Minimum Crossing Problems on Graphs

by

Patrick Young Roh

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Combinatorics and Optimization

Waterloo, Ontario, Canada, 2007

©Patrick Roh 2007

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required field revision, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis will address several problems in discrete optimization. These problems are considered hard to solve. However, good approximation algorithms for these problems may be helpful in approximating problems in computational biology and computer science.

Given an undirected graph $G = (V, E)$ and a family of subsets of vertices $\mathscr{S}$, the minimum crossing spanning tree is a spanning tree where the maximum number of edges crossing any single set in $\mathscr{S}$ is minimized, where an edge crosses a set if it has exactly one endpoint in the set. The physical mapping problem in computational biology and the interval routing problem in computer science can both be reduced to finding minimum crossing spanning trees. This thesis will present two algorithms for special cases of minimum crossing spanning trees.

The first algorithm is for the case where the sets of $\mathscr{S}$ are pairwise disjoint. It gives a spanning tree with the maximum crossing of a set being $2 \cdot OPT + 2$, where OPT is the maximum crossing for a minimum crossing spanning tree. This algorithm is an extension of an approximation algorithm for the minimum degree spanning tree due to Fürer and Raghavachari (Journal of Algorithms, 1994).

The second algorithm is for the case where the sets of $\mathscr{S}$ form a laminar family. Let $b_S \in \mathbb{Z}^+$ be a bound for each $S \in \mathscr{S}$. If there exists a spanning tree where each set $S \in \mathscr{S}$ is crossed at most b_S times, the algorithm finds a spanning tree where each set S is crossed $O(b_S \cdot \log n)$ times. From this algorithm, one can get a spanning tree with maximum crossing $O(OPT \cdot \log n)$. This algorithm combines ideas from an approximation algorithm for multicommodity flows on trees due to Garg, Vazirani, and Yannakakis (Algorithmica, 1997) and from an approximation algorithm for minimum degree minimum spanning trees due to Ravi, Marathe, Ravi, Rosenkrantz, and Hunt (Proceedings, ACM Symposium on Theory of Computing, 1993).

The best known approximation algorithm for minimum crossing spanning trees is due to Bilò, Goyal, Ravi, and Singh (Proceedings, International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, 2004). Their algorithm gives a spanning tree with maximum crossing $O(OPT \cdot \log n + \log |\mathscr{S}|)$. The first algorithm of this thesis has a better approximation when the sets of $\mathscr{S}$ are pairwise disjoint. The second algorithm of this thesis has an equivalent approximation (within a constant factor) when the sets of $\mathscr{S}$ form a laminar family. However, their algorithm is probabilistic while the second algorithm of this thesis is deterministic.

Given an undirected graph $G = (V, E)$, and a family of subsets of vertices $\mathscr{S}$, the minimum crossing perfect matching is a perfect matching where the maximum number of edges crossing any set in $\mathscr{S}$ is minimized. A proof will be presented showing that finding a minimum crossing perfect matching is NP-hard, even for the special case where the graph is bipartite and the sets of $\mathscr{S}$ are pairwise disjoint.

Acknowledgements

Thanks to my advisor Jochen Könemann for his guidance, insight and exposing me to problems in discrete optimization I truly found interesting.

Thanks to the readers for their help in making this a successful thesis.

Thanks to the Department of Combinatorics and Optimization for their support and help throughout both my undergraduate and graduate career.

Thanks to the faculty of the department including Joseph Cheriyan, Bill Cunningham, Steven Furino, Jim Geelen, Chris Godsil, Penny Haxell, David Jackson, and Nick Wormald for teaching me the beauty of combinatorics and optimization and pulling me away from the boredom of computer programming.

Thanks to my parents for their support in my pursuit of a higher education.

Contents

List of Figures **xi**

1 Introduction **1**
1.1 Solving real world problems using discrete optimization . . . 1
1.1.1 Solving the k-C1P problem . . . 2
1.2 A natural problem in optimization . . . 3
1.3 A matching problem . . . 3
1.4 Results in this thesis . . . 4
1.5 Outline . . . 4

2 Previous Work **7**
2.1 Overview of Previous Work . . . 7
2.1.1 Minimum Degree Spanning Trees . . . 7
2.1.2 Degree of Minimum Spanning Trees . . . 7
2.1.3 Directed Minimum Degree Spanning Trees . . . 9
2.1.4 Minimum Crossing Spanning Trees . . . 9
2.2 Approximation algorithms for the MDST problem . . . 10
2.2.1 Swapping edges in a spanning tree . . . 10
2.2.2 A local search algorithm for the MDST problem . . . 11
2.2.3 An additive 1-approximation algorithm for the MDST problem . . . 12
2.3 Approximation algorithm for BDMSTs . . . 15

3 MCSTs on Disjoint Sets **21**
3.1 Algorithm . . . 21
3.2 Analysis . . . 25

4 MCSTs on Laminar Sets **27**
4.1 Overview . . . 27
4.2 Background . . . 28
4.2.1 Representing $\mathscr{S}$ . . . 28
4.2.2 Multicommodity flows and trees . . . 29
4.3 The Algorithm . . . 34
4.3.1 Extending multicommodity flows on trees . . . 34
4.3.2 Subroutine for forced multicommodity flows . . . 36
4.3.3 Analysis of the subroutine . . . 38

4.3.4 Final Algorithm 40
4.3.5 Performance 40

5 Minimum Crossing Perfect Matchings **43**
5.1 Overview 43
5.2 Polyhedral viewpoint 43
5.3 A hard matching problem 45

6 Open Problems and Further Research **49**

Bibliography **51**

List of Figures

2.1 Example of a witness set 10
2.2 Example of a good swap 10
2.3 A sequence of good swaps and how it improves degree 13
2.4 Example of how swaps dont interfere 14
2.5 Example of the auxiliary graph G_i 16
2.6 Examples of how to pair off vertices in a tree. 18

3.1 Example of an $\mathscr{S}$-spanning tree 21
3.2 The number of edges required to cross sets of $\mathscr{A}$ in a spanning tree of G 22
3.3 Witness sets for disjoint vertex sets 23
3.4 How a swap is identified for disjoint sets 24
3.5 The induced subgraphs can be disconnected 25

4.1 Graph G with family $\mathscr{S}$ and the laminar tree A 29
4.2 Example of graph G with family $\mathscr{S}$ and the corresponding multicommodity flow problem . 30
4.3 Example of checking edges in reverse delete 33
4.4 How a b-matching can have too many components 35
4.5 Graph G, laminar family $\mathscr{S}$, edges $F \subseteq E$, and the corresponding auxiliary graph B_F 36
4.6 P_{e_i} on the graph B_F. 40

5.1 6-cycle graph, 2 disjoint sets, integral solution to MCPM linear program 44
5.2 6-cycle graph, 2 disjoint sets, half-integral solution to MCPM linear program 44
5.3 A non-integral objective value and dual solution for MCPM linear program 45
5.4 Example of the auxiliary graph for an instance of the 3-dimensional matching problem. . . . 46
5.5 An example of a 3DM and the corresponding MCPM 47

Chapter 1

Introduction

In the real world, there are problems involving finite, countable objects that involve minimizing or maximizing something such as cost, the number of objects used, size of some set, etc., given some constraints on how these objects interact. Discrete optimization is an area of mathematics that deals with many of these problems. This is an area with a rich history of solving some of these problems exactly and in an acceptable timeframe. However, there are many problems in discrete optimization that are considered difficult to solve, either exactly or in a reasonable timeframe. One direction mathematicians have taken for these problems is to find *approximation algorithms*. These algorithms give solutions within a specific timeframe and are proven to be within some value of the optimal solution. Given a problem, an algorithm gives an α-*approximation* for that problem if for any instance of the problem, it gives a solution whose value is within a factor of α of the optimal solution for the instance.

The problems that are considered difficult are divided up into *complexity classes* based on the difficultly of solving or even approximating the problem. The class of problems that can be solved exactly in a reasonable timeframe is known as P. These problems can be solved by an algorithm that runs in polynomial time. The main complexity class discussed in this thesis is NP. These are problems where given a solution to a problem, one can verify if the solution is optimal in polynomial time. P is contained in NP but it is not known if P=NP or not. In general, mathematicians believe that P$\neq$NP.

The problems discussed in this thesis are at least as hard as any problem in NP. The thesis will present approximation algorithms for these problems.

1.1 Solving real world problems using discrete optimization

The summary of the following two real world problems comes from [2].

In computational biology, one of the things the human genome project is attempting to do is to reconstruct the relative position of DNA fragments along the genome, given information on their pairwise overlap. This problem is known as the *physical mapping problem*. For an instance of this problem, there exists a collection of clones and a set of genomic inserts called probes. Each probe defines a single location where a given subset of clones coincide. Given a probe/clone pair, using biological techniques one can determine whether the clone contains the probe as a subsequence. The result of concatenating multiple clones from different parts of the genome and producing a clone that is no longer a simple substring of the chromosome is called chimerism. The new clone is chimeric. The problem is to construct the order that the probes would occur along the original chromosome that is consistent with the given probe-clone incidence matrix. The construction can be

done easily if there is no chimerism. The following is a more formal definition of the problem: Given a probe-clone incidence matrix A, where the rows are indexed by probes and the columns byclones, the entry $a_{ij} = 1$ if and only if probe i occurs in clone j, otherwise $a_{ij} = 0$. Given that there is no chimerism, the problem reduces to finding a permutation of rows such that in each column, the ones are consecutive. This new problem is known as 1-C1P and can be solved in polynomial time. If there is chimerism and each chimeric clone is a concatenation of at most k clones, the problem reduces to finding a permutation of rows such that there are at most k blocks of consecutive ones in each column. This new problem is known as k-C1P. More information on the physical mapping problem can be found in [15].

In network design, given a set of IP routing tables sharing the same host space, one may attempt to reassign the IP addresses to the hosts such that the maximum size of any IP routing table is minimized. This is known as the *interval routing problem*. The following is a more formal definition of the problem: Let $R = \{r_1, \dots, r_n\}$ be a set of n routers and $H = \{h_1, \dots, h_m\}$ be a set of m destination hosts. Each router $r_j \in R$ has outdegree δ_j and a routing table specifying the outedges to use for each host. The problem is to choose the IP addresses of the m hosts and construct the n IP routing tables such that the maximum number of entries used in a table is minimized. It is known that a ρ-approximation algorithm for the k-C1P problem implies a $2\rho \cdot \log m$-approximation algorithm for the interval routing problem. More information on the problem can be found in [1].

1.1.1 Solving the k-C1P problem

The k-C1P problem can be solved by reducing it to another optimization problem. Consider a complete graph $G = (V,E)$, with an m-dimensional cost function $c : E \to \{0,1\}^m$. Given a tour D of G, let the m-dimensional vector $c(D) = \sum_{e \in E(D)} c(e)$. The *vector travelling salesman* (vTSP) problem is to minimize $\|c(D)\|_\infty$ over all tours D of G.

The k-C1P problem can be reduced to the vTSP problem as follows. Let A be the $x \times y$ matrix from the k-C1P problem. For each row of A, there is a vertex in G. For each edge (i, j) in G, let $c(e)$ be the XOR-vector $a_i \, XOR \, a_j = \{a_{i1} \, XOR \, a_{j1}, \dots, a_{iy} \, XOR \, a_{jy}\}$. Let π be the permutation induced by a solution T for the vTSP problem. Let A^π be the matrix that results by applying π to the rows of A. Let $b(A^\pi)$ be the maximum number of blocks of consecutive ones in A^π. Therefore, $b(A^\pi) = \frac{\|c(T)\|_\infty}{2}$.

Given a spanning tree T of G, let the m-dimensional vector $c(T) = \sum_{e \in E(T)} c(e)$. The *vector minimum spanning tree* (vMST) problem is to minimize $\|c(T)\|_\infty$ over all spanning trees T of G. Since Hamming distance obeys the triangle inequality, using Euler Tour shortcutting techniques, a $2r$-approximation for the vTSP problem can be derived from an r-approximation to the vMST problem. After reducing the k-C1P problem on a matrix A to the vTSP problem on a complete graph $G = (V,E)$, the vMST problem on G can be formulated as a minimum crossing spanning tree problem on G.

Let $G = (V,E)$ be an undirected graph and $\mathscr{S}$ be a family of subsets of vertices. An edge *crosses* $S \in \mathscr{S}$ if it contains exactly one endpoint in S. The *minimum crossing spanning tree* is a spanning tree that minimizes the maximum number of edges crossing any single set in $\mathscr{S}$.

Let $V_j = \{v_i \in V | a_{ij} = 0\}$. Let $\mathscr{S} = \{V_1, \dots, V_y\}$. Each column j of A can be viewed as a subset of the vertices of G. Since the cost of an edge (i, j) is just $a_i \, XOR \, a_j$, the lth coordinate of $c(i, j)$ corresponds to the set V_l and is 1 if and only if $(i, j) \in V_l$. Since for any spanning tree T, $c(T) = \sum_{e \in E(T)} c(e)$, the ith coordinate of $c(T)$ is the number of edges of T crossing V_i. Thus, the minimum crossing spanning tree minimizes $\|c(T)\|_\infty$. Given an r-approximation algorithm for the problem of finding a minimum crossing spanning tree, there is an r-approximation algorithm for the vMST problem.

1.2 A natural problem in optimization

Aside from having real world applications, the problem of finding minimum crossing spanning trees it is a generalization of a well-known open problem in graph theory and discrete optimization. Given an undirected graph $G = (V, E)$, one may want to know whether there is a path in G that uses every vertex in V exactly once. Such a path is known as a Hamiltonian path. Determining if G has a Hamiltonian path is in a class of problems called NP-complete.

The set of NP-complete problems is a subset of the problems in NP. For any given problem in NP, there exists a polynomial-time computable reduction that converts instances of the problem into an equivalent instance of a problem that is NP-complete. Thus, given any two NP-complete problems, they are equivalent in the sense that there exist polynomial-time computable reductions that convert instances of one problem into equivalent instances of the other and vice versa. However, since any NP-complete problem is essentially as hard as any problem in NP, there is no known means of solving any NP-complete problem in polynomial time. There are also problems that fall under the category of NP-hard. For an NP-hard problem, there exists a polynomial-time computable reduction that converts any instance of an NP-complete problem into an equivalent instance of the NP-hard problem. NP-hard problems are essentially problems that are at least as difficult as any problem in NP, so there is no known means of solving any NP-hard problem in polynomial time.

There is a reduction from the Hamiltonian path problem to the minimum degree spanning tree problem. Given a graph G, the problem is to find a spanning tree T of G such that the maximum degree of T is minimized. A Hamiltonian path is simply a spanning tree where the maximum vertex degree is 2. No spanning tree can have a lower maximum vertex degree except for the trivial cases where G is the complete graph on 1 or 2 vertices. Since the Hamiltonian path problem is a special case of the minimum degree spanning tree problem, the minimum degree spanning tree problem is NP-hard. However, it is a well studied problem and there exist good algorithms that find spanning trees with close to minimum degree. These algorithms will be discussed in detail later in this thesis. Further generalizations of this problem have also been analyzed including adding edge costs to find such trees of minimum cost, looking at directed graph versions of the problem, and looking at bounds on the vertex degree rather than minimizing the maximum vertex degree.

The minimum degree spanning tree problem itself can be generalized further. Consider a minimum degree spanning tree. Each vertex can be considered as a vertex set of size one. One can consider the "degree" of each set where "degree" refers to the edges with exactly one endpoint in the set. The minimum degree spanning tree problem can be generalized by considering vertex sets that may be arbitrary. An edge is considered to "cross" a set if it has exactly one endpoint in the set. The problem is, given a family of subsets of vertices, to find a spanning tree of G where the maximum number of edges "crossing" any set is minimized (i.e. minimize the maximum "degree" of a set). Note that all the edges crossing a specific set form a cut of the graph. In fact, this new generalized version of the minimum degree spanning tree problem is just the minimum crossing spanning tree problem. For the rest of the thesis, the minimum crossing spanning tree problem will be viewed as a problem over a graph and a family of sets instead of cuts. Since both the Hamiltonian path problem and the minimum degree spanning tree problem are special cases of the minimum crossing spanning tree problem, the minimum crossing spanning tree problem is NP-hard.

1.3 A matching problem

In attempting to approximate minimum crossing spanning trees, one approach involves the use of T-joins, which are a generalization of matchings. Finding matchings of graphs is a well studied problem with many

efficient algorithms. Finding perfect matchings of bipartite graphs in particular is considered an easy problem. Minimum crossing perfect matchings are a generalization of perfect matchings. Given an undirected graph and a family of subsets of vertices of the graph, the minimum crossing perfect matching problem is to find a perfect matching of the graph where the maximum number of edges "crossing" any single set is minimized. It turns out that this is not an easy problem, even for bipartite graphs. This thesis will prove that the problem is in fact NP-hard, even for bipartite graphs.

1.4 Results in this thesis

This thesis will present three main results. Let OPT denote the maximum crossing of a set in a minimum crossing spanning tree.

The first result is an approximation algorithm for the minimum crossing spanning tree problem for the special case where the sets are pairwise disjoint. The algorithm presented will be an extension of an algorithm for minimum degree spanning trees by Fürer and Raghavachari [10]. The following will be proven:

Theorem 1. *There exists a $2 \cdot OPT + 2$-approximation algorithm that runs in polynomial time for approximating minimum crossing spanning trees over a family of pairwise-disjoint subsets of vertices.*

The second result is an approximation algorithm for the minimum crossing spanning tree problem for the special case where the sets form a laminar family. A family of sets is laminar if for any two sets from the family, either one set is contained in the other or the sets are disjoint. The algorithm will be an extension of an algorithm for multicommodity flows on trees by Garg, Vazirani, and Yannakakis [12] and borrow ideas from an approximation algorithm for bounded degree minimum spanning trees [19]. Given bounds b_i for each set S_i in the laminar family, the algorithm will give a spanning tree where each set S_i is crossed at most $\log_{6/5} n \cdot b_i$ times, where n is the number of vertices, or show that no spanning tree satisfies the bounds. A spanning tree satisfying the bounds for each set is a bounded crossing spanning tree. The thesis will show how this algorithm for approximating bounded crossing spanning trees can be applied to approximating minimum crossing spanning trees. Thus the following wil be proven:

Theorem 2. *There is a deterministic $O(\log n)$-approximation algorithm that runs in polynomial time for approximating minimum crossing spanning trees and bounded crossing spanning trees where the family of subsets of vertices is a laminar family.*

The third result is a hardness proof for the minimum crossing perfect matching problem. The proof will reduce an arbitrary instance of the 3-dimensional matching problem, which is NP-complete, to an instance of a special case of the minimum crossing perfect matching problem. The following will be proven:

Theorem 3. *Finding a minimum crossing perfect matching of a graph is NP-hard, even if the graph is bipartite and the family of subsets of vertices is pairwise-disjoint.*

1.5 Outline

Here is an outline of the rest of this thesis.

