio of the algorithm for k-Steiner tree is the one for k-Steiner tree when $k \to \infty$.

The LCA algorithm is a greedy approaches. It fits into the general framework shown in Figure 4.3 [13, 33]. $k \in N$ is fixed. Consider the subsets K' of R with at most k terminals and let K be the collection of those $t \in K'$ for which SMT(t) is a full Steiner tree. The algorithm start with a Steiner tree that is obtained by taking a minimum spanning tree in the distance graph which only includes the terminals. In each step the algorithm try to add element of K to improve the current solution by using a Steiner minimum tree. It is more than one $t \in K$ that could improve the current solution, the algorithm use a selection function f to decide which t will be chosen next. Then how to decide the selection function f is the key point of this approximation algorithm [13]. We will describe the f in LCA later.

```
Let K be the set of full components of each up to k terminals;
    i ← 0;
    While a component which can improve the solution exists do:
      Choose t_{i+1} ∈ K that minimizes the selection function f_i;
      i ← i + 1;
      i_max ← i;
    Output a Steiner tree using t_1, ···, t_max.
```

Figure 4.3: A general framework for greedy algorithms

Contraction is a concept that will be used in LCA. Denote the minimum spanning tree in the graph for a set of required vertices R by $MST(R)$. Assume we add a new edge e between a pair of terminals. Define as:

$$MST(R/e) := \text{a minimum spanning tree for } R \text{ in } G + e.$$

For the Contraction Lemma [27], assume that we insert a certain full component by adding a set E_1 of new edges to the graph. Denote E_0 as a set of edges which has already been added for earlier full components. The length of the current intermediate solution is denoted by $mst(R/E_0)$. Then given another full component E_2, we can obtain the Contraction Lemma.

$$mst(R/E_0) - mst(R/E_0E_2) \geq mst(R/E_0E_1) - mst(R/E_0E_1E_2).$$

The loss of a Steiner tree introduced by [20] measures the length needed to connect the Steiner point of a full component to its terminals. The loss of a set of Steiner vertices $A \subseteq U$ is a minimum length forest $Loss(A) \subseteq E$ in which every Steiner vertex $v \in S$ is connected to a terminal $r \in R$. Contracting the loss of a full component means that for every edge between the loss components, a new edge with the same weight is inserted between the corresponding terminals. We denote it as $loss := |Loss|$.

We use the Contraction Lemma in this problem as well since it is required that the length of the newly inserted edges do not depend on the previous loss contractions involving the same Steiner vertices. By a simple preprocessing (duplicating Steiner vertices), we can achieve that no two full components of the graph share a Steiner vertex. While the length of SMT_k does not change, the instance grows by a factor which is at most a polynomial of the input size. The set K from the general framework will refer to the preprocessed instance.

Lemma 4.3.1 (Karpinski, Zelikovsky [27]) *The length of the loss of a Steiner tree is at most half of its total length.*

PROOF. It is not difficult to prove the inequality $loss \leq \frac{smt}{2}$ for full components. It is easy to see that any full component can be transformed into a complete binary tree of which the leaves are exactly the terminals. This can be obtained by adding some new terminals and new edges of length 0. After that, for each internal vertex, choose from the two edges leading to its children the cheapest one. This will generate a subgraph that includes the loss of the full component with length at most half of the total length. ■

The currently best $(1+\frac{\ln 3}{2})$-approximation algorithm is presented in the following. Its general framework is the same as the one for greedy algorithms, and it does not entirely contract the selected full component but the loss component.

This equation is used for its length: [13]

$$m(\cdot) := mst(R/loss(\cdot)),$$

where m is the length of a minimum spanning tree after the loss of certain full components has been contracted. Because of the preprocessing, after adding some new edges between terminals we can get the effect of a loss contraction. We use the contraction lemma to prove as follows.

Suppose that some of the full components $T_1, T_2, \cdots, T_i$ have already been chosen, then the length of the corresponding Steiner tree is [13]

$$cost(T_1, T_2, \cdots, T_i) = m(T_1, T_2, \cdots, T_i) + loss(T_1, T_2, \cdots, T_i). \tag{4.1}$$

By the preprocessing step and the definition of m. The selection function [13]

$$f_i(T) := \frac{loss(T)}{m(T_1, T_2, \cdots, T_i) - m(T_1, T_2, \cdots, T_iT)} \tag{4.2}$$

is applied to compare the loss of a new full component T with the reduction of m. Then the loss contracting algorithm fits into the general framework. Now we will see that $f_i(T_{i+1}) \leq 1$ for all i.

Theorem 4.3.2 (Robins, Zelikovsky [27]) *The loss contracting algorithm computes a* $(1 + \frac{\ln 3}{2})$ *approximation for* SMT_k.

Before proving the theorem, we still need to describe some notations. As we mentioned before, $cost_i = m_i + loss$. Let $T_1^*, T_2^*, \cdots, T_{jmax}^*$ be the full components of a Steiner minimum tree. Then $smt_k = m^* + loss^*$, where $m^* := m(T_1^*, T_2^*, \cdots, T_{jmax}^*)$.

Theorem 4.3.3 [13] *The Steiner tree with full components* $T_1^*, T_2^*, \cdots, T_{imax}^*$ *returned by the loss contracting algorithm satisfies.*

$$cost(T_1^*, T_2^*, \cdots, T_{imax}^*) \leq smt_k + loss^* \ln\left(1 + \frac{mst - smt_k}{loss^*}\right) \tag{4.3}$$

PROOF. A full component T reduces the length of the current intermediate solution if and only if $f_i(T) < 1$ because [13]

$$\begin{aligned} & cost(T_1, \cdots, T_{i_{max}}) - cost(T_1, \cdots, T_i T) \\ = \; & m(T_1 \cdots, T_i) + loss(T_1 \cdots, T_i) - m(T_1 \cdots, T_i T) - loss(T_1 \cdots, T_i T) \\ = \; & m(T_1 \cdots, T_i) - m(T_1 \cdots, T_i T) - loss(T). \end{aligned}$$

Then the next step is to bound the value of $f_i(T_{i+1})$. Let $T_1^*, T_2^*, \cdots, T_{j_{max}}^*$ be the full components of an optimal Steiner tree. The greedy choice implies,

$$f_i(T_{i+1}) \leq \min_j f_i(T_j^*). \tag{4.4}$$

Using the inequality [13]

$$f_i(T_{i+1}) \leq \frac{\sum_j |T_j^*|}{\sum_j mst(R/T_1 \cdots, T_i) - mst(R/T_1 \cdots, T_i T_{jmax}^*)}, \tag{4.5}$$

we can get [13]

$$f_i(T_{i+1}) \leq \frac{\sum_j loss|T_j^*|}{\sum_j m(T_1 \cdots, T_i) - m(T_1 \cdots, T_i T_j^*)}. \tag{4.6}$$

Due to the Contraction Lemma the denominator is bounded from below by [13]

$$m(T_1 \cdots, T_i T^* \cdots, T_{jmax}^*) \leq m(T_1 \cdots, T_i T^* \cdots, T_{j-1}^* T_j^*). \tag{4.7}$$