Chapter 2 will present previous work done on related spanning tree problems. The chapter will present three algorithms in detail. Ideas from these algorithms will be used in constructing algorithms for the minimum crossing spanning tree problem.

Chapter 3 will present the first result of an approximation algorithm for the special case of the minimum crossing spanning tree problem where the vertex sets are pairwise-disjoint.

Chapter 4 will present the second result of an appoximation algorithm for the special case of the minimum crossing spanning tree problem where the sets form a laminar family.

Chapter 5 will look at the minimum crossing perfect matching problem. The main focus of the chapter will be the third result that the problem is NP-hard.

Chapter 6 will outline some open problems related to minimum crossing spanning trees and minimum crossing perfect matchings and potential areas of future research.

Chapter 2

Previous Work

2.1 Overview of Previous Work

Many of the results of approximation algorithms are given using the following notation. Let $f(n)$ and $g(n)$ be positive real-valued functions on n, where n is from the set of nonnegative integers. If there exists constants $c > 0$ and $N \geq 0$ such that $f(n) \leq c \cdot g(n)$ for all $n > N$, then $f(n) = O(g(n))$. If $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$.

2.1.1 Minimum Degree Spanning Trees

Let $G = (V,E)$ be an undirected graph on n vertices. Let H be a subgraph of G. Given a vertex $v \in V$, let $deg_H(v)$ denote the vertex degree of v in H. Let $\Delta(H) = \max_{v \in V} deg_H(v)$ denote the maximum vertex degree of H.

Definition 2.1. Given a graph G, a *minimum degree spanning tree* (MDST) is a spanning tree T of G such that $\Delta(T)$ is minimized (i.e. the maximum vertex degree of T is minimized).

Let T^* be an MDST for a graph G. Let $\Delta^* = \Delta(T^*)$. The problem of finding a Hamiltonian path in a graph G is known to be NP-complete [11]. For a graph G with at least three vertices, G has a Hamiltonian path if and only if G has a MDST T^* where $\Delta(T^*) = 2$. Thus the problem of finding an MDST of G is NP-hard. The MDST problem has been well studied and there exist good approximation algorithms for it.

In [9], Fürer and Raghavachari gave a local search approximation algorithm that outputs a spanning tree with maximum vertex degree at most $b\Delta^* + \lceil \log_b n \rceil$, where $b > 1$ is an arbitrary constant, in polynomial time. The algorithm is discussed in more detail on page 11. In [10], Fürer and Raghavachari presented a more complex approximation algorithm that outputs a spanning tree with degree $\Delta^* + 1$ in polynomial time. This algorithm is outlined on page 15.

Since Fürer and Raghavachari's algorithm gives the best possible approximation, unless P = NP, the focus has shifted to other problems related to finding MDSTs. In many cases, ideas for approximating MDSTs can be applied to other areas as this thesis will show.

2.1.2 Degree of Minimum Spanning Trees

Definition 2.2. Given a graph $G = (V,E)$ and a cost function $c : E \rightarrow \mathbb{Z}^+$ on the edges of G, a *minimum spanning tree* (MST) is a spanning tree T of G such that the total cost of the edges of T is minimized.

MSTs are a well-studied combinatorial structure. As with spanning trees, mathematicians have looked at minimizing the maximum vertex degree of MSTs.

Definition 2.3. Given a graph G and a cost function $c : E \to \mathbb{Z}^+$ on the edges of G, a *minimum degree minimum spanning tree* (MDMST) is a minimum spanning tree T of G such that $\Delta(T)$ is minimized.

In [7], Fischer extended both algorithms by Fürer and Raghavachari for approximating MDSTs to approximate MDMSTs. Let T^* now be an MDMST. Again, let $\Delta^* = \Delta(T^*)$. The first local search algorithm was extended to an algorithm that computes an MST with maximum vertex degree at most $b\Delta^* + \lceil \log_b n \rceil$ in $O(n^{4+\frac{1}{\log b}})$ time. The second more complex algorithm was extended to an algorithm that computes an MST with maximum vertex degree at most $q \cdot (\Delta^* + 1)$, where q is the number of different edge costs, in polynomial time.

Rather than focus only on minimizing the maximum vertex degree of an MST, there has been a lot of research on looking at bounding the degrees of vertices and then finding a spanning tree of minimum cost that satisfies those degree bounds.

Definition 2.4. Given a graph G, a vertex degree bound b, and a cost function $c : E \to \mathbb{Z}^+$ on the edges, a *bounded degree minimum spanning tree* (BDMST) is a spanning tree T where every vertex degree is at most b and the cost of T is minimized. The degree bound on BDMSTs may also be non-uniform, where the degree bound b for every vertex is replaced by a degree bound function $b : V \to \mathbb{Z}^+$ that assigns a degree bound for each vertex of G.

In [19], Ravi et al. gave an approximation algorithm for the BDMST problem. The algorithm starts with a set of edges $F = \emptyset$, finds edges to connect the components of (V, F), and adds them to F by solving a T-join problem. Given a bound b on the degree of the vertices of G, let OPT_b be the minimum cost of a spanning tree where the maximum vertex degree is at most b. The algorithm gives a spanning tree where the maximum vertex degree is $O(b \log \frac{n}{b})$ and the cost is $O(OPT_b \log \frac{n}{b})$ in polynomial time. The algorithm is outlined on page 17. Ravi et al. also generalized the algorithm to handle Steiner trees, generalized Steiner forests, and the node-weighted version of the BDMST problem.

In [16], Könemann and Ravi gave an algorithm, using Lagrangean duality, for finding a BDMST where the vertex degree bounds are uniform. Given constants $\omega > 0$ and $\beta > 1$, the algorithm gives a spanning tree where the maximum vertex degree is at most $(1+\omega)\beta b + \log_\beta n$ and the cost is at most $(1+\frac{1}{w})OPT_b$. The algorithm runs in polynomial time.

In [17], Könemann and Ravi's algorithm was extended to an algorithm for the BDMST problem where the vertex degree bounds were non-uniform. The new algorithm uses Lagrangean duality combined with repeated use of Kruskal's algorithm for solving MSTs. Given a bound b_v on the degree of each vertex $v \in V$, let OPT be the cost of an MST where each vertex v has degree at most b_v. Given constants $\omega > 1$ and $\beta > 1$, the new algorithm gives a spanning tree where the degree of each vertex v is at most $\frac{\omega}{\omega-1}\beta b_v + 2\log_\beta n$ and the cost is at most ωOPT. The running time is $O(mn^5 \log n)$.

In [3], Chandhuri et al. gave a different approach to finding a BDMST. Their algorithm uses the idea of push-relabel used to solve network flow problems by Goldberg in [14]. Given a constant $\beta > 0$, the algorithm gives a spanning tree where the maximum vertex degree is at most $2(1+\beta)b + O(\sqrt{(1+\beta)b})$ and the cost is at most $(1+\frac{1}{\beta})OPT_b$.

In [13], Goemans gave an approximation algorithm to the BDMST problem with the best known bounds. The algorithm uses concepts from matroid and polyhedral theory. The algorithm gives a spanning tree with maximum degree $b+2$ and the cost is at most OPT_b or shows that no spanning tree with maximum vertex degree at most b exists. Since the algorithm outputs a tree of cost at most OPT_b, the algorithm can approximate

the MDMST problem as well. He also conjectured that it is possible to improve the approximation of the maximum vertex degree to $b+1$, similar to the algorithm of Fürer and Raghavachari for the MDST problem.

2.1.3 Directed Minimum Degree Spanning Trees

The case of G being a directed graph has also been studied. An analog of MDSTs for directed graphs exists.

Definition 2.5. Given a directed graph G and a rooted node r, a *directed minimum degree spanning tree* (DMDST) is a directed subgraph T of G such that the underlying graph of T is a spanning tree of the underlying graph of G, there exists a directed path from every node of T to r, and the maximum indegree of a vertex of T is minimized.

Let T^* now be a DMDST and Δ^* denote the maximum indegree of a vertex of T^*. In [8], Fürer and Raghavachari gave a polynomial time algorithm that outputs a DMDST with indegree $O(\Delta^* \log n)$. In [18], Krishnan and Raghavachari extended the local search algorithm of Fürer and Raghavachari used for MDSTs to give an algorithm that outputs a DMDST where the maximum indegree of a vertex is at most $c\Delta^* + \lceil \log_c n \rceil$ for a constant $c > 1$. However, the runtime of this algorithm is quasi-polynomial as it runs in $O(n^{\log_c n + O(1)})$ time.

2.1.4 Minimum Crossing Spanning Trees

For the following, let $G = (V,E)$ be a graph on n vertices, $H = (W,F)$ be a subgraph of G, and $\mathscr{S}$ be a family of subsets of vertices of G, where $\mathscr{S} = \{S_1, \ldots, S_k\}$, $S_i \subseteq V$, $1 \le i \le k$. Given $S \subseteq V$, let $\delta(S) \subseteq E$ denote the set of edges with exactly one endpoint in S (i.e. $\delta(S) = \{e = (u,v) \in E : |\{u,v\} \cap S| = 1\}$). If $e \in \delta(S)$, then *e crosses S*. Let $deg_H(S) = |\delta(S) \cap F|$ (i.e. $deg_H(S)$ is the *degree* of S in H). The notation Δ will be extended for when a family of subsets of vertices $\mathscr{S}$ is defined. Let $\Delta(H) = \max_{1 \le i \le k} deg_H(S_i)$ denote the maximum crossing of H over $\mathscr{S}$.

Definition 2.6. Given a graph G and a family of subsets of vertices $\mathscr{S}$, a *minimum crossing spanning tree* (MCST) is a spanning tree T of G such that $\Delta(T)$ is minimized.

Let T^* be an MCST and $\Delta^* = \Delta(T^*)$. Note that in this thesis, T^* and Δ^* will be used interchangably between the MDST and MCST problem. Let $r = \max_{e \in E} |\{S \in \mathscr{S} : e \in \delta(S)\}|$ (i.e. an edge of G crosses at most r sets). Note that $k = |\mathscr{S}|$.

In [15], Greenberg and Istrail extended Fürer and Raghavachari's local search algorithm for MDSTs to find MCSTs. However, their algorithm is designed for solving the physical mapping problem in computational biology. For solving the general MCST problem, their algorithm gives a spanning tree with a maximum crossing of $O(r\Delta^* + \log n)$ but does not run in polynomial time. The runtime of the algorithm is bounded by $O(k^{\log r})$ iterations which is not polynomial in n.

In [2], Bilò, Goyal, Ravi, and Singh presented the best known polynomial time approximation algorithms for the MCST problem. Starting with $F = \emptyset$, one of the algorithms they present chooses one edge at a time to add to F. The edges are chosen to connect components of (V,F) and minimize $\Delta(F)$ until F is a spanning tree. The algorithm gives a spanning tree with maximum crossing at most $4r(\log n)\Delta^*$ in polynomial time.

[2] also presented a randomized-rounding algorithm for the MCST problem. Given a linear program relaxation for the MCST integer program, the algorithm finds a fractional solution and then rounds the values with probability based on the fractional solution. The algorithm gives a connected subgraph with maximum crossing $O(\Delta^* \log n + \log k)$ with high probability in polynomial time.

2.2 Approximation algorithms for the MDST problem

In this section, two previously known algorithms for the MDST problem will be presented. This section will describe some key ideas that will be useful in later parts of this thesis. The two algorithms are from [9] and [10], where Fürer and Raghavachari gave 2 polynomial time algorithms to approximate the MDST problem. The first will show the key idea of *swapping* edges. The second will show a clever way of swapping edges.

The following lemma will help in calculating a bound on the maximum degree of a spanning tree that is output by the algorithms.

Lemma 2.1. *[10] Let $W \subseteq V$. Let p be the number of components when W is removed from G (i.e. removing the edges of G adjacent to vertices of W leaves $|W| + p$ components). Then* $\Delta^* \geq \left\lceil \frac{|W|+p-1}{|W|} \right\rceil$.

The lemma uses W as a *witness set* for a lower bound on Δ^*.

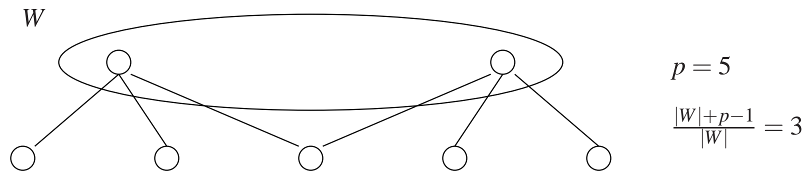


Figure 2.1: Example of a witness set

2.2.1 Swapping edges in a spanning tree

Let T be a spanning tree of G. Consider an edge (u,v) not in T. The subgraph $T+(u,v)$ contains a unique cycle C. By taking any edge $(w,z) \in C$, $T+(u,v)-(w,z)$ is also a spanning tree of G. Let $\langle (u,v),(w,z) \rangle$ denote this *swap* of edges of T.

Let w be a vertex in C. Let k be the degree of w. If $\max\{deg_T(u), deg_T(v)\}+1 < deg_T(w)$, then applying the swap $\langle (u,v),(w,z) \rangle$ to T can reduce the degree of w by one without increasing the degree of another vertex to k or higher. Such a swap is called a *good swap* for w. Applying these good swaps will be the main focus of the following MDST algorithms.

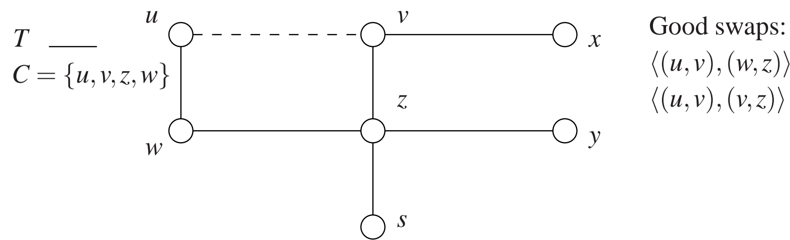


Figure 2.2: Example of a good swap

2.2.2 A local search algorithm for the MDST problem

Given an arbitrary spanning tree T, one could continuously apply good swaps to T until no more good swaps exist.

Definition 2.7. If no good swaps exist, then T is called *locally optimal*.

It is not known how to find a locally optimal tree in polynomial time. However, Fürer and Raghavachari presented a solution to this problem. One could just apply good swaps for any vertex v where $\Delta(T) - \lceil \log_b n \rceil \leq deg_T(v) \leq \Delta(T)$, where $b > 1$ is an arbitrary constant.

Definition 2.8. If there are no good swaps for any vertex v where $\Delta(T) - \lceil \log_b n \rceil \leq deg_T(v) \leq \Delta(T)$, $b > 1$, then T is called *pseudo-optimal*.

In [9], Fürer and Raghavachari's local search algorithm for the MDST problem is to take any spanning tree T of G and apply good swaps for vertices of degree at least $\Delta(T) - \lceil \log_b n \rceil$ until T is pseudo-optimal. Let $\mathscr{S}_d$ denote the vertices with degree at least d in T. Note that $\mathscr{S}_d \subseteq \mathscr{S}_{d-1}$

Lemma 2.2. *[9] Let $b > 1$ be a constant. Given a spanning tree T, there exists $d \in \{\Delta(T) - \lceil \log_b n \rceil + 1, \ldots, \Delta(T)\}$ such that $|\mathscr{S}_{d-1}| \leq b|\mathscr{S}_d|$.*

Proof. Assume to the contrary that for every $d \in \{\Delta(T) - \lceil \log_b n \rceil + 1, \ldots, \Delta(T)\}$ that $|\mathscr{S}_{d-1}| > b|\mathscr{S}_d|$. By repeating this inequality, the result is

$$\begin{aligned} |\mathscr{S}_{\Delta(T)-\lceil \log_b n \rceil}| &> b \cdot |\mathscr{S}_{\Delta(T)-\lceil \log_b n \rceil + 1}| \\ &> b^2 \cdot |\mathscr{S}_{\Delta(T)-\lceil \log_b n \rceil + 2}| \\ &> \ldots \\ &> b^{\lceil \log_b n \rceil} \cdot |\mathscr{S}_{\Delta(T)}| \\ &\geq n \cdot |\mathscr{S}_{\Delta(T)}|. \end{aligned}$$

Since $|\mathscr{S}_{\Delta(T)}| \geq 1$ and $|\mathscr{S}_{\Delta(T)-\lceil \log_b n \rceil}| \leq n$, this is a contradiction. □

This lemma will help in analyzing the maximum vertex degree of a pseudo-optimal spanning tree.

Theorem 2.1. *[9] A pseudo-optimal tree T of G has maximum degree $\Delta(T) < b\Delta^* + \lceil \log_b n \rceil$ for any constant $b > 1$.*

Proof. Given a pseudo-optimal spanning tree T of G and a constant $b > 1$, choose a corresponding d from Lemma 2.2. Therefore $|\mathscr{S}_{d-1}| \leq b|\mathscr{S}_d|$. Removing the vertices of $\mathscr{S}_d$ (i.e. any vertex with degree $\geq d$) from T yields a forest F containing trees $T_1, \ldots, T_p$. Note that F contains p components. Let $(u,v) \in E - T$ be an edge that connects two components of F. $T + (u,v)$ contains a unique cycle C. Since (u,v) connects two components of F, C contains a vertex of $w \in \mathscr{S}_d$. SInce T is pseudo-optimal, both u and v are in $\mathscr{S}_{d-1}$. Therefore, removing any edges adjacent to vertices in $\mathscr{S}_{d-1}$ will split G into at least $p + |\mathscr{S}_d|$ components. By Lemma 2.1,

$$\Delta^* \geq \left\lceil \frac{p + |\mathscr{S}_d| - 1}{|\mathscr{S}_{d-1}|} \right\rceil.$$

Consider T. Each vertex of $\mathscr{S}_d$ has degree at least d and at most $|\mathscr{S}_d| - 1$ edges of T have both endpoints in $\mathscr{S}_d$. Therefore removing $\mathscr{S}_d$ from T leaves

$$p \geq d|\mathscr{S}_d| - 2(|\mathscr{S}_d| - 1)$$

trees and

$$\Delta^* \geq \frac{d|\mathscr{S}_d| - 2(|\mathscr{S}_d| - 1) + |\mathscr{S}_d| - 1}{|S_{d-1}|} = \frac{|\mathscr{S}_d| \cdot (d-1) + 1}{|\mathscr{S}_{d-1}|}.$$

By Lemma 2.2,

$$\Delta^* \geq \frac{|\mathscr{S}_d| \cdot (d-1) + 1}{b|\mathscr{S}_d|} \geq \frac{d-1}{b}.$$

Therefore $d \leq b\Delta^* + 1$ but by the choice of d, $d \geq \Delta(T) - \log_b n + 1$ so

$$\Delta(T) \leq b\Delta^* + \log_b n.$$

□

Theorem 2.2. *[9] A pseudo-optimal spanning tree can be found in polynomial time.*

Proof. The theorem is proved using a potential function argument. Given a vertex v of a spanning tree T, let $\Phi(v) = 3^{deg_T(v)}$ be the potential of v. Let $\Phi(T) = \sum_{v \in V} \Phi(v)$ be the potential of T. Note that

$$n \cdot 3^2 \leq \Phi(T) \leq n \cdot 3^{\Delta(T)}.$$

Each good swap performed to get a pseudo-optimal tree reduces the degree of a vertex $v \in S_d$ where $d \geq \Delta(T) - \lceil \log_b n \rceil + 1$. The reduction in the potential of T after performing a good swap is at least

$$\begin{aligned}
(3^d + 2 \cdot 3^{d-2}) - 3 \cdot 3^{d-1} &= 3 \cdot 3^{d-2} \\
&\geq 3^{\Delta(T) - \lceil \log_b n \rceil - 1} \\
&= \Omega\left(\frac{3^{\Delta(T)}}{n}\right) \\
&= \Omega\left(\frac{\Phi(T)}{n^2}\right).
\end{aligned}$$

Therefore each good swap to get a pseudo-optimal tree reduces the potential by a polynomial factor. After $O(n^2)$ good swaps, the potential is reduced by a constant factor and so the number of good swaps is $O(n^3)$.

□

2.2.3 An additive 1-approximation algorithm for the MDST problem

In [10], Fürer and Raghavachari improved on their local search algorithm. This algorithm will be presented in detail because it will later be extended to give an approximation algorithm for the MCST problem where the sets are disjoint.

Consider a spanning tree T of $G = (V, E)$. Earlier a good swap $\langle (u,v),(w,z) \rangle$ was defined as a swap where $\max\{deg_T(u), deg_T(v)\} + 1 < deg_T(w) = k$. This ensured that the degree of u or v did not become greater than or equal to k. A situation could occur where there exists a swap $\langle (u,v),(w,z) \rangle$ for w where $\max\{deg_T(u), deg_T(v)\} + 1 = deg_T(w)$. However, it may be possible to find a good swap for u (or v) and then the swap $\langle (u,v),(w,z) \rangle$ would be a good swap. Figure 2.3 gives an example.

Let Δ denote $\Delta(T)$. Instead of reducing the degrees of vertices with degree at least $\Delta - \lceil \log n \rceil$, the algorithm will focus on reducing the degree of vertices in $\mathscr{S}_\Delta$. The algorithm will start with all the vertices in $\mathscr{S}_{\Delta-1}$ being marked *bad* and all other vertices being marked *good*. The algorithm will iteratively take T and

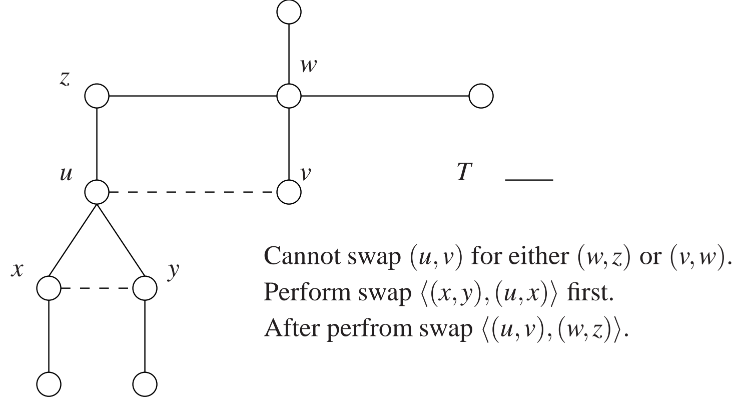


Figure 2.3: A sequence of good swaps and how it improves degree

mark vertices in $\mathscr{S}_{\Delta-1}$ as good if there exists good swaps for them. The definition of good swap is changed in order to use the idea of good vertices. A swap $\langle (u,v),(w,z)\rangle$ for w is good whenever both u and v are good.

Consider the new notion of a good swap $\langle (u,v),(w,z)\rangle$ for $w \in \mathscr{S}_{\Delta-1}$. Before the swap, u and v must be good vertices. Therefore either 1) u has degree at most $\Delta-2$ or 2) u has degree $\Delta-1$ and there is another good swap that can reduce the degree of u. Similarly for v. Therefore there is a sequence of swaps, including $\langle (u,v),(w,z)\rangle$, that can reduce the degree of w without adding any new vertex to $\mathscr{S}_{\Delta}$.

Given a spanning tree T, let $B = \mathscr{S}_{\Delta-1}$ represent the bad vertices and F^0 represent the forest $T \setminus B$ with components $F_1^0,\dots,F_{p_0}^0$. Consider the edges of the form $(u,v) \in E$ where F_u^0, F_v^0 are the components of F^0 containing u and v respectively and $F_u^0 \neq F_v^0$. These edges are precisely the edges that can potentially be in a good swap to reduce the degree of a vertex in B. If such an edge (u,v) is added to T, then $T+(u,v)$ contains a cycle C. Therefore, every bad vertex on C has a good swap using (u,v). Given such an edge (u,v), the algorithm will mark each bad vertex $w \in C$ as good and set $wit(w) = (u,v)$ to track the edge that can be used to reduce the degree of w. There exists at least one bad vertex on C since $u \in F_u^0$, $v \in F_v^0$, $F_u^0 \neq F_v^0$. The set B will then change and a new forest F^1 will represent $T \setminus B$ with components $F_1^1,\dots,F_{p_1}^1$. This process of finding edges between components in the forest is continuously repeated. Instead of recalculating each F^{i+1}, the algorithm will join the F^i components containing a vertex in the cycle C, the new good vertices in C, and the F^i components adjacent to these new good vertices in T into one component of F^{i+1}.

Eventually, there may be a good swap $\langle wit(w),(w,\bar{w})\rangle$, where $(w,\bar{w})$ is an edge of T, for a vertex w of degree Δ. The algorithm will mark w as good and apply swap $\langle wit(w),(w,\bar{w})\rangle$ to reduce the degree of w. If a vertex v in $wit(w)$ has degree at least $\Delta-1$, then v was marked good earlier and there is a good swap $\langle wit(v),(v,\bar{v})\rangle$ that the algorithm will apply, where $(v,\bar{v})$ is an edge of T, to reduce the degree of v. The algorithm will continuously check if $u \in wit(v)$ has degree at least $\Delta-1$, if so then apply a swap using $wit(u)$, and check the degree for the vertices in $wit(u)$, and repeat until T is changed to a new spanning tree with maximum vertex degree at most Δ but with w having degree $\Delta-1$ and no new vertices having degree Δ. If no vertex w of degree Δ is marked good and there is no edge of G with endpoints in different components of the forest F^i, the algorithm will return the spanning tree.

Note that the series of swaps to reduce the degree of w does not conflict with each other. Let $wit(w) = (u,v)$ and i be the iteration where the good swap $\langle wit(w),(w,\bar{w})\rangle$ is identified by the algorithm. u and v are in separate components F_u^i and F_v^i of forest F^i. The edges adjacent to w are not in F^i since w is bad. If

u has degree at least $\Delta-1$, then there is a good swap $\langle wit(u),(u,\bar{u})\rangle$ where edges $wit(u)$ and $(u,\bar{u})$ are in F^i_u. Clearly, swap $\langle wit(u),(u,\bar{u})\rangle$ will not affect swap $\langle wit(w),(w,\bar{w})\rangle$ since $wit(w)$ and $(w,\bar{w})$ are not in F^i_u. Similar for v. If both u and v have degree less than $\Delta-1$, then no further swaps are necessary. The same logic can be inductively applied to u and v. See Figure 2.4 for an example.

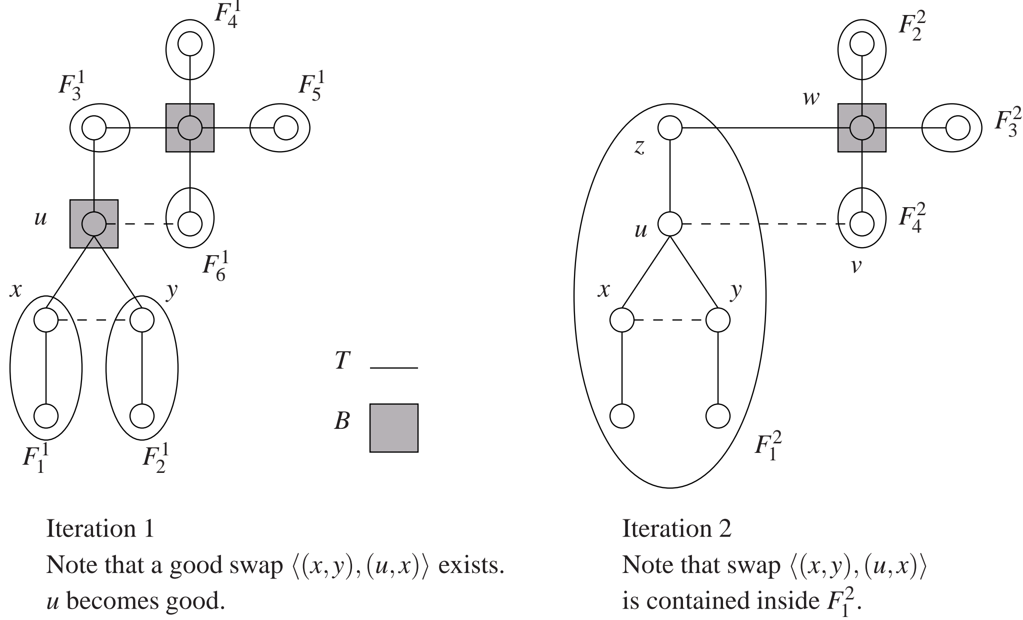


Figure 2.4: Example of how swaps dont interfere

Algorithm 2.1 outlines this algorithm formally.

Theorem 2.3. *[10] Algorithm 2.1 outputs a spanning tree with maximum vertex degree at most* Δ^*+1.

Proof. Let T be the spanning tree output by Algorithm 2.1. Let $B\subseteq\mathscr{S}_{\Delta-1}$ be the bad vertices remaining at the end of the algorithm. Let $F=\{T_1,\ldots,T_p\}$ be the components of T after removing the vertices in B.

Since there are p components in F and $|B|$ bad vertices, any spanning tree of G requires at least $p+|B|-1$ edges. Note that by the termination conditions of the while loop, there is no edge between components of F in G. Therefore, any spanning tree of G requires an edge connecting each component of F to a vertex in B. Each vertex in B has degree at least $\Delta-1$. In any spanning tree of G, at most $(|B|-1)$ edges can be used to connect vertices of B together. Each of these edges is counted twice in the degrees of the bad vertices. Therefore,

$$p\geq|B|(\Delta-1)-2(|B|-1).$$

Therefore, any spanning tree of G has at least

$$p+|B|-1\geq|B|(\Delta-1)-2(|B|-1)+|B|-1=|B|(\Delta-2)+1$$

edges. All of these edges are adjacent to vertices in B. Therefore, a vertex in B has degree at least

$$\left\lceil\frac{|B|(\Delta-2)+1}{|B|}\right\rceil\geq\Delta-1$$

Algorithm 2.1 Computing a spanning tree with maximum vertex degree at most Δ^*+1

1: Given a connected graph $G=(V,E)$.
2: Find a spanning tree T of G.
3: Let Δ be the maximum degree of a vertex in T.
4: Mark all $v \in \mathscr{S}_{\Delta-1}$ as bad.
5: Let F^0 be the components of $T \setminus \mathscr{S}_{\Delta-1}$.
6: $i=0$.
7: **while** $\exists(u,v)\in E$ where $F_u^i \neq F_v^i$ **do**
8: Find all bad vertices in the cycle C_{uv} in $T+uv$ and mark them as good.
9: **if** $\exists$ good vertex w of degree Δ **then**
10: Reduce the degree of w by swapping in $wit(w)$ and iteratively performing all other necessary swaps, and go to step 3.
11: **end if**
12: Obtain F^{i+1} from F^i by joining the F^i-components and good vertices along cycle C_{uv}.
13: $i=i+1$.
14: **end while**
15: Return T.

in T. This is a lower bound for Δ^*. Therefore, $\Delta \leq \Delta^*+1$. □

Theorem 2.4. *[10] Algorithm 2.1 will output a spanning tree in polynomial time.*

Proof. The sum of the degrees of the vertices of T is $2n-2$. Therefore the number of vertices of degree Δ is $O(\frac{n}{\Delta})$. Let a *phase* of the algorithm be the steps taken to remove a vertex from $\mathscr{S}_\Delta$. There are $O(\frac{n}{\Delta})$ phases required to remove all the vertices from $\mathscr{S}_\Delta$. Therefore, there are

$$O\left(\sum_{k=2}^{n}\frac{n}{k}\right)=O(n\log n)$$

phases in total. Each phase can be implemented in nearly linear time using Tarjan's fast disjoint set union-find algorithm for maintaining connected components [6]. The algortihm runs in $O(mn\alpha(m,n)\log n)$, which is polynomial on n, where α is the inverse Ackerman function. □

2.3 Approximation algorithm for BDMSTs

In this section, an approximation algorithm for the BDMST problem will be presented. This section will describe more key ideas that will be useful in later parts of this thesis. The algorithm presented here is a simplification of an algorithm in [19].

The algorithm is constructive. F will denote the potential edges chosen by the algorithm for a spanning tree of G. The algorithm starts with $F=\emptyset$. For each iteration of the algorithm, a set of edges that connects components of (V,F) are chosen and added to F. In each iteration, the edges will be chosen in such a way that each vertex will have their degree increase by at most b, the cost of the edges added is at most OPT_b, and at the same time the number of components of (V,F) will be reduced by a constant factor. After $O(\log n)$ iterations, the graph (V,F) will be connected and will satisfy the approximation guarantees for the cost and the maximum vertex degree.

The algorithm will use a well-known combinatorial structure on graphs.

Definition 2.9. Given a graph $G = (V, E)$ and a set of vertices $T \subseteq V$, a *T-join* of G is a set of edges $M \subseteq E$ such that the degree of every vertex of T is odd and the degree of every other vertex of V is even.

It is well known that T-joins can be found efficiently [5]. T-joins are useful because of the following lemma.

Lemma 2.3. *[19] Given any T-join J, J contains $|T|/2$ edge-disjoint paths. The endpoints of these paths results in a pairing of the T vertices in J.*

Proof. $v \in T$ is an odd vertex degree in J. The sum of the vertex degrees must be even in the component of J containing v. Therefore there must be some vertex $w \in T$ connected to v in J. Take the shortest path P from any $v \in T$ to another vertex $w \in T$. Remove the edges of P from J to get J'. Remove v and w from T to get T'. v and w are paired off together. J' is a T'-join. By induction, J' conatins $|T'|/2$ edge-disjoint paths such that the endpoints of these paths are precisely the vertices in T'. □

The algorithm will attempt to find a T-join in each iteration. For each component C of (V, F), exactly one vertex in C will be in T. By using a T-join, the algorithm will pair-off the components of (V, F) with a path connecting each pair of components. From each path, the edges that connect different components of (V, F) will be added to F.

During each iteration i, the algorithm will construct an auxiliary graph G_i. Let $\mathscr{C}$ be the collection of components of (V, F). G_i is essentially the same graph as G except that any edge of G that has endpoints in the same component of $\mathscr{C}$ will have a cost of zero. For each component $C \in \mathscr{C}$, an arbitrary vertex in C will be assigned to T. For the case where $|\mathscr{C}|$ is odd, a dummy vertex z is added to G_i with zero cost edges connecting z and the vertices representing components in $\mathscr{C}$. z will also be added to T. By adding z to T, this will ensure that the size of T is even, making the T-join possible. An example of the auxiliary graph G_i is shown in Figure 2.5.

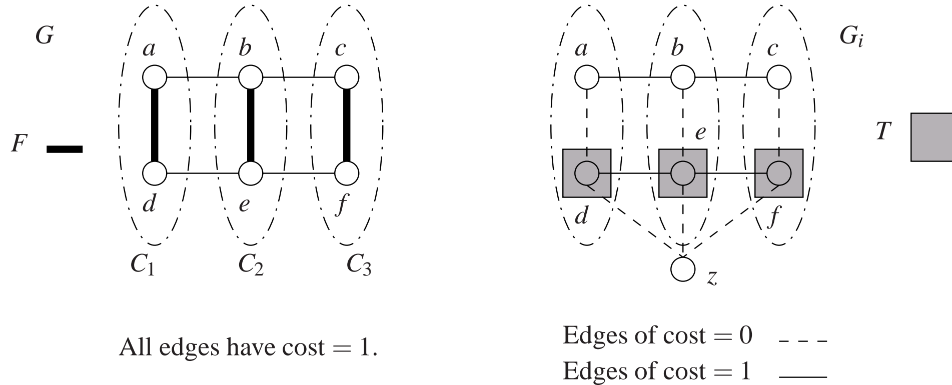


Figure 2.5: Example of the auxiliary graph G_i

Given the graph G_i, the algorithm will find a degree bounded T-join J of minimum cost on G_i, where each vertex of J must have degree at most b. The algorithm is outlined formally in Algorithm 2.2.

Similar to how the T vertices in J can be paired off by edge-disjoint paths contained in J, the following is true for trees.

Algorithm 2.2 Finding an approximate minimum cost spanning tree with bounded degree

```
Given G = (V,E) on n vertices, costs c : E → Z^+, degree bound b.
F = ∅, i = 1
while There are more than O(b) components in F do
    Let 𝒞 = {C_1, ..., C_p} be the set of connected components of (V,F).
    Construct auxiliary graph G_i = (V_i, E_i) with costs c' : E_i → Z^+ as follows.
    Let G_i = G. Let T = ∅.
    for Each e = (u,v) ∈ E do
        if u ∈ C_u, v ∈ C_v, C_u ≠ C_v then
            Set c'(e) = c(e).
        else
            Set c'(e) = 0.
        end if
    end for
    for Each C_j ∈ 𝒞 do
        Add an arbitrary vertex of C_j to T.
    end for
    if |𝒞| is odd then
        Add z to V_i.
        Add edge e = (v,z) to E_i with cost c(e) = 0, ∀v ∈ T.
        Add z to T.
    end if
    Find a degree bounded T-join J of minimum cost on G_i.
    From J, add the corresponding edges of G to F.
    i = i + 1.
end while
Contract the components of 𝒞 and find an MST M of the resulting graph.
F = F ∪ M.
Output an MST of F.
```

Claim 2.1. *[19] Let Q be a tree, $S \subseteq V(Q)$, $|S|$ is even. There is a pairing of the vertices in S such that the unique uv-paths between each pair (u,v) are edge-disjoint.*

Proof. If $|S| = 2$, there is a unique path in Q with endpoints between the two vertices in S. If $|S| > 2$, root Q at an arbitrary vertex r. Let v be the vertex furthest from r such that the subtree R rooted at v contains at least 2 vertices from S. Any path between two vertices from S that is contained in R must use v.