By monotonicity [13]

$$m(T_1 \cdots, T_i T_1^* \cdots, T_{jmax}^*) \leq m(T_1^* \cdots, T_{jmax}^*), \tag{4.8}$$

we obtain the inequality [13]

$$f_i(T_{i+1}) \leq \frac{loss(T_1^* \cdots, T_{jmax}^*)}{m(T_1 \cdots, T_i) - m(T_1^* \cdots, T_{jmax}^*)} = \frac{loss^*}{m_i - m^*}. \tag{4.9}$$

Using $f_i(T_{i+1}) \leq 1$ and the previous equation we can estimate [13]

$$\begin{aligned} loss(T_1 \cdots, T_{i_{max}}) &= \sum_i loss(T_i) = \sum f_{i-1}(T_i)(m_{i-1} - m_i) \\ &\leq \sum_i \min\left(1, \frac{loss^*}{m_{i-1} - m^*}\right)(m_{i-1} - m_i). \end{aligned}$$

Clearly $m_0 = mst \geq smt_k$, and we will show that $smt_k \geq m_{i_{max}}$. Therefore the previous equation is bounded by [13]

$$\begin{aligned}
& \int_{m_{i_{max}}}^{mst} \min\left(1, \frac{loss^*}{x - m^*}\right) dx \\
= & \int_{m_{i_{max}}}^{mst-m^*} \min\left(1, \frac{loss^*}{x}\right) dx \\
= & \int_{m_{i_{max}}-m^*}^{loss^*} 1\, dx + loss^* \cdot \int_{loss^*}^{mst-m^*} \frac{dx}{x} \\
= & \; loss^* - m_{i_{max}} + m^* + loss^* \cdot \ln\left(\frac{mst - m^*}{loss^*}\right) \\
= & \; smt_k - m_{i_{max}} + loss^* \cdot \ln\left(1 + \frac{mst - smt_k}{loss^*}\right),
\end{aligned}$$

and the lemma follows.

Then we can prove Theorem 3.3.2. Since $smt_k \geq \frac{mst}{2}$, we have $mst - smt_k \leq smt_k$. It follows that

$$cost(T_1, \cdots, T_{i_{max}}) \leq smt_k\left(1 + \frac{loss^*}{smt_k} \cdot \ln\left(1 + \frac{smt_k}{loss^*}\right)\right). \tag{4.10}$$

Now we apply the inequality $loss^* \leq \frac{smt_k}{2}$. Elementary calculus shows that $\max\{x \cdot \ln(1 + \frac{1}{x}) \mid 0 \leq x \leq \frac{1}{2}\}$ is attained for $x = \frac{1}{2}$. Therefore,

$$cost(T_1, \cdots, T_{i_{max}}) \leq smt_k\left(1 + \frac{\ln 3}{2}\right). \tag{4.11}$$

So the performance ratio is $(1 + \frac{\ln 3}{2})$.

4.4 Previous Best Approximation Algorithm for the kMTR Problem

Let $T_1^*, T_2^*, \cdots, T_m^*$ be the set of trees in an optimal k-tree routing. Recall that every T_i^* might contain some Steiner nodes and might also contain some destination nodes which are *not* allowed to receive data (but act as Steiner nodes).

Let $c(T_i^*)$ denote the cost of tree T_i^*, which is defined to be the sum of the weights of the edges in the tree. Let $R^* = \sum_{i=1}^{m} c(T_i^*)$ be the cost of the routing tree. Since every destination node d_i in the tree T_j^* satisfies $w(s, d_i) \leq c(T_j^*)$, we have

$$\sum_{i=1}^{n} w(s, d_i) \leq k \times R^*. \tag{4.12}$$

In the $(2.4+\rho)$-approximation algorithm [20], the first step is to apply the currently best approximation algorithm for the metric Steiner tree problem (which has the worst-case performance ratio ρ) to obtain a Steiner tree T on $s \cup D$ in the underlying network G. Since the cost of an optimal Steiner tree is a lower bound of R^*, we know that the cost of tree T is upper bounded by ρR^*. Note that tree T is not necessarily a feasible routing tree since some branch rooted at the source s might contain more than k destination nodes. Nonetheless, if there is any branch which contains at most k destination nodes, we can just leave the branch alone in the next step. For branches containing more than k destination nodes, we perform the following partition on each of them in the second step. To present the partition technique, we need the following lemmas.

Lemma 4.4.1 [18] *Given a tree T containing $n \geq 3$ nodes, it is always possible to partition it into two subtrees which have at most one common node and the number of the nodes in both subtrees fall in the closed interval $[\frac{1}{3}n, \frac{2}{3}n]$.*

PROOF. Root tree T at any node, say node v. For every node u in T, let $c(u)$ denote the number of the nodes in the subtree rooted at u, inclusive. Let r be the bottommost node with its $c(r) > \frac{2}{3}n$. Note that r is unique and it could be the root node v. Now re-root tree T at node r. It is easy to see that in the newly rooted tree none except r can have c-value greater than $\frac{2}{3}n$. ■

If there exists a child node w of r satisfying $c(w) \in [\frac{1}{3}n, \frac{2}{3}n]$, then cutting off edge (r, w) from T gives two subtrees which do not overlap and the number of the nodes in both subtrees fall in the closed interval $[\frac{1}{3}n, \frac{2}{3}n]$. In another case, every child node w of r has $c(w) < \frac{1}{3}n$. Numbering these child nodes as $w_1, w_2, \cdots, w_l$. There is some i such that the number of the nodes in the subtree obtained by eliminating subtrees rooted at child nodes $w_1, w_2, \cdots, w_i$ falls in the interval $[\frac{1}{3}n, \frac{2}{3}n]$; and the number of the nodes in the subtree obtained by eliminating subtrees rooted at child nodes $w_{i+1}, w_{i+2}, \cdots, w_l$ falls in the interval $[\frac{1}{3}n, \frac{2}{3}n]$ too. Note that these two subtrees only have one common node which is the root node r.

Lemma 4.4.2 [18] *Given a Steiner tree T containing $n \geq 3$ destination nodes, it is always possible to partition it into two subtrees which have at most one common*

node, either Steiner or destination, and the number of the destination nodes in both subtrees falls in the closed interval $[\frac{1}{3}n, \frac{2}{3}n]$.

PROOF. The proof is similar to the above proof expect that when calculating c-values only those destination nodes are counted. ■

For a branch of T (rooted at the source s) containing more than k destination nodes, delete the edge incident at s to get a subtree denoted as T_1. The second step of the approximation algorithm is to recursively partition T_1 into subtrees each containing no more than k destination nodes using Lemma 4.4.2. We distinguish two cases. In the first case, T_1 contains more than $\frac{5}{4}k$ destination nodes. In this case, we apply Lemma 4.4.2 directly to partition it into two subtrees denoted by T_{11} and T_{12}. It is clear that both subtrees contain at least $\frac{5}{12}k$ destination nodes. The partition stops when each subtree contains no more than k destination nodes, which is called a final subtree for convenience. A final subtree resulted from the first case is called a type-1 final subtree. In the second case, T_1 contains more than k but at most $\frac{5}{4}k$ destination nodes. In this case, the farthest $\frac{1}{2}k$ destination nodes from the source s are treated temporarily as Steiner nodes during the partition and their identities are recovered after the partition. In this way, the number of the destination nodes in each of the two derived subtrees falls in the interval $(\frac{1}{6}k, \frac{1}{2}k + \frac{1}{2}k] = (\frac{1}{6}k, k]$. Therefore, both subtrees become final. Such final subtrees are called type-2 final subtrees. Note that type-2 final subtrees always come in pairs, meaning that a pair of them results from one direct partition in the second case.