If $v \in S$ and R contains exactly two vertices v and w from S, v and w can be paired off in R by a path P. P is completely contained in R. Taking Q and removing the subtree R and the edge connecting R to the rest of Q gives a new tree Q' with special vertices S' where $|S'| = |S| - 2$. P is disjoint from Q'.

If $|S \cap R| \geq 3$ or $|S \cap R| = 2$ and $v \notin S$, then consider two vertices $u,w \in S \cap R$ paired off by a path P. Taking Q and removing P, except the vertex v, along with the subtrees rooted at u and w, gives a new tree Q' with special vertices S' where $|S'| = |S| - 2$. P is edge-disjoint from Q'.

Figure 2.6 shows some examples of how this induction on the size of the tree is done. By induction on the number of special vertices, Q' contains a pairing of the vertices in S' such that the paths between each pair are edge-disjoint. □

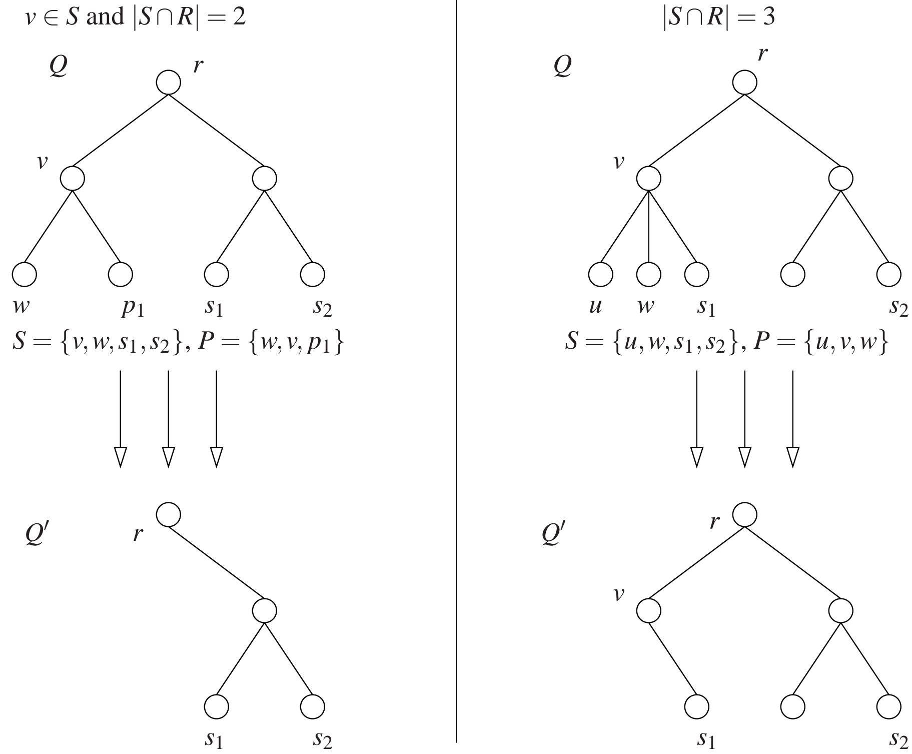


Figure 2.6: Examples of how to pair off vertices in a tree.

The claim will help prove the following lemma.

Lemma 2.4. *Let T be given by iteration i of Algorithm 2.2. There exists a T-join of G_i of cost at most OPT_b that has no vertex with degree more than b.*

Proof. Assume that (V, F) has an even number of components. The odd case is similar. Consider a BDMST Q of G. Let the vertices of T be special vertices in Q. By Claim 2.1, there is a pairing of the T vertices using paths in Q. Let J be the union of these paths. Since each path has T vertices for endpoints and each vertex of T is the endpoint of exactly one path, J is a T-join of G_i. The vertices of Q have degree at most b. Therefore the vertices of J have degree at most b. In G_i, edges either have the same cost as they have in G or they have no cost. Therefore, since the total cost of Q is OPT_b, the total cost of J is at most OPT_b. □

Now it is proven that finding a T-join in Algorithm 2.2 is possible for every iteration. Using this, one can prove the efficiency of the algorithm.

Definition 2.10. Given a graph $G = (V, E)$ and a vertex degree bound function $b : V \to \mathbb{Z}^+$, a b-factor is a set of edges $M \subseteq E$ such that $deg_M(v) = b_v$, $\forall v \in V$.

Lemma 2.5. *A degree bounded T-join of minimum cost can be found in polynomial time.*

Proof. Let $G = (V,E)$ be a graph with edge costs $c : E \rightarrow \mathbb{Z}^+$ and vertex degree bound d. Let $T \subseteq V$. The T-join J is constructed using b-factors. The bound function $b : V \rightarrow \mathbb{Z}^+$ is set such that b_v is either d or $d-1$ where $v \in T$ if and only if b_v is odd. For each vertex $v \in V$, $d/2$ loops are added where v is the endpoint. Set the cost of each loop to zero. A minimum cost b-factor M on G can be found in polynomial time [5]. Let J be M without any loops. The total cost of J is equal to the total cost of M. Removing the loops maintains the parity of each vertex degree. Since M satisfies the degree constraints, J is a degree bounded T-join. If there is another T-join K of lower cost, then loops of zero cost can be added to each vertex v of K until the vertex degree of v is equal to b_v. This would give a b-factor will lower cost than M, a contradiction. Thus, J is a degree bounded T-join of minimum cost. Since M is found in polynomial time, J is found in polynomial time. □

Lemma 2.6. *[19] Algorithm 2.2 finishes in polynomial time after $O(\log \frac{n}{b})$ iterations.*

Proof. Note the T-join J found in each iteration of Algorithm 2.2, where T contains a vertex from each component of $\mathscr{C}$ (and possibly z). By Lemma 2.3, finding J results in a pairing of the components of $\mathscr{C}$. Each pair of components is connected by a path. From each path, the edges connecting components of $\mathscr{C}$ are added to F. After adding these edges, F will contain a path between each pair of components of $\mathscr{C}$. Thus, the number of components in (V,F) will decrease by a constant factor during each iteration. Since the algorithm starts with n components and stops to compute an MST when the number of connected components is $O(b)$, there are $O(\log \frac{n}{b})$ iterations. By Lemma 2.5, the T-join J in each iteration of Algorithm 2.2 can be solved in polynomial time. Thus, the algorithm runs in polynomial time. □

Lemma 2.7. *[19] The maximum degree of a vertex in the spanning tree output by Algorithm 2.2 is $O(b \log \frac{n}{b})$.*

Proof. In each iteration, edges are added to F only after finding the T-join J. The vertices of J have degree at most b. Only edges of J that connect components of $\mathscr{C}$ are added to F. Thus, in each iteration, the degree of the vertices in (V,F) increase by at most b. By Lemma 2.6, Algorithm 2.2 finishes after $O(\log \frac{n}{b})$ iterations. At the end of Algorithm 2.2, each vertex of F has degree at most $O(b \log \frac{n}{b})$. The same holds for any spanning tree of F at the end of the algorithm. □

Theorem 2.5. *[19] Algorithm 2.2 outputs a spanning tree that has maximum vertex degree $O(b \log \frac{n}{b})$, total cost $O(OPT_b \log \frac{n}{b})$, and runs in polynomial time.*

Proof. By Lemma 2.4, the total cost of the edges added to F during each iteration is at most OPT_b. By Lemma 2.6, there are $O(\log \frac{n}{b})$ iterations. At the second last step of the algorithm, the edges added to F form an MST of $G/\mathscr{C}$, the graph G where each component of $\mathscr{C}$ is contracted to a single vertex. The cost of the edges from the MST added is at most OPT_b. Therefore, the cost of F is $O(OPT_b \log \frac{n}{b})$. An MST of F will not have more cost so the total cost of the spanning tree output by Algorithm 2.2 is $O(OPT_b \log \frac{n}{b})$. The maximum degree is given by Lemma 2.7. The runtime is given by Lemma 2.6. □

Chapter 3

MCSTs on Disjoint Sets

3.1 Algorithm

Let $G = (V, E)$ be a graph, $\mathscr{S} = \{S_1, \ldots, S_k\}$ be a family of subsets of vertices, and T^* be an MCST. Let $\Delta^* = \Delta(T^*)$. This chapter will present an approximation algorithm for the MCST problem where the sets of $\mathscr{S}$ are pairwise disjoint. This algorithm will use similar ideas from Algorithm 2.1. Instead of starting with any arbitrary spanning tree as in Algorithm 2.1, the algortihm will focus on a special type of spanning tree.

Definition 3.1. Let $G = (V, E)$ be a graph and $S \subseteq V$. Let $G[S] = (S, E')$ be the subgraph of G with vertices S and edges E' which are the edges of G with both endpoints in S. $G[S]$ is the subgraph of G *induced* by S.

Definition 3.2. Let G be a connected graph. Let $\mathscr{S} = \{S_1, \ldots, S_k\}$ be a family of pairwise disjoint sets of vertices of G. Let T be a spanning tree of G. T is an $\mathscr{S}$-spanning tree of G if both $T[S_i]$ and $G[S_i]$ have the same number of components for each $S_i \in \mathscr{S}$.

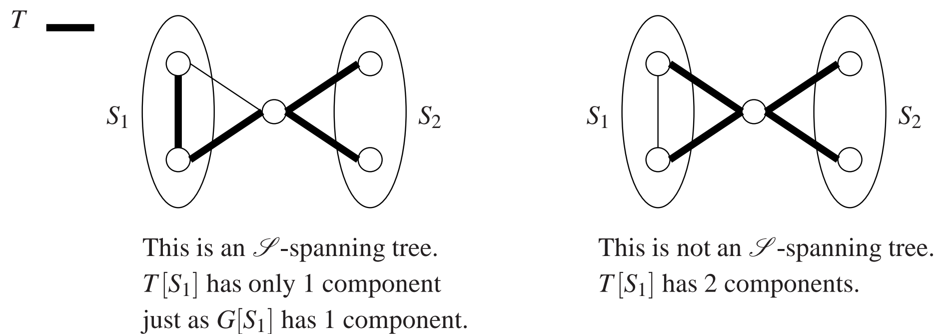


Figure 3.1: Example of an $\mathscr{S}$-spanning tree

$\mathscr{S}$-spanning trees are easy to find by simply finding a maximal spanning forest on each vertex set in $\mathscr{S}$ and then extending that to a spanning tree of G. $\mathscr{S}$-spanning trees are useful due to the following lemma.

Lemma 3.1. *Let G be a connected graph. Let $\mathscr{S} = \{S_1, \ldots, S_k\}$ be a family of pairwise disjoint vertex sets of G. If T is a spanning tree of G, then there exists an $\mathscr{S}$-spanning tree T' where $\Delta(T') \leq \Delta(T)$.*

Proof. Let T be a spanning tree of G which is not an $\mathscr{S}$-spanning tree. There is a set $S_i \in \mathscr{S}$ where $G[S_i]$ has less components than $T[S_i]$. Therefore there is an edge e in $G[S_i]$ which can connect two components in $T[S_i]$. The graph $T+e$ has a cycle C containing e. C has two edges f and g crossing S_i since e is connecting two components of $T[S_i]$. Therefore $T' = T+e-f$ is a spanning tree of G where $\Delta(T') \leq \Delta(T)$. □

From the above lemma, we can conclude that focusing on $\mathscr{S}$-spanning trees is sufficient. This is equivalent to contracting each component of $G[S_i]$, $S_i \in \mathscr{S}$ into a single vertex. The next lemma will provide the machinery to find a bound on the results of the algorithm. Using this lemma, an analogous result to Lemma 2.1 about witness sets for pairwise-disjoint sets can be derived. Given a spanning tree $T = (V,F)$ of $G = (V,E)$ and $S \subseteq V$, let $\delta_T(S) = \{e = uv \in F : |\{u,v\} \cap S| = 1\}$.

Lemma 3.2. *Let $G = (V,E)$ be a connected graph. Let $\mathscr{S} = \{S_1,\ldots,S_k\}$ be a family of pairwise disjoint sets of vertices of G. Let T be an $\mathscr{S}$-spanning tree of G. Let $\mathscr{A} \subseteq \mathscr{S}$. Let $F = \{F_1,\ldots,F_q\}$ be the components of $T \setminus \bigcup_{A\in\mathscr{A}} A$ (the tree T with any vertices in a set in $\mathscr{A}$ removed along with any adjacent edges). Let $R = \{e \in \delta_T(A) : A \in \mathscr{A}\}$. If there is no edge $(u,v) \in E$ such that $u \in F_i, v \in F_j, i \neq j$, then every spanning tree of G must have at least $|R|$ edges crossing sets in $\mathscr{A}$.*

Proof. Let $r = |R|$ and $\mathscr{A} = \{S_{i_1},\ldots,S_{i_\tau}\}$. Since T is a spanning tree, removing r edges from T yields $r+1$ components. Let T' be a spanning tree of G with $r' < r$ edges crossing sets in $\mathscr{A}$. Then removing the edges crossing sets in $\mathscr{A}$ leaves $r'+1 < r+1$ components. Suppose that $T[S_{i_1}] \cup \ldots \cup T[S_{i_\tau}]$ has $m \geq 0$ components. Since T is an $\mathscr{S}$-spanning tree, then $T'[S_{i_1}] \cup \ldots \cup T'[S_{i_\tau}]$ has $m' \geq m$ components. Thus the components remaining in $T' \setminus \bigcup_{A\in\mathscr{A}} A$ are $F' = \{F'_1,\ldots,F'_{q'}\}$ where

$$q' = r' + 1 - m' < r + 1 - m = q.$$

However, since there is no edge $(u,v) \in E(G)$ such that $u \in F_i, v \in F_j, i \neq j$, $G[V \setminus \bigcup_{A\in\mathscr{A}} A]$ has q components.

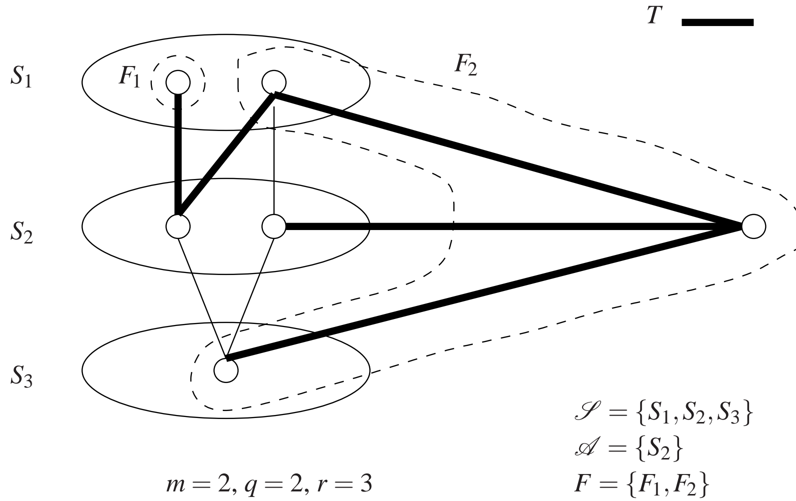


Figure 3.2: The number of edges required to cross sets of $\mathscr{A}$ in a spanning tree of G

Therefore $q' \geq q$, contradicting our previous result. □

From this lemma we get a lower bound for Δ^*. The following corollary is the version of Lemma 2.1 about witness sets for disjoint vertex sets instead of vertices.

Corollary 3.1. *Let $\mathscr{S}$ be a collection of disjoint sets of vertices of a connected graph G. Let T be an $\mathscr{S}$-spanning tree of G. Let $\mathscr{A} \subseteq \mathscr{S}$ such that $T \setminus \bigcup_{A \in \mathscr{A}} A$ has components $F = \{F_1, ..., F_q\}$. Let $R = \{e \in \delta_T(A) : A \in \mathscr{A}\}$. If there is no edge $(u,v) \in E(G)$ such that $u \in F_i, v \in F_j, i \neq j$, then*

$$\Delta^* \geq \left\lceil \frac{|R|}{|\mathscr{A}|} \right\rceil.$$

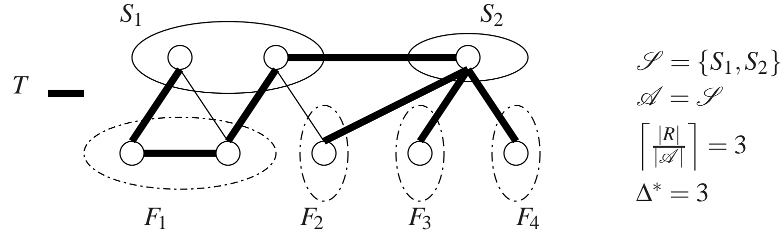


Figure 3.3: Witness sets for disjoint vertex sets

The algorithm to be presented will start with an $\mathscr{S}$-spanning tree T and perform edge swaps to reduce the number of edges crossing a set, similar to Algorithm 2.1.

Let Δ denote $\Delta(T)$. Let $\mathscr{S}_i = \{S \in \mathscr{S} : deg_T(S) \geq i\}$. Note that $\mathscr{S}_i \subseteq \mathscr{S}_{i-1}$. This notation for MCSTs is analogous to the use of $\mathscr{S}_i$ for vertices in MDSTs. Similar to Algorithm 2.1, the focus of the algorithm presented in this chapter will be on reducing the size of $\mathscr{S}_\Delta$.

The algorithm initializes with all the sets in $\mathscr{S}_{\Delta-1}$ as bad and any other set good. Any edge of G crossing a bad set is considered bad (i.e. e is bad if $e \in \delta(S_i)$ where $S_i \in \mathscr{S}$ is bad). All other edges of G are considered good.

Just like swaps for MDSTs, an edge (u,v) will be added to T and a bad edge (w,z) from the unique cycle of $T + (u,v)$ will be removed. Consider the swap $\langle (u,v),(w,z) \rangle$. Let $w \in S_w \in \mathscr{S}$, $z \notin S_w$, where S_w is a bad set. This would make (w,z) a bad edge. The swap is good for S_w if edge (u,v) is good. When applying the good swap $\langle (u,v),(w,z) \rangle$ for S_w, S_w becomes good and so do all edges in $\delta(S_w)$. If $z \in S_z \in \mathscr{S}$ and S_z is bad, S_z becomes good as well as the edges in $\delta(S_z)$.