After all the subtrees become final, for every type-1 final subtree, pick exactly $\frac{5}{12}k$ destination nodes therein and order them according to their distances from the source s increasingly. For every pair of type-2 final subtrees, pick exactly the first $\frac{1}{6}k$ closest destination nodes (from the source s) in each subtree and order them according to their distances from the source s increasingly. In addition, the $\frac{1}{2}k$ farthest destination nodes left out during the last partition are also picked. These $\frac{1}{2}k$ destination nodes are distributed half to each pair of final subtrees. In this way, each final subtree has been associated with exactly $\frac{5}{12}k$ destination nodes, where the first $\frac{1}{6}k$ closest of them reside in the final subtree (while the other $\frac{1}{2}k$ may not). For simplicity, we leave out the branches of the Steiner tree T each of which contains no more

than k destination nodes and only consider these final subtrees. We denote these final subtrees as $B_1, B_2, \cdots, B_l$. For each B_i, let the associated destination nodes be $i_1, i_2, \cdots, i_{\frac{5}{12}k}$, where $d_G(s,i_1)d_G(s,i_2)\cdots d_G(s,i_{\frac{5}{12}k})$. It follows that

$$\sum_{i=1}^{l}\sum_{j=1}^{\frac{5}{12}k} d_G(s,i_j) \leq k \times R^*. \tag{4.13}$$

Using the order $d_G(s,i_1) \leq d_G(s,i_2) \leq \cdots \leq d_G(s,i_{\frac{5}{12}k})$, we have

$$\sum_{i=1}^{l} d_G(s,i_1) \leq \frac{12}{5}R^* = 2.4R^*. \tag{4.14}$$

This indicates that after the partition, we obtain a set of subtrees each containing at most k destination nodes and adding the shortest paths to connect them to the source s gives us a feasible k-tree routing. This resultant routing tree has a cost no more than $(2.4+\rho)R^*$. We have known that $\rho = 1.55$, so the performance ratio is about 3.95.

4.5 A $(2+\rho)$-Approximation Algorithm for the kMTR Problem

In the $(2+\rho)$-approximation algorithm, we firstly apply the currently best approximation algorithm for the metric Steiner tree problem (which has the worst-case performance ratio ρ) to obtain a Steiner tree T on $s \cup D$ in the underlying network G. Since the cost of an optimal Steiner tree is a lower bound of R^*, we know that the cost of the tree T is upper bounded by ρR^*, that is, $c(T) \leq \rho R^*$. Note that the tree T is not necessarily a feasible routing tree yet since some branch rooted at the source s might contain more than k destination nodes. We treat T in the following way: if there is any branch that contains no more than k destination nodes, leave it alone for the next step.

Lemma 4.5.1 [22] *Given a Steiner tree T containing n destination nodes, where $k < n \leq \frac{3}{2}k$ and $k \geq 3$, randomly select $n - \frac{1}{2}k + 1$ destination nodes from the tree to form a set D_0. Then, it is always possible to partition the tree into two subtrees*

T_1 with destination node set D_1 and T_2 with destination node set D_2 which have at most one common node (either destination node or Steiner node). $0 < |D_1|, |D_2| \leq k$, $D_1 \cap D_0 \neq \emptyset$, and $D_2 \cap D_0 \neq \emptyset$.

PROOF. Root tree T at any node, which could be either a destination node or a Steiner node.

In this rooted tree, for every node v, let $c(v)$ denote the number of the destination nodes in the subtree rooted at v (inclusive). Let r denote the farthest (from the root) node which has $c(r) > n - \frac{1}{2}k$. Note that in the case that there is no node having a c-value greater than $n - \frac{1}{2}k$, r is set to be the root. Since $k < n \leq \frac{3}{2}k$, r is uniquely defined. Re-root the tree T at node r.

By duplicating the root node r, we can partition T into two subtrees (both rooted at r) T_1 with destination node set D_1 and T_2 with destination node set D_2. Our partition goal is to minimize $|D_2| - |D_1|$, assuming without loss of generality that $|D_2| \geq |D_1|$. If it already holds that $0 < |D_1|, |D_2| \leq k$, $D_1 \cap D_0 \neq \emptyset$, and $D_2 \cap D_0 \neq \emptyset$, then we can obtain the two desired subtrees. Otherwise, $|D_1| < \frac{1}{2}k$ and $|D_2| > n - \frac{1}{2}k$ must hold. We proceed to examine subtree T_2 which must have multiple branches and each of them contains at most $n - \frac{1}{2}k$ destination nodes.

Number these branches as $T_{21}, T_{22}, \cdots, T_{2\ell}$, with the destination node sets D_{21}, D_{22}, $\cdots, D_{2\ell}$, respectively. We distinguish two cases. In the first case, there is a branch, say T_{2i}, such that $|D_{2i}| \geq \frac{1}{2}k$. It follows from $|D_{2i}| \leq n - \frac{1}{2}k \leq k$ that re-partitioning T to have only T_{2i} in subtree T_2, while all the other branches rooted at r are included into subtree T_1, gives the desired partition. That is, $0 < |D_1|, |D_2| \leq k$, $D_1 \cap D_0 \neq \emptyset$, and $D_2 \cap D_0 \neq \emptyset$. In another case, every branch contains less than $\frac{1}{2}k$ destination nodes: $|D_{2i}| < \frac{1}{2}k$, for $i = 1, 2, \cdots, \ell$. Since $|D_0| = n - \frac{1}{2}k + 1 > \frac{1}{2}k + 1$, there are at least two branches, say T_{21} and T_{22}, that both contain destination nodes from D_0 (which is not the root node r). Again, we do the re-partitioning by removing T_{21} from T_2 while including it in T_1. This gives us a new pair of subtrees T_1 and T_2 that satisfies $0 < |D_1|, |D_2| \leq k$, $D_1 \cap D_0 \neq \emptyset$ and $D_2 \cap D_0 \neq \emptyset$, which proves the Lemma.