Applying a good swap $\langle (u,v),(w,z) \rangle$ for S_w to T, where $w \in S_w \in \mathscr{S}$ and $z \notin S_w$ (i.e. (w,z) crosses S_w), will reduce the degree of S_w in T. Before the swap, (u,v) is a good edge. Therefore either 1) (u,v) does not cross any set or 2) (u,v) crosses only good sets. For the second case, either each good set has degree at most $\Delta - 2$, or it has degree $\Delta - 1$ but there is some other good swap that can reduce its degree.

Given an $\mathscr{S}$-spanning tree T, let $\mathscr{B} = \mathscr{S}_{\Delta-1}$ represent the initial bad sets and F^0 represent the forest when the vertices in bad sets are removed from T along with any adjacent edges. Let $F_1^0, \ldots, F_{p_0}^0$ denote the components of F^0. Any edge of the form $(u,v) \in E$, where F_u^0, F_v^0 are the components of F^0 containing u and v respectively and $F_u^0 \neq F_v^0$, is a good edge that can potentially be in a good swap to reduce the degree of a set in $\mathscr{S}_{\Delta-1}$. If such an edge (u,v) is added to T, then $T + (u,v)$ contains a cycle C. If a bad set S_w has a vertex on C, then there is an edge of C crossing S_w. Therefore, every bad set that has a vertex in C has a good swap using (u,v). Given such an edge (u,v), the algorithm will mark each bad set S_w containing a vertex of C as good and set $wit(S_w) = (u,v)$ to track the edge that can be used to reduce the degree of S_w.

There exists at least one bad set containing a vertex of C since $u \in F_u^0$, $v \in F_v^0$, $F_u^0 \neq F_v^0$. The set $\mathscr{B}$ will then change and a new forest F^1 will represent $T \setminus (\cup_{S \in \mathscr{B}} S)$ with components $F_1^1, \ldots, F_{p_1}^1$. This process of finding edges between components in the forest is repeated. Instead of recalculating each F^{i+1}, the algorithm will join the F^i components containing a vertex in the cycle C, the new good sets containing vertices in C, and the F^i components adjacent to the new good sets in T into one component of F^{i+1}. Figure 3.4 shows an example of this process of finding F^i.

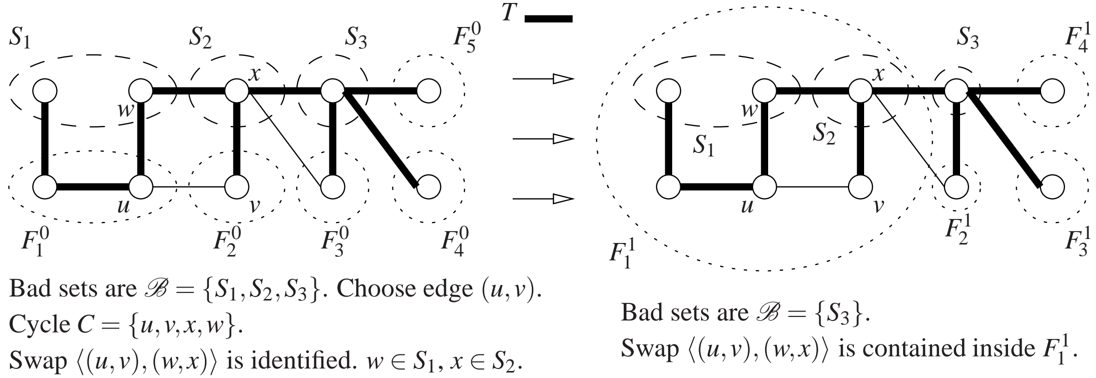


Figure 3.4: How a swap is identified for disjoint sets

Eventually, similar to Algorithm 2.1, either a set of degree Δ is marked good or there are no edges connecting different components of the forest F^i. If a set $S_w \in \mathscr{S}_\Delta$ is marked good, then there is a good swap $\langle wit(S_w), (w,\bar{w}) \rangle$, where $(w,\bar{w}) \in \delta(S_w)$ is an edge of T, that the algorithm will apply to reduce the degree of S_w. However, if $wit(S_w)$ crosses a set $S_v \in \mathscr{S}_{\Delta-1}$, then further swaps are required to ensure no new sets are added to $\mathscr{S}_\Delta$. Since the set S_v would have to be good in order to have a good swap with $wit(S_w)$, there must be a good swap for S_v that the algortihm has already identified. The algorithm will check if S_w has a $wit(S_w)$ value, and if so then apply a good swap using $wit(S_w)$ to reduce the degree of S_w. The algorithm will then repeat the previous step on the sets containing the vertices of $wit(S_w)$. This process is repeated until T is changed to a new $\mathscr{S}$-spanning tree with any set of $\mathscr{S}$ having maximum degree at most Δ but with S_w having degree $\Delta - 1$ and no new set of $\mathscr{S}$ having degree Δ. If no set of $\mathscr{S}_\Delta$ is marked good and there is no edge of G with endpoints in different components of the forest F^i, the algorithm will return the $\mathscr{S}$-spanning tree and the remaining bad sets will be a witness set that gives a certificate of the quality of the solution.

Note that as with Algorithm 2.1, the series of swaps to reduce the degree of S_w does not conflict with each other. Let $wit(S_w) = (u,v)$ and i be the iteration where the good swap $\langle wit(S_w), (w,\bar{w}) \rangle$, $(w,\bar{w}) \in \delta(S_w)$, is identified by the algorithm. u and v are in separate components F_u^i and F_v^i of forest F^i. The edges crossing S_w are not in F_i since S_w is bad. If u is in a set S_u of degree $\Delta - 1$, then there is a good swap $\langle wit(S_u), (u,\bar{u}) \rangle$, $(u,\bar{u}) \in \delta(S_u)$, identified at iteration $j < i$. Note that edges $wit(S_u)$ and $(u,\bar{u})$ are in F_u^{j+1} and since $j+1 \leq i$, $F_u^{j+1} \subseteq F_u^i$. $wit(S_w)$ and $(w,\bar{w})$ are not in F_u^i since the swap $\langle wit(S_w), (w,\bar{w}) \rangle$ was identified at iteration i. Therfore, swap $\langle wit(S_u), (u,\bar{u}) \rangle$ will not affect swap $\langle wit(S_w), (w,\bar{w}) \rangle$. Similarly, the same argument holds for for v. Swaps for u and v do not interfere with each other since $F_u^i \neq F_v^i$. The same logic can be inductively applied to S_u with the subgraph F_u^i and S_v with the subgraph F_v^i. Note that $S_u \neq S_v$ since T is an $\mathscr{S}$-spanning tree. Refering back to Figure 3.4, note how the identified swap for F^i is contained in a component of F^{i+1}.

Algorithm 3.1 outlines this algorithm formally.

Algorithm 3.1 Algorithm to find an MCST given disjoint sets of vertices

1: Given a connected graph $G = (V, E)$.
2: Find any $\mathscr{S}$-spanning tree T of G.
3: Mark all sets in $\mathscr{S}_{\Delta-1}$ as bad and let $F^0 = T \setminus (\bigcup_{S \in \mathscr{S}_{\Delta-1}} S)$ with components $F_1^0, \ldots, F_{p_0}^0$.
4: $i = 0$.
5: **while** $\exists(u,v) \in E$ where $F_u^i \neq F_v^i$ **do**
6: Find all bad sets which contain a vertex in the cycle C_{uv} in $T + (u,v)$ and mark them as good.
7: **if** $\exists$ good set $S_w \in \mathscr{S}_\Delta$ **then**
8: Reduce the degree of S_w by swapping in $wit(S_w)$ and iteratively performing all other necessary swaps, and go to step 3.
9: **end if**
10: Obtain F^{i+1} from F^i by joining the F^i-components and good sets of $\mathscr{S}$ along the cycle C_{uv}.
11: $i = i + 1$.
12: **end while**
13: Return tree T.

3.2 Analysis

Theorem 3.1. *Algorithm 3.1 returns a tree where $\Delta \leq 2\Delta^* + 2$.*

Proof. Let T be the tree computed by Algorithm 3.1. If T_α was the starting tree of the algorithm, note that T is an $\mathscr{S}$-spanning tree since none of the edges in $T_\alpha[S_i], 1 \leq i \leq p$ have been removed. Let $\mathscr{B} \subseteq \mathscr{S}_{\Delta-1}$ be the collection of bad sets that remain after the algorithm ends. Let $\mathscr{F} = F_1, ..., F_p$ be the set of trees that remain after removing all the vertices in the bad sets in $\mathscr{B}$ from T. The while loop of the algorithm ensures that G has no edges with endpoints in different trees of $\mathscr{F}$. The number of edges crossing the sets of $\mathscr{B}$ is:

$$C \geq |\mathscr{B}|(\Delta - 1) - d$$

where d is the number of edges crossing two sets in $\mathscr{B}$. Thus $d \leq (1/2)|\mathscr{B}|(\Delta)$. Note that in the MDST case, $d \leq |\mathscr{B}| - 1$. Here a similar assumption is not possible since each induced subgraph $T[S_i]$ on a set $S_i \in \mathscr{S}$ may be disconnected. In general, the number of double counted edges is not bounded by $|\mathscr{B}| + c$ for any constant c.

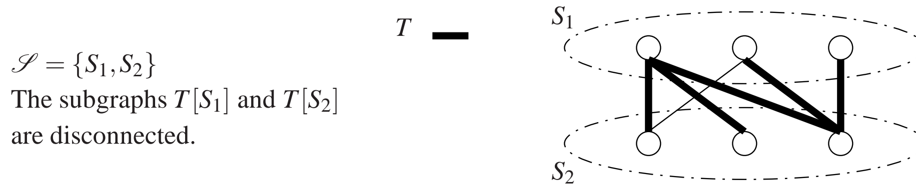


Figure 3.5: The induced subgraphs can be disconnected

Therefore,

$$C \geq (1/2)|\mathscr{B}|\Delta - |\mathscr{B}|.$$

We also know that all these edges are incident to vertices in the sets of $\mathscr{B}$. By Corollary 3.1, one of the sets

in $\mathscr{B}$ has at least

$$\left\lceil \frac{C}{|\mathscr{B}|} \right\rceil \geq \left\lceil \frac{(1/2)|\mathscr{B}|\Delta - |\mathscr{B}|}{|\mathscr{B}|} \right\rceil \geq \frac{\Delta - 2}{2}$$

edges crossing it in any spanning tree of G. Therefore,

$$\Delta^* \geq \frac{\Delta - 2}{2}$$

and so,

$$\Delta \leq 2\Delta^* + 2.$$

□

Theorem 3.2. *Algorithm 3.1 runs in polynomial time.*

Proof. Let $n = |V(G)|$. Since the sets of $\mathscr{S}$ are pairwise disjoint, there are at most n sets. Each set contains at most n vertices. Each vertex is adjacent to at most $n-1$ vertices. Therefore for any spanning tree T of G,

$$\sum_{S \in \mathscr{S}} deg_T(S) \leq n \cdot n \cdot (n-1) < n^3.$$

The number of sets S where $deg_T(S) = \Delta$ is $O(\frac{n^3}{\Delta})$. Algorithm 3.1 reduces $deg_T(S)$ for a set S by one through each *phase* (steps 3-10). Each phase identifies a good edge connecting two components of F^i, finds the cycle C_{uv}, the bad edge wz, and the bad set B. One phase can be done in polynomial time. The maximum degree of T is decreased by one in $O(\frac{n^3}{\Delta})$ phases. The number of phases is

$$O\left(\sum_{k=2}^{n} \frac{n^3}{k}\right) = O(n^3 \log n^3) = O(n^3 \log n).$$

□

In summary,

Theorem 1. *There exists a $2\Delta^* + 2$-approximation algorithm that runs in polynomial time for approximating MCSTs over a family of pairwise-disjoint subsets of vertices.*

Chapter 4

MCSTs on Laminar Sets

4.1 Overview

Definition 4.1. Let $\mathscr{S} = \{S_1, \ldots, S_k\}$, $S_i \subseteq V, 1 \leq i \leq k$, be a family of subsets of vertices of a graph $G = (V,E)$. $\mathscr{S}$ is a laminar family if $S_i \cap S_j \neq \emptyset$ implies that $S_i \subseteq S_j$ or $S_j \subseteq S_i$, $1 \leq i, j \leq k$.

In this chapter, the focus will be on a special case of MCSTs where the sets in $\mathscr{S}$ form a laminar family. Approximating the MCST of a graph will be done by approximating a related type of spanning tree.

Definition 4.2. Given a graph G, a family of subsets of vertices $\mathscr{S}$, and a bound b on the degree of any set of $\mathscr{S}$, a *bounded crossing spanning tree* (BCST) is a spanning tree T of G with the maximum degree of a set in $\mathscr{S}$ being at most b. As with BDMSTs, the degree bounds may be non-uniform, where the degree bound b for every set is replaced by a degree bound function $b : \mathscr{S} \to \mathbb{Z}^+$ that assigns a degree bound for each set of $\mathscr{S}$.

The algorithm to be presented will find a (non-uniform) BCST for a laminar family of sets. Given a bound b_i on the number of edges crossing each set $S_i \in \mathscr{S}$, the algorithm will find a spanning tree T where each set S_i is crossed $O(b_i \cdot \log n)$ times, or indicate that there is no spanning tree which crosses each set S_i at most b_i times. This algorithm can be used to approximate an MCST. Let T^* be an MCST of G and $\Delta^* = \Delta(T^*)$. Since $1 \leq \Delta^* \leq n-1$, one can perform a binary search on the possible values of Δ^*. Let b denote the possible value of Δ^* given by the binary search. The algorithm can attempt to find a spanning tree where the degree bound b_i for each set $S_i \in \mathscr{S}$ is set to b. If no such tree exists, the algorithm will fail to find a spanning tree and increase b for binary search. If the algorithm outputs a spanning tree with every set of $\mathscr{S}$ having degree $O(b \cdot \log n)$, the algorithm will set b lower. Eventually, the binary search will identify the lowest possible value Δ^* might be and the algorithm will give a spanning tree with every set of $\mathscr{S}$ having degree $O(\Delta^* \cdot \log n)$.

The algorithm is constructive. It will start will a set of potential edges $F = \emptyset$ for the final spanning tree T. In each iteration, a set of edges will be chosen to add to F. The edges will be chosen such that each set $S_i \in \mathscr{S}$ is crossed at most b_i times. If after α iterations F spans the graph G, then a spanning tree of these potential edges will be a spanning tree of G where each set S_i is crossed at most $O(\alpha \cdot b_i)$ times.

In order to bound the value of α, the algorithm will borrow ideas from Algorithm 2.2. The algorithm will start with a set of potential edges $F = \emptyset$. Thus (V,F) is a graph containing n components. In each iteration, the edges will be chosen such that when they are added to F, the number of components in (V,F) will reduce by a constant factor. After $O(\log n)$ iterations, the number of components will be reduced to one, F will be a

spanning subgraph of G, and $\alpha = O(\log n)$. Algorithm 2.2 uses T-joins to reduce the number of components. However, finding T-joins that cross any set of $\mathscr{S}$ a bounded number of times is a hard problem, which will be proven in the next chapter. Instead another approach is taken using the following combinatorial structures.

Definition 4.3. Given a laminar family of subsets of vertices $\mathscr{S} = \{S_1, \dots, S_p\}$ and a bound function $b : \mathscr{S} \to \mathbb{Z}^+$, a *maximum cross-free-cut b-matching* is a set of edges (possibly choosen multiple times) M of maximum cardinality such that each set $S_i \in \mathscr{S}$ is crossed at most b_i times.

Note that if the cross-free-cut b-matching spans the graph and is connected, then a spanning tree of the edges in M is a BCST for the bound function b.

The concept of *multicommodity flow* is also important. For the following definitions, let Q be a set of elements called *commodities*. Consider a graph $G = (V,E)$. For each element $q \in Q$, let there be a vertex pair (s_q, t_q), $s_q, t_q \in V$. Let $c : E \to \mathbb{Z}^+$ be a capacity function on the edges.

Problem 4.1. The *maximum multicommodity flow* problem is to route flow of each commodity $q \in Q$ between (s_q, t_q) such that the flow along each edge $e \in E$ is no more than $c(e)$ and the total flow of all the commodities of Q is maximized.

Definition 4.4. Given a maximum multicommodity flow problem on a graph G, a *multicut* is a set of edges M such that for any commodity $q \in Q$, there is no s_q, t_q-path in $G \setminus M$.

The problem of maximum multicommodity flow can be restricted as follows.

Problem 4.2. Given a specific (s_q, t_q)-path for each $q \in Q$, the *forced multicommodity flow* problem is to route flow of each commodity q along the specific (s_q, t_q)-path such that the flow along each edge $e \in E$ is no more than $c(e)$ and the total flow of all the commodities of Q is maximized.

Note that the forced multicommodity flow problem is essentially a path-packing problem. This problem is essential for the algorithm presented in this chapter.

During each iteration, the algorithm will run a subroutine to choose the edges to add to F. The subroutine will borrow ideas from [12] which will be outlined later in this chapter. The algorithm in [12] is for the maximum multicommodity flow problem on trees and gives a 1/2-approximation. The paper also contains a reduction from the problem of finding a maximum cross-free-cut b-matching to that of finding a maximum integral multicommodity flow on trees. The subroutine will perform a similar reduction of choosing edges to add to F to a forced integral multicommodity flow problem. This reduction will be done in multiple steps. The subroutine will then find a constant factor approximation of the corresponding forced integral multicommodity flow problem by extending the ideas of [12]. Using the solution to the corresponding forced integral multicommidty flow problem, a set of edges will be chosen to add to F. The constant factor of the approximation will give a constant-factor reduction of the components of (V,F).

4.2 Background

4.2.1 Representing $\mathscr{S}$

$\mathscr{S}$ is by our assumption a laminar family of subsets of vertices of the graph $G = (V,E)$. Given any two sets of $\mathscr{S}$, either one is contained in the other or they are disjoint. This gives a natural hierarchy to the sets of $\mathscr{S}$ where a set is considered "higher up" than any set contained in it. Often, hierarchal structures can be represented by rooted trees.

As in [12], we can represent $\mathscr{S}$ as a rooted tree $A = (V', E')$, called a *laminar tree*. The laminar tree A is constructed as follows:

- For each set $S_i \in \mathscr{S}$, add a vertex i to V'. Add a root vertex r to V'.
- For each inclusion-wise maximal set $S_i \in \mathscr{S}$, add the edge (i,r) to E'.
- For every other set $S_i \in \mathscr{S}$, if S_j is the inclusion-wise minimal set containing S_i, add the edge (i,j) to E' (later refered to as e_{S_i}.
- Construct the function $g : V \to V'$ as follows. For each vertex $v \in V$, if S_i is the inclusion-wise minimal set containing v, let $g(v) = i$. If v is not contained in a set, then let $g(v) = r$.

Figure 4.1 gives an example of a laminar tree.

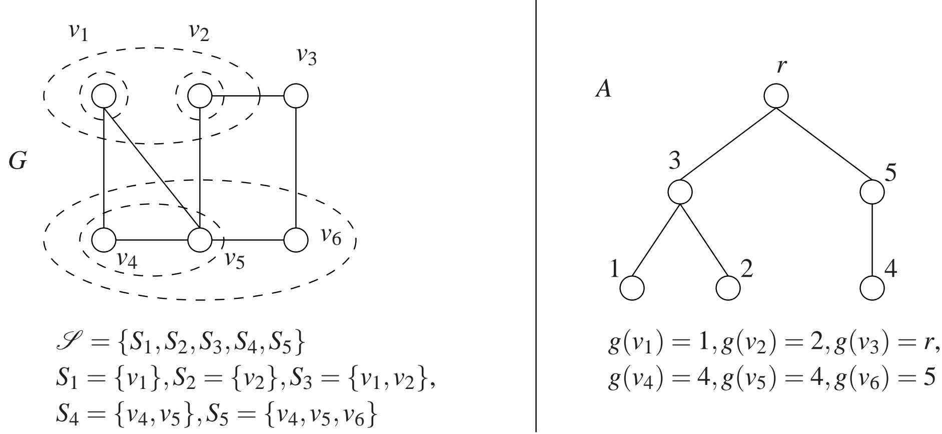


Figure 4.1: Graph G with family $\mathscr{S}$ and the laminar tree A

Note that not only does the laminar tree maintain the hierarchal structure of the sets of $\mathscr{S}$, but the function g encodes in which sets each vertex is contained. If $g(v) = i, v \in V, i \in V'$, then for each vertex j on the unique ir-path in A, $v \in S_j \in \mathscr{S}$ in G. Because of the function g, intuitively one could view each vertex i of A not just as the set $S_i \in \mathscr{S}$ but as the "inside" of S_i and the subtree of A rooted at i represents precisely everything contained in S_i.

4.2.2 Multicommodity flows and trees

In [12], Garg et al. reduce finding a maximum cross-free-cut b-matching on a graph $G = (V,E)$ to finding a maximum integral multicommodity flow on the corresponding laminar tree $A = (V', E')$. The reduction is as follows.

Consider an edge $(u,v) \in E$. The edge $e = (u,v) \in E$ will be represented by a commodity e with the corresponding vertex pair $\{s_e, t_e\} = \{g(u), g(v)\}$ in A. Since A is a tree, the $s_e t_e$-path is unique.

Currently, each vertex of A except r represents a set in $\mathscr{S}$. There is also a function g mapping each vertex v of G to the vertex of A representing the set containing v. If each vertex i of A is viewed as representing the "inside" of the set $S_i \in \mathscr{S}$, it would be helpful if something represented the "border" of S_i where edges cross S_i (i.e. $\delta(S_i)$).

In A, there is a unique ir-path. For any vertex α where the unique αr-path in A contains the vertex i, S_α is contained in S_i in G. For any vertex β where the unique ir-path in A contains the vertex β, S_i is contained

in S_β in G. For any other vertex z in A, S_i and S_z are disjoint in G. Intuitively, if the vertex i represents the "inside" of the set S_i, the first edge $e_{S_i} = (i, j)$ on the path from i to r represents the "border" of S_i.

To find an integral multicommodity flow on A, flow is pushed for each commodity between the corresponding vertex pair. As each commodity represents an edge of G, each unit of flow pushed on A for a given commodity will represent the corresponding edge in G being chosen once. Whenever an edge $e = (u, v)$ crosses a set S_i in G, the corresponding $g(u)g(v)$-path P_e uses the "border" edge e_{S_i}. Given a set S_i, let b_i be the bound on the number of edges to cross S_i in the cross-free-cut b-matching. Thus, for edge e_{S_i} in A, a capacity will be set where $c(e_{S_i}) = b_i$. This will ensure that for any given integral multicommodity flow satisfying the capacity function c, the corresponding cross-free-cut b-matching will not violate the bound b_i for set S_i. See Reduction 4.1 for a formal outline of reducing an instance of finding a maximum cross-free-cut b-matching to finding a maximum integral multicommodity flow.

Reduction 4.1 Reducing finding a maximum cross-free-cut b-matching to finding a maximum integral multicommodity flow

Given graph $G = (V, E)$; laminar family $\mathscr{S} = \{S_1, \ldots, S_k\}$, $S_i \subseteq V$, $1 \le i \le k$; function $b : \mathscr{S} \to \mathbb{Z}^+$;
Construct laminar tree $A = (V', E')$ and function g.
Construct capacity function $c : E' \to \mathbb{Z}^+$ where $c(e_{S_i}) = b(S_i)$.
for Each $e = uv \in E$ **do**
  Let e be a commodity where $(s_e, t_e) = (g(u), g(v))$.
  Let P_e be the unique $s_e t_e$-path in A.
end for

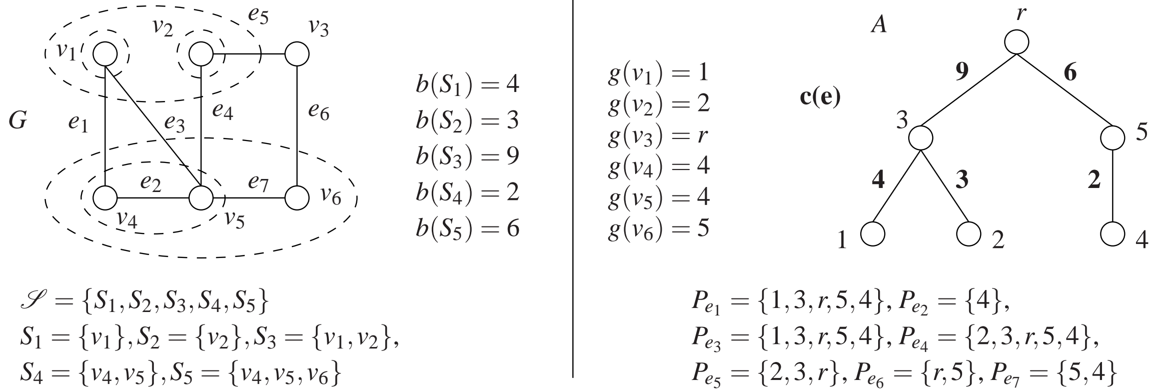


Figure 4.2: Example of graph G with family $\mathscr{S}$ and the corresponding multicommodity flow problem

Note that the reduction keeps track of the unique $s_e t_e$-path in A for each e. Since A is a tree, each $s_e t_e$-path is unique and keeping track of these paths is not necessary. However, the laminar tree will be extended to a larger graph that is not a tree. It is there that the paths must be known and thus a forced integral multicommodity flow must be used instead of a regular integral multicommodity flow. This will be explained in further detail later in this chapter.

By finding a maximum integral multicommodity flow on A, one has a maximum cross-free-cut b-matching on G. As stated earlier, if that b-matching was connected and spanning G, one could take the b-matching and

find a bounded crossing spanning tree of G. Although finding an integral multicommodity flow on trees is NP-hard, Garg et al. [12] present a 1/2-approximation primal-dual algorithm for integral multicommodity flows on trees.

Consider a multicommodity flow problem on a tree $A = (V', E')$ with edge capacities $c : E' \to \mathbb{Z}^+$ and q commodities with vertex pairs (s_i, t_i), $1 \leq i \leq q$. Let P_i be the unique $s_i t_i$-path in A.

The linear program for the multicommodity flow problem on A is:

$$\begin{array}{lrcll} \text{maximize} & \sum_{i=1}^{q} f_i & & & \\ \text{subject to} & \sum_{i: e \in P_i} f_i & \leq & c_e, & e \in E', \\ & f_i & \geq & 0, & 1 \leq i \leq q. \end{array}$$

The corresponding dual (the linear programming relaxation for finding a minimum multicut) is:

$$\begin{array}{lrcll} \text{minimize} & \sum_{e \in E'} d_e c_e & & & \\ \text{subject to} & \sum_{e \in P_i} d_e & \geq & 1, & 1 \leq i \leq q, \\ & d_e & \geq & 0, & e \in E'. \end{array}$$

The corresponding complementary slackness conditions are:

$$\begin{array}{ll} 1. & f_i > 0 \quad \Rightarrow \quad \sum_{e \in P_i} d_e = 1. \\ 2. & d_e > 0 \quad \Rightarrow \quad \sum_{i: e \in P_i} f_i = c_e. \end{array}$$

Rather than enforce these complementary slackness conditions, the first one will be relaxed to:

$$f_i > 0 \Rightarrow \sum_{e \in P_i} d_e \leq 2.$$

This relaxed first condition implies that for each commodity where there is flow, the path used to push flow for that commodity will contain at most two edges in the multicut. Note that in order to find a b-matching, an edge for the b-matching is chosen if in the multicommodity flow problem the corresponding flow has a value of at least one. The flow that the algorithm finds should be one such that if $f_i > 0$, then $f_i \geq 1$. Maintaining an integral flow is sufficient for satisfying this requirement.

Lemma 4.1. *[12] If a multicommodity flow F and multicut M satisfy the relaxed complementary slackness conditions, then the amount of flow in F is at least half the capacity of M.*

Proof. Given than F and M satisfy the relaxed complementary slackness conditions, let f_i represent the flow of F for commodity i and $d_e = 1$ if e is in M, $d_e = 0$ if e is not in M.

$$\begin{array}{rcll} \sum_{i=1}^{q} f_i & \geq & \frac{1}{2} \sum_{i=1}^{q} \sum_{e \in P_i} d_e f_i & \text{by the relaxed first complementary slackness condition} \\ & \geq & \frac{1}{2} \sum_{e \in E'} d_e \cdot \sum_{e \in P_i} f_i & \\ & = & \frac{1}{2} \sum_{e \in E'} d_e c_e & \text{by the second complementary slackness condition} \end{array}$$

□

Corollary 4.1. *[12] Given a multicommodity flow F and a multicut M that satisfies the relaxed complementary slackness conditions, the flow F is at least half the value of the maximum integral multicommodity flow (it fact, it is at least half the value of any multicommodity flow).*

Proof.

$$\begin{aligned} \text{amount of flow of } F &\geq \frac{1}{2}\text{ capacity of } M \\ &\geq \frac{1}{2}\text{ capacity of minimum multicut} \\ &\geq \frac{1}{2}\text{ maximum multicommodity flow} \\ &\geq \frac{1}{2}\text{ maximum integral multicommodity flow.} \end{aligned}$$

□

The algorithm from [12] uses Corollary 4.1 by doing the following. Given the tree $A = (V', E')$, the algorithm picks an arbitrary vertex $r \in V'$ and roots A at r. If A is a laminar tree, then A is already rooted. For any other vertex $v \in V'$, let $level(v)$ denote the length of the unique vr-path in A. Given vertices $u, v \in V'$, let the lowest common ancestor, $lca(u, v)$, be the vertex on the unique uv-path with the lowest level. Note that r is the lowest level vertex in A.

The algorithm iterates through each vertex of A from highest level to lowest level. For each vertex $v \in V'$, the algorithm will greedily attempt to push flow for any commodity i where $lca(s_i, t_i) = v$ without violating any of the edge capacities. Note that by forcing the edge capacities to be integral, greedily pushing flow will result in the flow being integral. Whenever the flow on the edge is equal to its capacity, the edge is considered to be *saturated*. As each edge of A is saturated, it is added to an ordered list D. If multiple edges are saturated at the same time, they are added to D in arbitrary order. After the algorithm has iterated through every vertex of A, D will be a multicut. This is because the algorithm will it iterate through every commodity and will push as much flow as possible. If D was not a multicut, then some commodity can push more flow, contracting the fact that flow was pushed greedily. The algorithm will go through the edges in D in reverse order of their addition and remove any edge that is not needed in order for D to still be a multicut. This step is called *reverse delete*. It turns out that *reverse delete* is sufficient for making sure that the final flow and the multicut D satisfy the relaxed complementary slackness conditions. Algorithm 4.2 outlines the algorithm formally.

Given a $s_i t_i$-path P_i, $lca(s_i, t_i)$ splits P_i into two paths P_i^1 and P_i^2.

Lemma 4.2. *[12] After reverse delete in Algorithm 4.2, for any commodity i with positive flow and $j = 1, 2$, $|P_i^j \cap D| \leq 1$.*

Proof. Without loss of generality, consider path P_i^1. Assume that $|P_i^1 \cap D| \geq 2$. Let $e, e' \in P_i^1 \cap D$. Let e be the edge further away from the root r. Let $v = lca(s_i, t_i)$.

If e' was saturated before e was, then e was saturated while flow was being pushed for a commodity z. Let $u = lca(s_z, t_z)$. Since e' is saturated, then $level(u) < level(v)$. But then flow should have been pushed for commodity z before commodity i, a contradiction.

If e' was saturated the same time e was, for any commodity z with positive flow on e, either e' also has flow for z or flow was pushed for z before e was saturated and another edge would have been added to D. e

Algorithm 4.2 Approximating the maximum integral multicommodity flow in trees

Given tree $A = (V', E')$, edge capacities $c : E' \to \mathbb{Z}^+$, vertex pairs (s_i, t_i), $1 \leq i \leq q$.
Set flow to 0, multicut $D = \emptyset$.
for $v \in V$ in nonincreasing order of level **do**
  for Each (s_i, t_i) where $lca(s_i, t_i) = v$ **do**
    Greedily push flow from s_i to t_i while satisfying capacity constraints.
  end for
  Add every edge saturated to D in arbitrary order.
end for
Let $e_1, \ldots, e_l$ be the ordered list of edges in D.
for $j = l$ to 1 **do**
  If $D - \{e_j\}$ is a multicut of A, then $D = D \setminus \{e_j\}$.
end for

is not needed in D for the first case because of e'. e is not needed in D for the second case because the other edge in D would be looked at after e by reverse delete, a contradiction.

Thus, e' was saturated after e was and was checked first by reverse delete. Since e is still in D after reverse delete, there is a commodity z not using e'. Thus, $u = lca(s_z, t_z)$ is somewhere between e and e'.

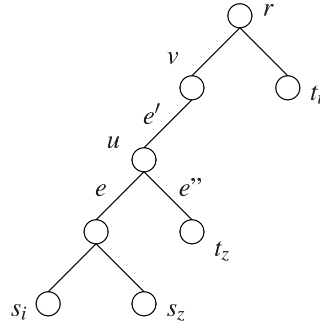


Figure 4.3: Example of checking edges in reverse delete

Since commodity i has positive flow, when flow was pushed for commodity z, another edge e'' was saturated. Algorithm 4.2 looked at u before v because u has a higher level than v. Thus, e'' is still in D when reverse delete checks e and should have removed e, a contradiction. □

Corollary 4.2. *After reverse delete in Algorithm 4.2, for any commodity i with positive flow* $|P_i \cap D| \leq 2$.

Thus, the integral multicommodity flow and multicut given by Algorithm 4.2 satisfies the relaxed complementary slackness conditions and therefore by Corollary 4.1 gives a 1/2-approximation for the maximum integral multicommodity flow. However, when applying this algorithm to the problem given by Reduction 4.1, there is no guarantee that the resulting integral multicommodity flow will correspond to a cross-free-cut b-matching that is connected. Algorithm 4.2 will be altered in the next section to force some form of connectivity guarantee in the corresponding cross-free-cut b-matching.

4.3 The Algorithm

4.3.1 Extending multicommodity flows on trees

Consider a maximum cross-free-cut b-matching problem with an additional constraint that the b-matching is connected. Applying Reduction 4.1 will produce a maximum integral multicommodity flow problem where the connectivity constraint is lost. The problem is that there is no simple way of identifying the components of the b-matching from the laminar tree A.

The algorithm will start with a set of potential edges $F = \emptyset$ and try to reduce the number of components of (V,F) by a constant factor in each iteration by adding edges to F. Assume that F is a set of potential edges chosen for a cross-free-cut b-matching. In order to track the number of components in the graph (V,F), the laminar tree A will be extended to a new graph, denoted by B_F. As edges are added to F after each iteration, the graph B_F will change as well.

Let $\mathscr{C} = \{C_1,\dots,C_p\}$ be the components of (V,F). In each iteration, the number of components can be reduced by at least a factor of 1/2 if each component of $\mathscr{C}$ is connected to another component by a new edge. Only edges connecting different components of (V,F) in G will be considered. The integral multicommodity flow problem to find these edges will be constructed as follows.

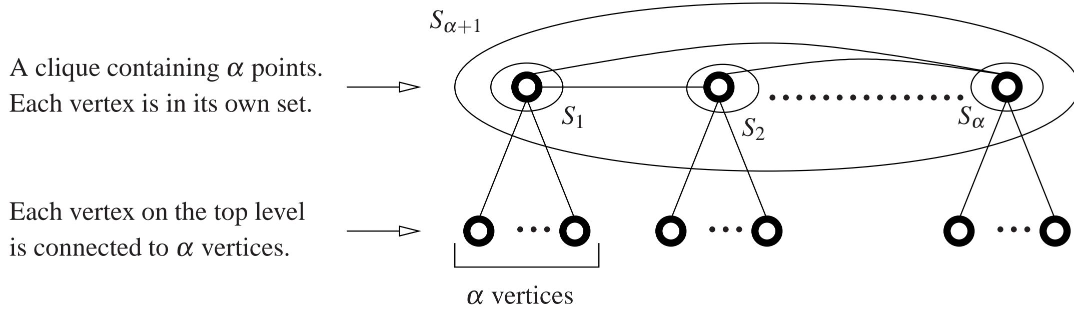
Initialize the construction of the graph $B_F = (\bar{V},\bar{E})$ and the forced integral multicommodity flow problem with the laminar tree A and the maximum integral multicommodity flow problem given by Reduction 4.1. Note that in each iteration, the integral multicommodity flow problem will only use commodities corresponding to edges of G that connect components of (V,F). For each vertex v in the original graph G, add the vertex v and the edge $(v,g(v))$ to B_F. This is equivalent to adding every vertex subset of size one to $\mathscr{S}$. Since these "sets" are not part of $\mathscr{S}$, the capacity of each of these new edges will be set to ∞.