Recall that every branch of T rooted at the source s is ignored for further consideration. In the following, we will focus on the operations performed on one branch of T (rooted at the source s) containing more than k destination nodes. First of

all, we delete the edge incident at s from the branch to get a subtree denoted as T_1. Secondly, if T_1 contains more than $\frac{3}{2}k$ destination nodes, we apply Lemma 4.4.2 to partition T_1 into two subtrees. We then repeatedly apply Lemma 4.4.2 to partition the resultant subtrees if they contain more than $\frac{3}{2}k$ destination nodes. At the end of this repeated partition, there will be a set of subtrees such that each contains no more than $\frac{3}{2}k$ destination nodes. It should be noted that each of them contains at least $\frac{1}{2}k$ destination nodes since we started with T_1 which contains more than $\frac{3}{2}k$ destination nodes. At this point, for those subtrees which contain no more than k destination nodes, we may leave them alone. For ease of presentation, we call the subtrees containing at most k destination nodes *final trees*. The subtrees become final at this point are called *type-1 final trees*. The non-final subtrees will become *type-2 final trees* after the next step of partition.

For each non-final-yet subtree again denoted by T_1, our third step is to apply Lemma 4.5.1 to partition it into two final subtrees. Let D_0 denote the set of closest $n - \frac{1}{2}k + 1$ (to the source s) destination nodes in T_1, where n is the total number of destination nodes in T_1. Let T_{11} and T_{12} denote the two resultant subtrees having destination node sets D_1 and D_2, respectively. By Lemma 4.5.1, $0 < |D_1| \leq k$, $D_1 \cap D_0 \neq \emptyset$, $0 < |D_2| \leq k$, and $D_2 \cap D_0 \neq \emptyset$. It is clear that type-2 final trees always come in a pair, since they result from one single partition by Lemma 4.5.1.

For each final tree, we pick the closest destination node therein and connect it to the source s. This gives a feasible k-tree routing. In what follows, we will estimate the total cost of these added edges and show that this total cost is at most twice of R^*.

First of all, for every type-1 final tree, we pick the $\frac{1}{2}k$ closest destination nodes therein to be the representatives for the tree. Suppose there are ℓ_1 type-1 final trees $T_1, T_2, \cdots, T_{\ell_1}$. Let the representatives for T_i be $d_{i,1}, d_{i,2}, \cdots, d_{i,\frac{k}{2}}$, in the order of non-decreasing distance from the source s. Secondly, for every pair of type-2 final trees T_1 and T_2, if any one of them contains no less than $\frac{1}{2}k$ destination nodes, then the $\frac{1}{2}k$ closest ones are picked to be the representatives for the tree; otherwise all the destination nodes, say m, are picked to be the representatives and additionally the $\frac{1}{2}k - m$ farthest (to the source s) destination nodes in the other tree are picked to be

the representatives. Therefore, every type-2 final tree has exactly $\frac{1}{2}k$ representatives, although some of them might not come from its own but its partner. Note that the reason we can do so is that the total number of the destination nodes in this pair of type-2 final trees is greater than k. Similarly, assume that there are ℓ_2 pairs of type-2 final trees T_{11}, T_{12}, T_{21}, T_{22}, $\cdots$,$T_{\ell_2 1}$, $T_{\ell_2 2}$. Let the representatives for T_{ih} be $d_{ih,1}$, $d_{ih,2}$, $\cdots$, $d_{ih,\frac{k}{2}}$, where h is either 1 or 2, in the order of non-decreasing distance from the source s. Also for every tree pair T_{i1} and T_{i2}, let $d^0_{i,1}$, $d^0_{i,2}$, $\cdots$, $d^0_{i,\frac{k}{2}}$ be the $\frac{1}{2}k$ closest destination nodes among all the destination nodes in both of them, and let $d^0_{i,\frac{k}{2}+1}$, $d^0_{i,\frac{k}{2}+2}$, $\cdots$, $d^0_{i,k}$ be the $\frac{1}{2}k$ farthest destination nodes among all the destination nodes in both of them.

It follows that

$$\sum_{i=1}^{\ell_1}\sum_{j=1}^{\frac{k}{2}} w(s,d_{i,j}) + \sum_{i=1}^{\ell_2}\sum_{j=1}^{k} w(s,d^0_{i,j}) \leq \sum_{i=1}^{n} w(s,d_i) \leq k \times R^*. \tag{4.15}$$

Using the non-increasing distance orderings of these destination nodes, we have

$$\sum_{i=1}^{\ell_1} w(s,d_{i,1}) + \sum_{i=1}^{\ell_2} (w(s,d^0_{i,1}) + w(s,d^0_{i,\frac{k}{2}+1})) \leq 2R^*. \tag{4.16}$$

Clearly, for every type-1 final tree T_i, the destination node $d_{i,1}$ is connected to the source s; also it is true that $d^0_{i,1}$ must serve as a representative for either type-2 final tree T_{i1} or type-2 final tree T_{i2} and thus it is connected to the source s. Suppose without loss of generality that $d^0_{i,1}$ is a representative for T_{i1}, then the closest destination node d in T_{i2} which is picked to be a representative has a distance no larger than the distance from the source to the destination node $d^0_{i,\frac{k}{2}+1}$. It follows that the total cost of the edges added to connect the source to the final trees to produce a feasible k-tree routing is at most $2R^*$. Therefore, the produced routing tree has a cost no more than $(2+\rho)R^*$.

Theorem 4.5.2 *kMTR ($k \geq 3$) admits a $(2+\rho)$-approximation algorithm, where ρ is the best performance ratio for approximating the metric Steiner tree problem.*

It is known that ρ is about 1.55 [13, 27]. Therefore, our approximation algorithm has a performance ratio of about 3.55. It is worth mentioning that the running time is dominated by the approximation algorithm for the metric Steiner tree problem.

4.6 Conclusion

We design a better approximation algorithm for the multicast k-tree routing problems with the worst case performance ratio $(2+\rho)$. On the way to this better approximation, an interesting tree partitioning technique has been developed. We believe this promising partitioning technique can be further combined with other existing methods to achieve better approximation algorithms.

Chapter 5

Conclusion

We have developed an averaging technique and a tree partition technique for designing better approximation algorithms for both of the kMPR and kMTR problems when $k \geq 3$. We present a 3-approximation algorithm for the kMPR problem. The previous best approximation algorithm has a performance ratio of 4. For the kMTR problem, our algorithm has the worst case performance ratio $(2 + \rho)$, where ρ is the best approximation ratio for the metric Steiner tree problem (which is about 1.55). The previous best approximation algorithm has a performance ratio of $(2.4 + \rho)$.

Part II

Binary Fingerprint Vector Clustering

Chapter 6

Clustering Binary Fingerprint Vectors with Missing Values

Designing approximation algorithms is a good way to get suboptimal solutions for many NP-hard optimization problems [12], typically in application domains such as networking. Nonetheless, in some other applications, we care more about the quality of the solution than the actual running time. In such circumstances, we choose to design exact algorithms to solve the problems as fast as possible, although they might still run in exponential time in the worst case.

In this part of the thesis, we will examine the problem of clustering binary oligonucleotide fingerprint vectors with missing values, which is an application model from the DNA microarray analysis. We will present an A* search algorithm to both minimize the number of clusters and resolve the missing values in the fingerprint vectors. Except the trivial exhaustive enumeration method, our search algorithm is the first exact algorithm that solves the problem optimally. Our search algorithm employs some existing work [11] on this problem. Experimental results on real datasets show that in terms of running time, our search algorithm is very competitive to a heuristic greedy search algorithm proposed in the literature, and in terms of quality, our search algorithm guarantees an optimal solution while the heuristic greedy search algorithm does not.

6.1 Introduction

It is widely believed that, in a living organism, thousands of its genes and their products (i.e., RNA and proteins) function in a complicated but orchestrated way. In the past several years, a new technology, called *DNA microarray* [23], has attracted tremendous interest among biologists. This technology can be used to monitor the whole genome on a single chip so that one can have a *whole* picture of the interactions among thousands of genes simultaneously. A DNA array is an orderly arrangement of known or unknown DNA samples, in order to provide a medium for matching these samples based on Watson-Crick base-pairing rules. Various array designs exist depending on the applications, e.g. *Oligonucleotide Fingerprinting* [16, 17, 7, 24, 8] , in which an array of oligonucleotide (20 to 80 oligos) or peptide nucleic acid (PNA) sequences (called *clones*) is synthesized either *in situ* (on-chip) or by conventional synthesis followed by on-chip immobilization. The array is then exposed to labeled sample DNA (called *probes*), hybridized, and the identity or abundance of complementary sequences is determined. Generally, a probe is a type of short, single-stranded fluorescence-labeled DNA. It will hybridize to the spot on the chip when the probe occurs as a substring of the clone on the spot. After hybridizing, all of the unbound probes will be washed off and the hybridization intensity values between the probe and the clones can be measured. The hybridization experiment, where a fingerprint is simply a vector consisting of the hybridization intensity values between the clone and the probes, is repeated for a set of probes to create fingerprints of the clones. In this way, oligonucleotide fingerprints [8, 28, 30, 32] are regarded as vectors containing hybridizing signal intensities.

Oligonucleotide fingerprinting [8, 24, 28, 30] is one of the best methods to characterize DNA clone libraries. It was adopted in many applications, such as gene expression profiling and DNA clone classification. In particular, it offers an effective way to extensively analyze microbial communities. In this part, we focus on the application of classification of the DNA clones, a problem arisen from the classifications of microorganisms [28, 30, 3, 9, 10].

In order to cluster the clone fingerprints, usually we distinguish hybridization intensity values by binary values where 1 means hybridization and 0 means the opposite. However, it is in general hard to determine whether the clones are hybridized or not. In fact, most of the current methods do not offer an effective way to clearly determine whether the clones are hybridized or not. Recently, a discrete approach, where reference intensity values are decided by controlling clones with known characteristics with respect to the probes that are included in the DNA array experiments, has been applied to the classification of microbial rDNA clones [11, 4]. By doing so, the oligonucleotide fingerprinting data can be normalized and binarized using these reference intensity values. The intensity value is set to 1 meaning hybridization, 0 no hybridization, and N means a missing value.

After normalizing and binarizing oligonucleotide fingerprinting data, the problem is transformed to identifying clusters and solving the problem of missing values in the fingerprints. Suppose there are totally n clones on the DNA array, and m probes. The oligonucleotide fingerprinting data is a set of n vectors of dimension m, and every vector entry takes a value of 1, 0, or N. We consider the problem of identifying clusters and resolving the missing values in the fingerprints simultaneously. A vector containing no N entry is called a *resolved* vector. For a pair of vectors containing some N entries, it is possible that through assigning a 1 or a 0 to every N, the two resolved vectors become identical. If this is the case, we say that these two vectors can be resolved into a cluster. Our task is to assign a 1 or a 0 to every N in the given set of n vectors so that the number of distinct resolved vectors , which represent the number of clusters, is minimized. We call this combinatorial optimization problem the *Assignment Clustering Problem* or ACP for short.

One natural parameter in ACP is the maximum number of N's in a vector. When every vector contains no more than k N's, the problem is called kACP. It is known that 1ACP can be solved in polynomial time and kACP where $k \geq 3$ is NP-hard [11]. The complexity of 2ACP is unknown. On the approximability aspect, kACP where

$k \geq 2$ admits a 2^k-approximation algorithm [2].

The main task in this part is to solve kACP optimally. To achieve our target, we propose to employ A^* [21], a heuristic search algorithm, from the Artificial Intelligence community with some carefully designed heuristics and evaluation functions. As the reader will see, the 2^k-approximation algorithm for kACP serves as an evaluation function for our purpose. We tested our search algorithm on real datasets. The experimental results demonstrated that our exact algorithm runs fast. It could be used to produce some benchmark data for evaluating other algorithms developed in the past and in the future.

6.2 Previous Work

In [11], a greedy heuristic algorithm GCP based on *Clique Partitioning* was proposed. The key idea in the algorithm is to transform an instance of kACP into an instance of *Minimum Clique Partitioning*. For each given vector v^i, create a vertex denoted as v^i. For every pair of given vectors, if they can be resolved into a cluster through assigning suitable values to their N entries, then there is an edge connecting two corresponding vertices. Denote the obtained graph as $G = (V, E)$. It can be seen that the vectors residing in a common clique in G can be resolved into a cluster. Thus, the corresponding goal in the Clique Partitioning problem is to find a minimum number of cliques that include / cover all the vertices. Targeting at the minimization objective, GCP picks the maximum clique at every iteration, removes the vertices therein from the graph, and repeats this process till the graph becomes empty. It runs in time $O(k2^k n^2)$, where n is the number of given vectors. The theoretical performance guarantee of GCP [11] is much worse than its performance on real datasets, as the reader will see in the experiment results.

6.3 A 2^k-Approximation Algorithm for kACP

Definition 6.3.1 *A given vector set $E = \{a_1, a_2, a_3, \cdots, a_n\}$, we transform all the given vectors into resolved vector by assigning either 0 or 1 to those N's. The set*

of the resolved vector is $P = \{p_1, p_2, p_3, \cdots, p_m\}$. *Then we can obtain the subsets* $\{S_1, S_2, S_3, \cdots, S_m\} \subset E$. *Subset* S_k $(1 \leq k \leq m)$ *includes the given vectors which the resolved vector* p_k *represents. The goal is to find a minimum number of subsets that cover all of the elements in* E. *When each given vector contains at most* k $N's$, *we call this problem the* kACP *problem.*

To implement this goal, we first design the kACP program in terms of integer programming, and create a variable x_j for each subset S_j. If the subset is chosen, then $x_j = 1$; otherwise $x_j = 0$.

$$\min \sum_{j=1}^{m} x_j$$

subject to :

$$\sum_{j:a_i \in S_j} x_j \geq 1, \qquad \forall a_i \in E$$
$$x_j \in \{0, 1\}$$

As we know that the general integer program cannot be solved in polynomial time, but the resulting linear program (LP) can be solved in polynomial time, so we transform it to a linear program.

$$\text{Min} \sum_{j=1}^{m} x_j$$

subject to :

$$\sum_{j:a_i \in S_j} x_j \geq 1, \qquad \forall a_i \in E$$
$$x_j \in [0, 1]$$

Let OPT be the optimal result of this problem. For the kACP, an element can belong to at most 2^k subsets. Denote x_j^* as a result of the LP.

$$T = \left\{ S_j \,|\, x_j^* \geq \frac{1}{2^k} \right\}. \tag{6.1}$$

PROOF. T is a solution of kACP, and $|T| \leq 2^k \text{OPT}$.