For each component $C_i \in \mathscr{C}$, add the vertices $c_i, c'_i, \bar{c}_i$ to B_F and the edge (c_i,c'_i). Intuitively, the new flow problem will have flow between components. c_i will act as a starting vertex for flow leaving C_i. $\bar{c}_i$ will act as an ending vertex for flow entering C_i. In the b-matching problem, there is the possibility of choosing too many edges between a small number of components in (V,F) while other components have no adjacent edges added to F. An example is in Figure 4.4. This will be prevented by fixing a capacity of 1 to the edge (c_i,c'_i), representing at most one unit of flow leaving component C_i. For each vertex $v \in C_i$, add edges (v,c'_i) and $(v,\bar{c}_i)$ with capacity ∞. Now B_F contains the component structure of (V,F) in relation to $\mathscr{S}$.

A forced integral multicommodity flow problem is being used since the graph B_F is not a tree and thus there may exist multiple $s_e t_e$-paths for any given $e \in E$. The unique path used by each commodity e for the regular integral multicommodity flow problem on A encoded which sets of $\mathscr{S}$ are being crossed by the edge e in G. Similarly, each commodity e in the forced integral multicommodity flow problem must use a path in B_F that still encodes which sets of $\mathscr{S}$ are being crossed by the edge e in G. For each commodity e given by Reduction 4.1, let $P_e = \{v_0,\dots,v_q\}$ be the unique $s_e t_e$-path in A, where $e = (u,v)$ in G, $g(u) = v_0$, $g(v) = v_q$. In B_F, there will be two commodities e_1 and e_2. Let $u \in C_u \in \mathscr{C}$ and $v \in C_v \in \mathscr{C}$. The paths for each commodity will be $P_{e_1} = \{c_u, c'_u, u, v_0, \dots, v_q, v, \bar{c}_v\}$ and $P_{e_2} = \{c_v, c'_v, v, v_q, \dots, v_0, u, \bar{c}_u\}$. Note that since each edge of the form (c_i,c'_i) has capacity one, two commodities instead of one are required for each edge of G. If only one commodity was used per edge of G, each edge of G would have to be assigned to a specific component of (V,F) but each component may use at most one edge of G to connect to another component. This could potentially decrease the value of the optimal solution.

The way to construct B_F from A and the corresponding forced integral multicommodity flow problem is given formally in Reduction 4.3. Figure 4.5 gives an example of the forced integral multicommodity flow problem.

Note that there was a rooted vertex r in A. r is still a vertex in B_F. Just as r played a crucial role



Given this graph G, start with no edges chosen for the b-matching. There are $\alpha(\alpha+1)$ components.
Set the bound $b_i = \alpha + 2$ for every set S_i, $1 \le i \le \alpha$, $b_{\alpha+1} = \alpha^2$.
A spanning tree satisfying the bounds exists by finding a path of length α on the clique,
and choosing all the edges crossing $S_{\alpha+1}$.
If all the edges in the clique inside $S_{\alpha+1}$ are chosen for the b-matching,
for each S_i, $1 \le i \le \alpha$, three of the edges that cross S_i and $S_{\alpha+1}$ can be added to the b-matching.
The result is $\alpha(\alpha-3)+1$ components.

Figure 4.4: How a b-matching can have too many components

Reduction 4.3 Constructing the forced integral multicommodity flow problem on the graph B_F

Given graph $G = (V,E)$; laminar family $\mathscr{S} = \{S_1,\dots,S_k\}$, $S_i \subseteq V$, $1 \le i \le k$; function $b : \mathscr{S} \to \mathbb{Z}^+$; edges $F \subseteq E$
Let $\mathscr{C} = \{C_1,\dots,C_p\}$ be the connected components of (V,F).
Let $\bar{F} \subseteq E$ be the edges of G that have endpoints in different components of $\mathscr{C}$.
Apply Reduction 4.1 to $(V,\bar{F})$ to get $A = (V',E')$.
Let $B_F = (\bar{V},\bar{E})$ where $\bar{V} = V' \cup V$, $\bar{E} = E'$, edge capacities remain the same.
for Each $v \in V$ **do**
  Add edge $e = (v, g(v))$ to $\bar{E}$. Set $c(e) = \infty$.
end for
for Each component $C_i \in \mathscr{C}$ **do**
  Add vertices $c_i, c_i', \bar{c}_i$ to $\bar{V}$.
  Add edge $e = (c_i, c_i')$ to $\bar{E}$. Set $c(e) = 1$.
  for Each vertex $v \in C_i$ **do**
    Add edges $e = (v, c_i'), e' = (v, \bar{c}_i)$ to $\bar{E}$. Set $c(e), c(e') = \infty$.
  end for
end for
for Each $e = (u,v) \in \bar{F}$ **do**
  Consider $P_e = \{v_0,\dots,v_q\}$ given by Reduction 4.1 where $g(u) = v_0$, $g(v) = v_q$.
  Let e_1, e_2 be two commodities in the forced integral multicommodity flow problem.
  Let $u \in C_u$, $v \in C_v$.
  Let $P_{e_1} = \{c_u, c_u', u, v_0, \dots, v_q, v, \bar{c}_v\}$.
  Let $P_{e_2} = \{c_v, c_v', v, v_q, \dots, v_0, u, \bar{c}_u\}$.
end for

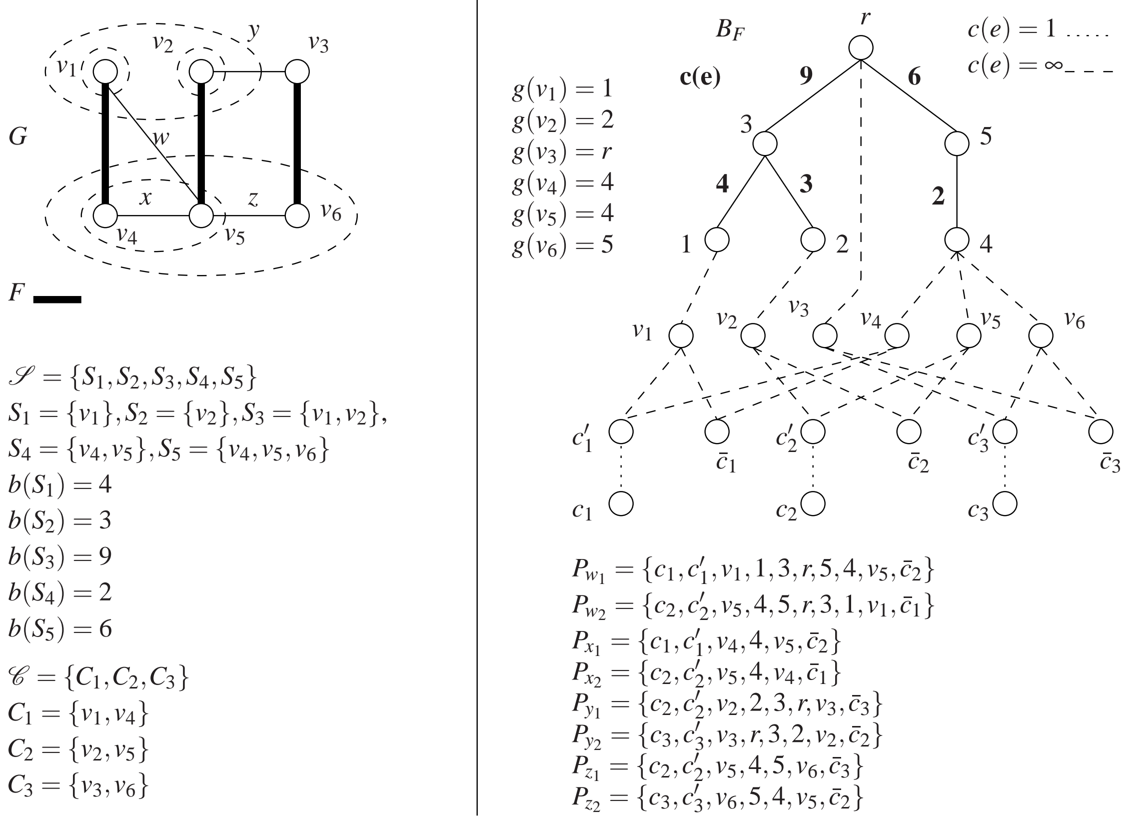


Figure 4.5: Graph G, laminar family $\mathscr{S}$, edges $F \subseteq E$, and the corresponding auxiliary graph B_F

in Algorithm 4.2, r will play a crucial role in the algorithm for the forced integral multicommodity flow problem. Here is a summary of the reductions. Instead of finding a BCST directly, the goal is to find a connected maximum cross-free-cut b-matching on G. Reduction 4.1 takes G and produces a maximum integral multicommodity flow problem on the laminar tree A. During each iteration of the final algorithm, there will be a set F of potential edges already chosen. Each iteration will use Reduction 4.3, which uses Reduction 4.1, to switch from finding the connected maximum cross-free-cut b-matching to a forced integral multicommodity flow problem on a graph B_F. The following subsections will show how edges are added to F.

4.3.2 Subroutine for forced multicommodity flows

In this section, we will outline a subroutine to connect the components of (V, F). The main idea will be to find edges of $G - F$ that are connecting components. Adding one of these edges to F would reduce the number of components of $\mathscr{C}$ by 1. In the subroutine, Reduction 4.3 will give an auxiliary forced integral multicommodity flow problem with an optimal solution corresponding to a set of edges of G where each component of $\mathscr{C}$ is being connected to some other component. The subroutine will then find a 1/3-approximation to the forced

integralmulticommodity flow problem in a way similar to how Algorithm 4.2 works.

Similar to Algorithm 4.2, the subroutine will be a primal-dual algorithm on the linear program for the forced multicommodity flow problem. The forced multicommodity flow problem requires that each commodity has a unique path it can flow on. Thus, we can use a linear program similar to the one for multicommodity flows on trees for the forced multicommodity flow problem on B_F.

The forced multicommodity problem is on $B_F = (\bar{V}, \bar{E})$ with edge capacities $c : \bar{E} \to \mathbb{Z}^+$ and q commodities with vertex pairs (s_i, t_i), $1 \le i \le q$. Let P_i be the assigned $s_i t_i$-path in B_F. Note that q will decrease as edges are added to F in each iteration.

The linear program for the forced multicommodity flow problem is:

$$\begin{aligned} \text{maximize} \sum_{i=1}^{q} f_i & & \\ \text{subject to} \sum_{i:e \in P_i} f_i &\le c_e, & e \in \bar{E}, \\ f_i &\ge 0, & 1 \le i \le q. \end{aligned}$$

The corresponding dual is:

$$\begin{aligned} \text{minimize} \sum_{e \in \bar{E}} d_e c_e & & \\ \text{subject to} \sum_{e \in P_i} d_e &\ge 1, & 1 \le i \le q, \\ d_e \ge 0, \quad e \in \bar{E}. & & \end{aligned}$$

The corresponding complementary slackness conditions are:

$$\begin{aligned} 1. \quad f_i > 0 \quad &\Rightarrow \quad \sum_{e \in P_i} d_e = 1. \\ 2. \quad d_e > 0 \quad &\Rightarrow \quad \sum_{i:e \in P_i} f_i = c_e. \end{aligned}$$

For the complementary slackness conditions, the subroutine will require a more relaxed first condition:

$$f_i > 0 \Rightarrow \sum_{e \in P_i} d_e \le 3.$$

A proof analogous to Lemma 4.1 and Corollary 4.1 shows that if the final algorithm outputs a flow F and a set of edges M that satisfies the relaxed complementary slackness conditions, then the flow F is at least 1/3 the value of the maximum integral forced multicommodity flow. This proof is postponed to the next subsection. Note that integral solutions to the dual of the forced multicommodity flow problem do not correspond to mutlicuts. This is because they do not block every $s_e t_e$-path in B_F. However, they do block each given $s_e t_e$-path P_e.

For each vertex $v \in \bar{V}$, let the level of v, $level(v)$, be the distance from v to the vertex $r \in \bar{V}$. Let $\bar{F}$ denote the edges of G with endpoints in different components of (V, F). Let $\mathscr{P} = \{P_e : e \in \bar{F}\}$ be the paths given by Reduction 4.3. For each path $P_e \in \mathscr{P}$, let the apex of P_e, $apex(P_e)$, be the vertex of lowest level in P_e. The subroutine will greedily push flow on the path of $\mathscr{P}$ with the apex with the highest level. Just like Algorithm 4.2, the edge capacities are integral. Thus, greedily pushing flow implies that the flow is integral. As each edge of B_F becomes saturated, it will be added to an ordered list D. Note that D is a feasible dual solution

to the forced multicommodity flow problem. The algorithm attempts to push flow for every commodity and either adds an edge on the corresponding path to D after pushing some flow or an edge of the path is already in D because no more flow can be pushed on that edge. If D is not feasible for the dual, then some commodity can push more flow, contradicting the fact that flow was pushed greedily. If multiple edges of B_F become satruated at the same time, they are added to D in arbitrary order. As with Algorithm 4.2, a reverse delete step will be performed on D such that the final flow and D will satisfy the relaxed complementary slackness conditions.

The subroutine for connecting components of (V,F) is given in Algorithm 4.4.

Algorithm 4.4 Subroutine for adding edges to F

Given $G=(V,E)$; laminar family $\mathscr{S}=\{S_1,\dots,S_k\}$, $S_i \subseteq V$, $1 \leq i \leq k$; function $b:\mathscr{S}\rightarrow\mathbb{Z}^+$; set of edges $F \subseteq E$.
Apply Reduction 4.3 to get the forced multicommodity flow problem on $B_F=(\bar{V},\bar{E})$.
Let $D=\emptyset$.
for $v \in \bar{V}$ in nonincreasing order of level **do**
  for $P_e \in \mathscr{P}$ where $apex(P_e)=v$ **do**
    Push flow along P_e while satisfying capacity constraints.
  end for
  Add every edge saturated to D in arbitrary order.
end for
Let $e_1,\dots,e_\alpha$ be the ordered list of edges in D.
for $j=\alpha$ down to 1 **do**
  if $D-\{e_j\}$ is a feasible dual solution of the forced multicommodity flow problem on G **then**
    $D=D\setminus\{e_j\}$.
  end if
end for

4.3.3 Analysis of the subroutine

Consider a tree T. One can easily orient the edges of T such that every vertex has out-degree of at most one.

Claim 4.1. *Assume that G has a BCST T and a set of edges F have already been chosen. The forced integral multicommodity flow problem on B_F from Reduction 4.3 has a feasible flow of value $|\mathscr{C}|-1$.*

Proof. Consider the graph $G/\mathscr{C}$ obtained from G by contracting each component $C_i \in \mathscr{C}$ to a single vertex representing C_i. The edges of T in $G/\mathscr{C}$ form a spanning subgraph of $G/\mathscr{C}$. Let T' be a spanning tree of that subgraph. One can orient the edges of T' so that every vertex has out-degree at most one. Consider Reduction 4.3 on (V,F). Each arc e of T' can be represented by a path on the auxiliary graph B_F by either path P_{e_1} or P_{e_2}. If the tail of e is component C_t and the head of e is component C_h, choose in B_F the path where the first edge is (c_t,c_t') and the last vertex is $\bar{c}_h$. Choosing the proper paths in B_F to represent the arcs of T' is a feasible forced integral multicommodity flow on B_F. Thus, if $|\mathscr{C}|=p$ then there is a feasible forced integral multicommodity flow of $p-1$ on B_F. □

Note that each component C_i of $\mathscr{C}$ is represented by the edge (c_i,c_i') with a capacity of one in B_F. This means that if any feasible forced integral multicommodity flow in B_F contains j paths, then the corresponding edges of G have endpoints in at least j different components of (V,F). However, the paths may correspond

to cycles in $G/\mathscr{C}$. In the worst case, the cycles could be of length 2 (i.e. the feasible flow contains one unit of flow for a commodity representing an edge of G from component C_α to component C_β and vice versa). Therefore, a feasible flow of value j corresponds to edges of $G/\mathscr{C}$ that reduce the number of components of (V,F) by at least $j/2$.

Lemma 4.3. *A maximum forced integral multicommodity flow on B_F corresponds to a set of edges of G that when added to (V,F) will reduce the number of components by a factor of 1/2.*

Proof. Combining the previous argument and Claim 4.1, finding a maximum forced integral multicommodity flow on B_F will ensure that adding the corresponding edges to F will reduce the number of components in $\mathscr{C}$ by a factor of 1/2. □

Lemma 4.4. *Let D be given by the subroutine, Algortihm 4.4. D is a feasible integral solution to the dual of the forced multicommodity flow problem.*

Proof. Consider D in the subroutine just before the reverse delete step. The subroutine iterates through each path P_{e_i}, $e \in E$, $1 \leq i \leq 2$, in nonincreasing order of the level of the apex of each path, and attempts to push flow along that path. If no flow was pushed on P_{e_i}, then there was a saturated edge z on the path P_{e_i}. z would have been added to D when it became saturated so there is an edge of P_{e_i} in D. If some flow was pushed on P_{e_i}, then flow was pushed until an edge z became saturated. Therefore z would have be added to D. Thus, each path P_{e_i}, $e \in E$, $1 \leq i \leq 2$ has an edge in D. Therefore D a feasible integral solution to the dual of the forced multicommodity flow problem before the reverse delete step.

During the reverse delete step, edges are removed from D only if the result would still be a feasible dual solution. Thus, the subroutine returns a feasible integral solution D to the dual of the forced multicommodity flow problem. □

Lemma 4.5. *Consider the end of the subroutine after the reverse delete step. Let P_{e_i}, $e \in E$, $1 \leq i \leq 2$ be a path with positive flow. Then $|P_{e_i} \cap D| \leq 3$.*

Proof. Only edges of finite capacity can be in D.

Note that the laminar tree $A = (V', E')$ is a subgraph of B_F (refer to Reduction 4.3). For each P_{e_i}, $e \in E$, $1 \leq i \leq 2$ where the flow is positive, consider $P_{e_i} \cap E'$ (the part of P_{e_i} on the laminar tree subgraph). See Figure 4.3.3 for an example.

Lemma 4.2 implies that $|P_{e_i} \cap D| \leq 2$.

The only other edge on P_{e_i} with finite capacity, besides the ones on E', is the edge (C_j, C'_j) with capacity one at the begining of P_{e_i}. Thus $|P_{e_i} \cap D| \leq 3$. □

Thus, the forced integral multicommodity flow and D output by the subroutine satisfy the relaxed complementary slackness conditions. In the worst case, D is an optimal solution to the dual of the forced multicommodity flow linear program. The result is the following corollary.

Corollary 4.3. *The flow output by Algorithm 4.4 is at least 1/3 the size of a maximum forced multicommodity flow in B_F.*

By taking the edges of G that correspond to commodities with positive flow, then Lemma 4.3 and Corollary 4.3 proves the following lemma about Algortihm 4.4.