If there is an element $a_i \notin T$, then

$$\sum_{j:a_i \in S_j} x_j^* < \frac{1}{2^k} \cdot |\{j : a_i \in S_j\}| \leq 1. \tag{6.2}$$

The equation (6.2) violates the linear programming constraint for a_i. ■

Next step we should prove that $|T| \leq 2^k$ OPT

$$\sum_{j=1}^{m} x_j \leq \sum_{j=1}^{m} (2^k . x_j^*) = 2^k \sum_{j=1}^{m} x_j^* \tag{6.3}$$

So we can use LP to design a 2^k-approximation algorithm for kACP.

Chapter 7

Exact Algorithms for ACP

7.1 A Polynomial Time Algorithm for the 1ACP Problem

For 1ACP problem, suppose there is a given vector set V. In the first step, we delete the given vectors which do not contain N from set V and store them to the set De so that all the vectors in De are resolved vectors. Then next step we delete the given vectors from set V which can be represented by the vectors in De. After this step we can obtain a new given vector set $V' = v_1, v_2, \cdots, v_m$, where v_i $(1 \leq i \leq m)$ contains one N. Now we transform each given vector v_i in two resolved vectors and store them in the set $R = r_1, r_2, \cdots, r_l$. Each resolved vector in R is represented by a vertex in a graph. If two resolved vectors can represent the same given vectors in V', connect these two nodes with an edge. Partition the set R to two sets. One is X which includes nodes with odd number of 1's, the other is Y which includes nodes with even number of 1's. Then the graph can be denoted as $G(X, Y, E)$. Since any two resolved vectors in X or Y cannot represent a given vector, the edges exist only between nodes in set X and nodes in set Y. Thus the graph G is a bipartite graph. Each of the given vectors can be represented by an edge in G. The 1ACP is transformed into finding a minimum number of vertices that can cover all of the edges. This is the minimum vertex cover problem. It is well known that the minimum vertex cover problem in bipartite graphs can be solved in polynomial time.

7.2 Heuristic Search for kACP

In this section, we describe our exact algorithm to find an optimal solution for an instance of kACP and it can be generalized to deal with kACP for any k. We adopted A*, one of the most famous heuristic search algorithms developed originally in Artificial Intelligence (AI), in our algorithm. A* is a Best-First search algorithm which has been used extensively in many areas of AI and has been successfully applied to various bioinformatics problems, the most notable of which is probably *Multiple Sequence Alignment* (MSA). MSA is a controversial problem in computational biology. This particular problem computes the similarity based on the biological properties of nuclei acid (or amino acid) among the DNA strands (or protein sequences). When this biological problem is mapped to a computing science problem, the formulation becomes finding the similarity between multiple strings. The similarity of two aligned characters relies on the cost function, which will return a distance (or score value). The similarity of the alignment, then, is the sum of all pair aligned characters distances (or all pair scores). The optimal pair-wise alignment is referred to align two strings and spaces could be inserted into each string to obtain the optimal similarity. The basic idea of A* is that rather than trying all possible search paths, trying and focusing on paths that seem to be getting nearer to the goal. For each state, A* uses both the exact distance from the root state, which is denoted as g, and a heuristic estimate of the remaining distance to the goal state (the *heuristic evaluation function*), which is denoted as h. The state with the smallest $(g + h)$ value is always expanded next by the algorithm and the algorithm is guaranteed to find the optimal (i.e. minimum weight) solution provided that h always underestimates the true distance to the goal state. Note that in this problem, the distance measures the number of clusters. A* fits for the MSA. A popular admissible heuristic function used is the sum of optimal pairwise alignments. The heuristic value is a lower bound since the cost of the actual alignment of each pair is at least as good as the cost of the optimal pair-wise alignment. The algorithm will first put the initial node into the OPEN list, which stores the nodes that are not fully considered. Then, at each step, it will select the best f-value node from the OPEN list to explore, and the algorithm terminates

when the goal node is found (or no solution when OPEN list becomes empty). If the solution cannot be found after such node is explored, we will put it into the CLOSE list, which prevents the repeated search.

The implementation of A* search for ACP problem follows the graph transformation used in the GCP [11]. The first step is to construct the graph $G = (V, E)$, where V contains all the given vectors and again a pair of vectors are adjacent only if they can be resolved into a cluster. Note that when there is a singleton in G, the isolated vector can be resolved arbitrarily and the resultant cluster contains only this vector. We may remove these singletons (and put the corresponding clusters, i.e. resolved vectors, in the solution) from the graph. After this, if there is any vector which belongs to only one maximal clique of G, then it is always a good idea to create a cluster to include all the vectors in the maximal clique. Therefore, again we can remove such maximal cliques from the graph G (and again put the corresponding resolved vectors in the solution). After all these preprocessing steps, in the resultant graph G every vector must belong to at least 2 maximal cliques and we are ready to start the A* search.

At every state in the search tree, the algorithm picks one maximum clique remaining in the graph G. It then chooses one arbitrary (random) vector, say v, from the clique. Since v belongs to at least 2 maximal cliques, there are different ways to assign values for N's in v to resolve it. The child states of the current state correspond to all the possible ways of resolving. For each child state, the heuristics can also be applied to remove possibly produced singletons and vectors belonging to unique maximal cliques together with the unique cliques. In the search, the distance g from the root state is defined as the number of clusters / cliques created so far. To estimate the h-value The 2^k-approximation algorithm [11] is run on the graph to obtain a clique partition. The number of cliques in the output clique partition divided by 2^k is taken as an estimated distance h to the goal state. The sum, $g + h$, is the value stored at the child state. The A* algorithm chooses the state with the minimum $g + h$ value to expand next. The algorithm terminates when the state to be expanded contains no more given vectors and it returns the $g + h$ value stored at the state as a solution.

The pseudocode of our implementation for the A* search algorithm on kACP is

provided in Figure 7.1. For convenience, we call the process to remove singletons *heuristic* #1, and the process to remove vectors belonging to unique maximal cliques together with their unique maximal cliques *heuristic* #2. We use $h(\ )$ to denote the evaluation function to estimate the distance from current state to the goal state, which is taken as the number of clusters returned by the 2^k-approximation algorithm divided by 2^k.

```
function A_Star(state)
    if Solution_found( )
        return f(state);
    for each successor u_i of state do
        apply heuristic #1;
        apply heuristic #2;
        f(u_i) = g(u_i) + h(u_i);
        add u_i to List T;
    remove state from List T;
    find a state new_state in List T having the minimum f value;
    A_Star(new_state);
```

Figure 7.1: A* implementation for kACP Function Solution_found( ) checks if a solution has been found; Function $g(\ )$ returns the exact distance from root state to state u_i; Function $h(\ )$ is the heuristic evaluation function; This implementation also uses an open list T to store the states waiting to be expanded.

Chapter 8

Experimental Results

We tested our implementation of the A* search algorithm on two real fingerprint datasets, provided by the authors of [11]. One of them is the *bacterial small subunit rRNA genes set* where $n = 1491$, $m = 27$. A vector in this dataset may contain up to 11 N's. The other set is *fungal small subunit rRNA genes*, where $n = 1507$, $m = 26$, and a vector in this dataset may contain up to 14 N's. The implementation was tested on all kACP instances generated from the datasets, where $k = 2, 3, \cdots, 14$. For every specific $k \geq 2$, the vectors in the datasets which contain more than k N's must be modified to be legitimate vectors. This was done by retaining the first k N's and assigning the value 0 to the others. We use two methods to implement both A* and GCP algorithms. These two methods are different from the step which remove the maximal cliques that contain a node only belong to one maximum clique. For the first method, we called $Method_1$ We create graph $G = (V, E)$, and get the information of all the maximal cliques. Each time we go through all the nodes on graph. Once we find that there exist a node only belong to one maximal clique, we remove this maximum clique from the graph. This step is ended when there are none of nodes in graph that only belong to one clique. For the second method, $Method_2$, we do not need all the maximum cliques information at first. After creating the graph $G = (V, E)$, We go through each node and its neighbor nodes. If all of its neighbor nodes are connected pairwise, It means that this node only belong to one clique. We remove this node and its neighbor nodes from graph. this step is ended if there are no such kind of nodes in the graph.

Table 8.1 illustrates the results of our implementation by using $Method_1$. We have also coded the GCP algorithm proposed in [11], using the same preprocessing strategies as in the A* search implementation. On Table 8.2, we implemented both A* and GCP by $Method_2$. The running time of both A* and GCP algorithms implemented by $Method_2$ are much faster than the running time of these two algorithms implemented by $Method_1$.

The results of our GCP algorithm implementation on all the generated datasets are also included in Table 8.1 and Table 8.2. It is interesting to note that the outputs from these two algorithms are the same for all the generated datasets. This might indicate that in practice, the GCP algorithm performs very well. On the other hand, more interestingly, the running times of these two implementations on the datasets differ insignificantly. This implies that in the case when an optimal solution must be guaranteed, the A* search algorithm could be a good candidate. Nonetheless, the preprocessing followed by the 2^k-approximation algorithm might not be a good choice, although it does run fast and does have a certain level of performance guarantee.

8.1 The Optimality of A* Search

From Tables 8.1 and 8.2, it is easy to notice that the 2^k-approximation algorithm doesn't usually produce optimal solutions; However, the heuristic GCP algorithm performs as good as the A* search on all generated instances. The instance in Figure 8.1 shows where GCP algorithm fails to compare the optimum. The solution by the GCP algorithm contains 7 clusters and the solution by the A* search algorithm contains only 6 clusters. Therefore, in the case where optimal solutions should be found, the A* search is preferable.

Dataset	n	m	k	$\tilde{k}$	A*	T_{A*}	GCP	T_{GCP}	Approx	T_{Approx}
Bacteria	1491	27	2	1.94	904	19.093	904	15.625	904	16.023
			3	2.73	841	28.650	841	28.260	841	28.450
			4	3.28	798	44.883	798	43.923	800(+2)	43.965
			5	3.59	786	51.103	786	50.993	786	51.102
			6	3.74	778	69.209	778	66.415	780(+2)	67.507
			7	3.79	773	77.640	773	76.800	775(+2)	76.560
			8	3.82	770	93.153	770	90.279	772(+2)	91.657
			9	3.83	769	104.770	769	104.800	771(+2)	104.670
			11	3.84	769	122.636	769	122.504	771(+2)	122.580
Fungi	1507	26	2	1.99	890	9.894	890	9.889	890	9.413
			3	2.91	783	32.146	783	30.994	785(+2)	31.035
			4	3.61	694	71.763	694	68.688	702(+8)	70.654
			5	4.04	633	113.603	633	108.876	635(+2)	110.876
			6	4.29	595	154.360	595	154.330	597(+2)	154.330
			7	4.43	572	196.041	572	196.592	572	196.006
			8	4.49	563	248.206	563	246.354	563	248.201
			9	4.52	559	277.609	559	276.708	559	277.012
			14	4.54	556	746.623	556	736.619	556	742.890

Table 8.1: The experimental results of A* search, GCP by $Method_1$, and the 2^k-approximation on all the generated datasets from the datasets in [11]. $\tilde{k}$ is the average number of N's in the generated instance of the kACP problem. T_{alg} records the running time(seconds) of the algorithm alg in a Linux PC with 1.0 GHz processor.

Dataset	n	m	k	$\tilde{k}$	A*	T_{A*}	GCP	T_{GCP}	Approx	T_{Approx}
Bacteria	1491	27	2	1.94	904	0.851	904	0.851	904	0.849
			3	2.73	841	2.273	841	2.274	841	2.271
			4	3.28	798	3.120	798	3.160	800(+2)	3.080
			5	3.59	786	3.194	786	3.193	786	3.193
			6	3.74	778	3.835	778	4.115	780(+2)	3.829
			7	3.79	773	3.965	773	3.785	775(+2)	3.961
			8	3.82	770	3.847	770	3.849	772(+2)	3.843
			9	3.83	769	4.047	769	4.125	771(+2)	4.043
			11	3.84	769	3.967	769	4.035	771(+2)	3.959
Fungi	1507	26	2	1.99	890	0.861	890	0.841	890	0.841
			3	2.91	783	2.664	783	2.756	785(+2)	2.660
			4	3.61	694	3.965	694	4.001	702(+8)	3.368
			5	4.04	633	4.438	633	4.486	635(+2)	4.429
			6	4.29	595	5.407	595	5.467	597(+2)	5.405
			7	4.43	572	5.197	572	5.196	572	5.197
			8	4.49	563	5.557	563	5.557	563	5.557
			9	4.52	559	5.698	559	5.697	559	5.696
			14	4.54	556	6.289	556	6.289	556	6.289

Table 8.2: Experimental results of A* search, GCP by $Method_2$, and the 2^k-approximation on all the generated datasets from the datasets in [11]. $\tilde{k}$ is the average number of N's in the generated instance of the kACP problem. T_{alg} records the running time(seconds) of the algorithm alg in a Linux PC with 1.0 GHz processor.

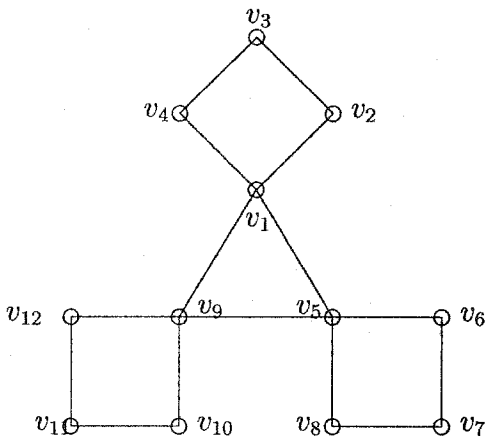

Figure 8.1: An instance of 2ACP where A* returns an optimal solution of 6 while the GCP algorithm doesn't. This instance contains 12 vectors: $v_1 = NN0000000$, $v_2 = 01N000000$, $v_3 = N11000000$, $v_4 = 11N000000$, $v_5 = 000NN0000$, $v_6 = 00001N000$, $v_7 = 000N11000$, $v_8 = 00011N000$, $v_9 = 000000NN0$, $v_{10} = 00000001N$, $v_{11} = 000000N11$, $v_{12} = 00000011N$.

Chapter 9

Contributions and Future Work

We studied the problem of clustering binary fingerprint vectors with missing values. We applied the heuristic search algorithm A* from the AI community to solve kACP optimally where each vector contains at most k missing values.

By now there are no exact algorithms to solve the kACP problem. The A* search algorithm provides a good way to obtain the optimal solutions. Compared with some existing greedy algorithms, the running time and experimental results demonstrated that this proposed exact algorithm is efficient.

Some subjects of interest in my future work are 1) examining the computational complexity for 2ACP; 2) if the problem is NP-hard, then designing better approximation algorithms for 2ACP in order to provide better evaluation functions for A^* search; 3) when generalizing A* search to kACP, designing better approximation algorithms and thus better evaluation functions; and 4) designing or composing benchmark datasets for evaluation, such that algorithms can be fairly compared.

Bibliography

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, Englewood Cliffs, N.J, 1993.

[2] M. J. Atallah. *Algorithms and Theory of Computation Handbook.* CRC Press LLC, 1999.

[3] T. Beissbarth, K. Fellenberg, B. Brors, R. Arribas-Prat, J. M. Boer, N. C. Hauser, M. Scheideler, J. D. Hoheisel, G. Schutz, A. Poustka, and M. Vingron. Processing and quality control of DNA array hybridization data. *Bioinformatics*, 16:1014–1022, 2000.

[4] A. Ben-Dor, R. Shamir, and Z. Yakhini. Clustering gene expression patterns. *Computational Biology*, 6:281–297, 1999.

[5] A. Borchers and D.-Z. Du. The k-Steiner ratio in graphs. *SIAM Journal on Computing*, 32:857–869, 1997.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1989.

[7] R. Drmanac and S. Drmanac. cDNA screening by array hybridization. *Methods in Enzymology*, 303:165–178, 1999.

[8] S. Drmanac, N. Stavropoulos, I. Labat, J. Vonau, B. Hauser, M. Soares, and R. Drmanac. Gene representing CDNA clusters defined by hybridization of 57,419 clones from infant brain libraries with short oligonucleotide probes. *Genomics*, 37:29–40, 1996.

[9] M. Eisen, P. Spellman, P. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences*, 25:14863–14868, 1998.

[10] B. Everitt. *Cluster Analysis.* Edward Arnold, London, 3rd edition, 1993.

[11] A. Figueroa, J. Borneman, and T. Jiang. Clustering binary fingerprint vectors with missing values for DNA array data analysis. In *Computational Systems Bioinformatics(CSB'03)*, pages 38–47, 2003.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman and Company, San Francisco, 1980.

[13] C. Gropl, S. Hougardy, T. Nierhok, and H. J. Promel. Approximation algorithms for the Steiner tree problem in graphs. In D.-Z. Du and X. Cheng, editors, *Steiner Trees in Industries*, pages 235–279. Kluwer Academic publishers, 2001.

[14] J. Gu, X. D. Hu, and M. H. Zhang. Algorithms for multicast connection under multipath routing model. *Information Processing Letters*, 84:31–39, 2002.

[15] R. L. Hadas. Efficient collective communication in WAN networks. In *Proceedings of IEEE ICCCN*, pages 61–66, 2000.

[16] E. Hartuv, A. Schmitt, J. Lange, S. Meier-Ewert, H. Lehrach, and R. Shamir. An algorithm for clustering cDNA fingerprints. *Genomics*, 66:249–256, 2000.

[17] R. Herwig, A. Poustka, C. Muller, C. Bull, H. Lehrach, and J. Obrien. Large-scale clustering of cDNA-fingerprinting data. *Genome Research*, 9:1093–1105, 1999.

[18] X. D. Hu, X. Jia, and M. H. Zhang. Routing algorithms for multicast under multi-tree model. In *Proceedings of IEEE INFOCOM*, 2004.

[19] C. Huitema. *Routing in the Internet.* Prentice Hall, 2000.

[20] M. Karpinski and A. Zelikovsky. New approximation algorithms for the Steiner tree problems. *Journal of Combinatorial Optimization*, 1:47–65, 1997.

[21] Kovitz. Wikipedia, the free encyclopedia. Website, 2001. `http://en.wikipedia.org/wiki`.

[22] G. Lin. An improved approximation algorithm for multicast k-tree routing. *Journal of Combinatorial Optimization*, 2004. Submitted.

[23] G. McLachlan, R. Bean, and D. Peel. A mixture model-based approach to the clustering of microarray expression data. *Bioinformatics*, 18:413–422, 2002.

[24] S. Meier-Ewert, J. Lange, H. Gerts, R. Herwig, A. Schmitt, J. Freund, T. Elge, R. Mott, B. Herrmann, and H. Lehrach. Comparative gene expression profiling by oligonucleotide fingerprinting. *Nucleic Acids Research*, 26:2216–2223, 1998.

[25] C. H. Papadimitrious and K. Steiglitz. *Combinational Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[26] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389—1401, 1957.

[27] G. Robins and A. Zelikovsky. Improved Steiner tree approximation in graphs. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 770–779, 2000.

[28] P. Tamayo, J. Slonim, D. Mesirov, J. Zhu, S. Kitareewan, E. Dmitrovsky, E. Lander, and T. Golub. Interpreting patterns of gene expression with selforganizing maps: methods and applications to hematopoietic differentiation. *Proceedings of the National Academy of Sciences*, 96:2907–2912, 1999.

[29] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, 1996.

[30] L. Valinsky, G. D. Vedova, A. Scupham, S. Alvey, A. Figueroa, B. Yin, R. Hartin, M. Chrobak, D. Crowley, T. Jiang, and J. Borneman. Analysis of bacterial community composition by oligonucleotide fingerprinting of rRNA genes. *Applied and Environmental Microbiology*, 68:3243–3250, 2002.

[31] E. W. Weisstein. Mathworld — a wolfram web resource. Website, 1996. http://mathworld.wolfram.com/NP-HardProblem.html/.

[32] E. Xing and R. Karp. Cliff: Clustering of highdimensional microarray data via iterative feature filtering using normalized cuts. *Bioinformatics*, 17:306–315, 2001.

[33] A. Zelikovsky. An 11/6-approximation algorithm for the network Steiner problem. *Algorithmica*, 9:79–83, 1993.

[34] X. Zhang, J. Wei, and C. Qiao. Constrained multicast routing in wdm networks with sparse light splitting. *IEEE INFOCOM*, pages 1781–1790, 2000.