Lemma 4.6. *Given a graph $G = (V,E)$ and a subgraph $(V,F), F \subseteq E$, of G, Algorithm 4.4 can find a set of edges $X \subseteq E \setminus F$ such that the edges of X cross each set $S_i \in \mathscr{S}$ at most b_i times and adding the edges X to (V,F) will reduce the number of components by a factor of at most $5/6$.*

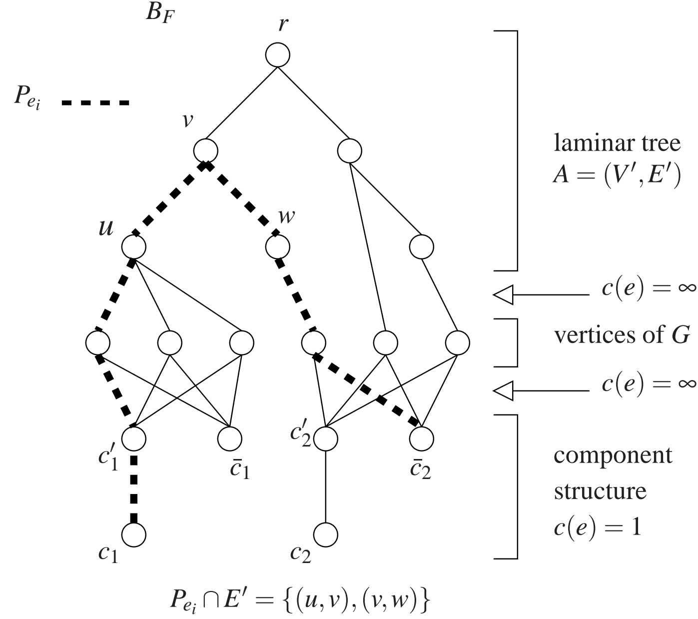


Figure 4.6: P_{e_i} on the graph B_F.

4.3.4 Final Algorithm

The final algorithm will use the subroutine to continuously add edges to F and reduce the number of components of (V,F) until F is a spanning subgraph of G. Given the 1/6-approximation of the subroutine, the algorithm should finish after $\left\lceil \frac{\log n}{\log(6/5)} \right\rceil$ times through the subroutine. If not, then the assumption that a BCST exists is false for the given bound function b. Algorithm 4.5 outlines the final algortihm.

4.3.5 Performance

Assuming a spanning tree exists which satifies the crossing restrictions, the subroutine finds at least 1/6 of the remaining edges needed for a connected spanning subgraph of G. After running the subroutine $\log_{6/5} n$ times, the resulting graph F will be connected and spanning G. Each time through the subroutine, no set $S_i \in \mathscr{S}$ is crossed more than b_i times so the spanning tree of F crosses each set S_i no more than $\frac{\log n}{\log(6/5)} \cdot b_i$ times.

Each conversion to the auxiliary graph can be done in polynomial time. During each time through the subroutine, each flow is pushed only once so the subroutine is done in $O(|E|) = O(n^2)$ time. Thus the algorithm runs in polynomial time.

In summary,

Theorem 2. *There is a deterministic $O(\log n)$-approximation algorithm that runs in polynomial time for approximating MCSTs and BCSTs where the family of subsets of vertices is a laminar family.*

Algorithm 4.5 Approximation algorithm to find BCST-L

Given: connected graph $G = (V, E)$; laminar family $\mathscr{S} = \{S_1, \ldots, S_k\}$, $S_i \subseteq V$, $1 \leq i \leq k$; $b : \mathscr{S} \to \mathbb{Z}^+$.
$F := \emptyset$, *count* $:= 0$.
while F is not a spanning subgraph of G **do**
  Use the subroutine, Algorithm 4.4.
  Add edges to F if the corresponding flow in the subroutine was positive.
  $count = count + 1$.
  if (V, F) is connected **then**
    Output spanning tree of (V, F).
  else if $count > \left\lceil \frac{\log n}{\log(6/5)} \right\rceil$ **then**
    No BCST exists.
  end if
end while

Chapter 5

Minimum Crossing Perfect Matchings

5.1 Overview

In analyzing the MCST problem for laminar sets, one possible approach to the problem is to look at a related T-join problem. As shown with Algorithm 2.2, the BDMST problem can be approximated by connecting components using structures related to T-joins. In a similar way, the MCST problem could be approximated. Finding T-joins is considered an easy problem. However, finding T-joins that minimizes the maximum number of times a set is crossed turns out to be a hard problem. A special case of the T-join problem is when T is the entire vertex set of a graph and the graph itself has a perfect matching.

Definition 5.1. Given a graph G and a family of subsets of vertices $\mathscr{S}$, a *minimum crossing perfect matching* (MCPM) is a perfect matching M of G such that $\Delta(M)$ is minimized.

This chapter will show that finding an MCPM is NP-hard.

5.2 Polyhedral viewpoint

One natural question is whether the linear programming relaxation for the MCPM integer program is integral. Here the focus will be on the linear program for bipartite graphs.

The integer program for the MCPM problem on a bipartite graph $G = (V, E)$ is:

$$\begin{aligned} \text{minimize } \Delta & & & \\ \text{subject to } \sum_{u \in V} x_{uv} &= 1 & & \forall v \in V \\ -\sum_{uv \in \delta(S)} x_{uv} + \Delta &\geq 0 & & \forall S \in \mathscr{S} \\ x_{uv} &\in \{0,1\} & & \forall uv \in E \\ \Delta &\geq 0. & & \end{aligned}$$

The constraint $x_{uv} \in \{0,1\}$ is relaxed to $x \geq 0$ in the linear programming relaxation. The dual of the linear

program is:

$$\begin{aligned}
\text{maximize } & \sum_{v \in V} y_v \\
\text{subject to } y_u + y_v - \sum_{S \in \mathscr{S} : uv \in \delta(S)} z_S & \leq 0 \quad \forall uv \in E \\
\sum_{S \in \mathscr{S}} z_S & \leq 1 \\
z_S & \geq 0 \quad \forall S \in \mathscr{S}.
\end{aligned}$$

The complementary slackness conditions are:

1. $x_{uv} > 0 \Rightarrow y_u + y_v = \sum_{S : uv \in \delta(S)} z_S$
2. $z_S > 0 \Rightarrow \sum_{uv \in \delta(S)} x_{uv} = \Delta$

Let M^* be an MCPM for the graph G. Let $\Delta^* = \Delta(M^*)$. Consider the simple example in Figure 5.1.

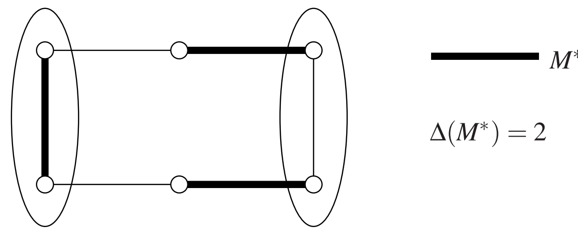


Figure 5.1: 6-cycle graph, 2 disjoint sets, integral solution to MCPM linear program

However, Figure 5.2 is a half-integral solution with a better value for Δ than Figure 5.1.

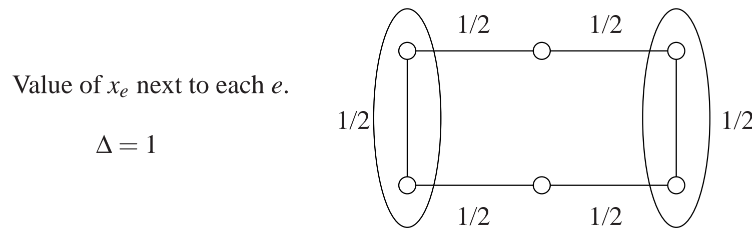


Figure 5.2: 6-cycle graph, 2 disjoint sets, half-integral solution to MCPM linear program

In fact, just letting G be an even length cycle and $\mathscr{S}$ consist of pairwise disjoint vertex sets can result in solutions to the linear program relaxation where even the objective value is not integral.

The solution given in Figure 5.3 satisfies the complementary slackness conditions and the objective value is non-integral.

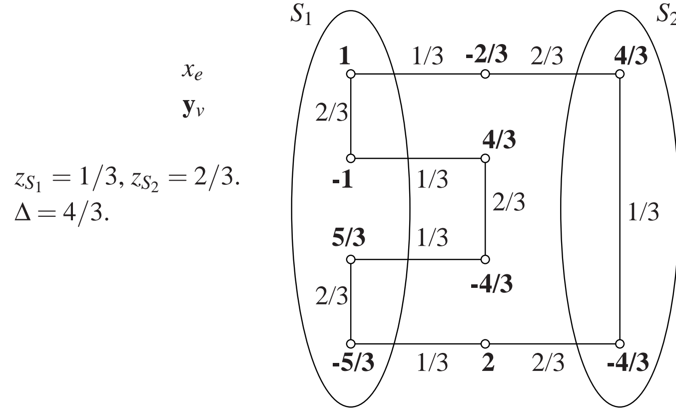


Figure 5.3: A non-integral objective value and dual solution for MCPM linear program

5.3 A hard matching problem

In this section, a reduction will be presented to show that finding an MCPM is NP-hard. In order to do so, it must be possible to take any instance of an NP-complete problem and reduce it in polynomial time to an instance of MCPM.

Definition 5.2. Given three pairwise dsjoint sets of elements W, X, and Y, where $|W| = |X| = |Y|$, and a set of triples $\mathscr{T} \subseteq W \times X \times Y$, a *three-dimensional matching* (3DM) is a subset of triples in $\mathscr{T}$ such that each element of $W \cup X \cup Y$ is contained in exactly one triple.

The problem of finding a 3DM of maximum cardinality is NP-complete [11]. The reduction will reduce an instance of the problem of finding a 3DM of maximum cardinality to an instance of the MCPM problem. The instance of the MCPM problem will be the special case where the graph is bipartite and the sets in $\mathscr{S}$ are pairwise disjoint.

Consider an instance of the 3DM problem with the three sets $W = \{w_1, \ldots, w_q\}$, $X = \{x_1, \ldots, x_q\}$, and $Y = \{y_1, \ldots, y_q\}$, and triples $\mathscr{T} \subseteq W \times X \times Y$. Without loss of generality, assume that each element of $W \cup X \cup Y$ is contained in at least one triple. Construct an auxiliary graph $G = (V, E)$ as follows:

- For each $x_j \in X$, construct a vertex $x_j \in V$.
- For each $y_k \in Y$, construct a vertex $y_k \in V$.
- For each triple $(w_i, x_j, y_k) \in \mathscr{T}$, construct the vertices $a_{ijk}, \bar{a}_{ijk} \in V$ and the edges $(a_{ijk}, \bar{a}_{ijk})$, (a_{ijk}, x_j), $(\bar{a}_{ijk}, y_k) \in E$.
- For each $w_i \in W$, construct the vertex set $S_{w_i} = \{a_{ijk}, \bar{a}_{ijk} : (w_i, x_j, y_k) \in \mathscr{T}\}$.

See Figure 5.4 for more detail.

Let $\mathscr{S} = \{S_w : w \in W\}$. Note that $x_j, y_k \notin S_w$, for all $x_j \in X$, $y_k \in Y$, $S_w \in \mathscr{S}$. a_{ijk} and $\bar{a}_{ijk} \in S_{w_\alpha}$ if and only if $i = \alpha$. Thus $\mathscr{S}$ is a family of pairwise disjoint subsets of V.

Consider the partition (A, B) of V where $A = \{x_j : \forall j\} \cup \{\bar{a}_{ijk} : \forall i, j, k\}$ and $B = \{y_k : \forall k\} \cup \{a_{ijk} : \forall i, j, k\}$. There is no edge in G with both endpoints in A or both endpoints in B. Thus G is a bipartite graph.

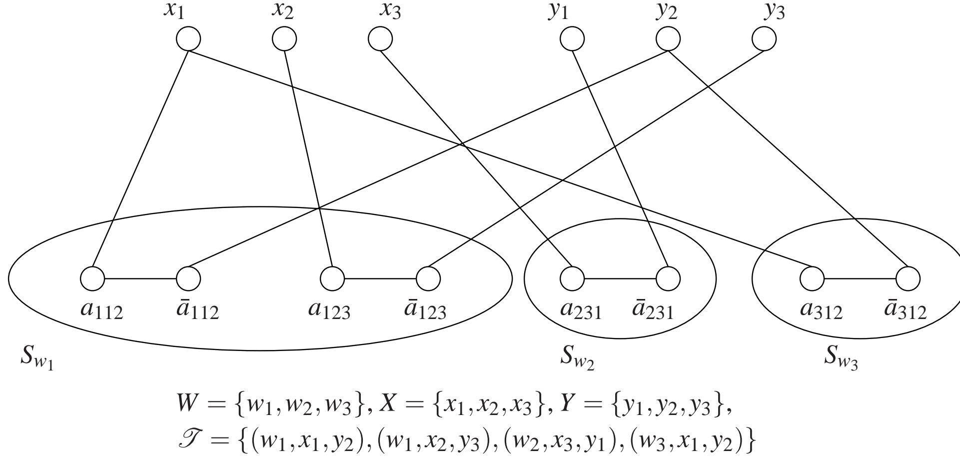


Figure 5.4: Example of the auxiliary graph for an instance of the 3-dimensional matching problem.

Note that each triple $(w_i, x_j, y_k) \in \mathscr{T}$ corresponds to the vertices $\{a_{ijk}, \bar{a}_{ijk}\}$.

Observation 5.1. By the construction of G, each $a_{ijk} \in V$ is only adjacent to $\bar{a}_{ijk}$ and $x_j \in V$. Similarly, each $\bar{a}_{ijk} \in V$ is only adjacent to a_{ijk} and $y_k \in V$. Thus, in any perfect matching M of G, either $(a_{ijk}, \bar{a}_{ijk}) \in M$ or both (x_j, a_{ijk}) and $(y_k, \bar{a}_{ijk}) \in M$.

Observation 5.2. Each a_{ijk} and $\bar{a}_{ijk} \in V$ is contained in the set $S_{w_i} \in \mathscr{S}$. By the construction of G, each $x_j \in V$ is only adjacent to a_{ijk}, $\forall (w_i, x_j, y_k) \in \mathscr{T}$. Similarly, each $y_k \in V$ is only adjacent to $\bar{a}_{ijk}$, $\forall (w_i, x_j, y_k) \in \mathscr{T}$. Thus, every edge adjacent to an x_j or y_k vertex crosses a set $S_{w_i} \in \mathscr{S}$. Thus, given a perfect matching M of G, $\Delta(M) > 0$,

Lemma 5.1. *Given a set $S_{w_i} \in \mathscr{S}$ and a perfect matching M of G, $|M \cap \delta(S_{w_i})|$ is even.*

Proof. $S_{w_i} = \{a_{ijk}, \bar{a}_{ijk} : (w_i, x_j, y_k) \in \mathscr{T}\}$. By the construction of G, the edges crossing S_{w_i} are $\delta(S_{w_i}) = \{(x_j, a_{ijk}), (y_k, \bar{a}_{ijk}) : (w_i, x_j, y_k) \in \mathscr{T}\}$. From Observation 5.1, $(x_j, a_{ijk}) \in M$ if and only if $(y_k, \bar{a}_{ijk}) \in M$. Thus, the edges in $M \cap \delta(S_{w_i})$ can be paired off. □

Corollary 5.1. *Given a perfect matching M of G, $\Delta(M) \geq 2$.*

Proof. By Observation 5.2 and Lemma 5.1. □

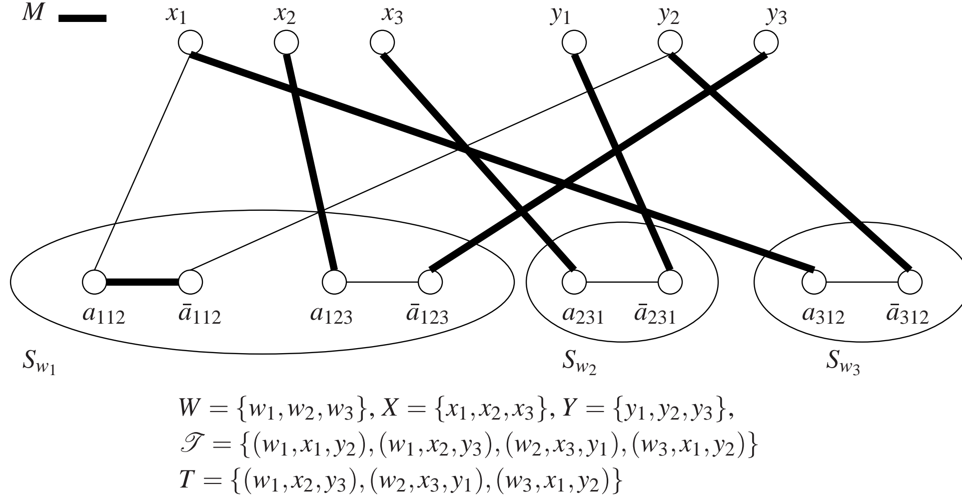
Observation 5.3. $|W| = |X| = |Y| = q$. Let M be a perfect matching of G. By Observation 5.2, the edges adjacent to each $x_j \in X$ and $y_k \in Y$ must cross a set of $\mathscr{S}$. Therefore, there must be at least $2q$ edges in M that cross the sets in $\mathscr{S}$. Since there are q sets, if $\Delta(M) = 2$, then each set in $\mathscr{S}$ has 2 edges crossing it.

The following will show that there is an MCPM M of G, where $\Delta(M) = 2$, if and only if there is a 3DM $T \subseteq \mathscr{T}$. This will be done by showing that the triple $(w_i, x_j, y_k) \in T$ if and only if the edge $(a_{ijk}, \bar{a}_{ijk}) \notin M$.

Theorem 5.1. *Given a bipartite graph $G = (V, E)$ and a family $\mathscr{S}$ of pairwise-disjoint subsets of V, finding an MCPM is NP-hard.*

Proof. Consider an instance of the 3DM problem and its auxiliary graph G. Let $T \subseteq \mathscr{T}$ be a 3DM. Construct a matching M of G as follows. Consider each triple $(w_i, x_j, y_k) \in \mathscr{T}$. If $(w_i, x_j, y_k) \notin T$, let $(a_{ijk}, \bar{a}_{ijk}) \in M$. If $(w_i, x_j, y_k) \in T$, let (x_j, a_{ijk}) and $(\bar{a}_{ijk}, y_k) \in M$. Since each element of X is in exactly one triple in T, for each $x_j \in V$, only one edge in M has x_j as an endpoint. The same holds for each $y_k \in V$. Note that vertices $a_{ijk}, \bar{a}_{ijk} \in V$ correspond to the triple $(w_i, x_j, y_k) \in \mathscr{T}$. Thus, each $a_{ijk} \in V$ is the endpoint of only one edge in M. The same holds for each $\bar{a}_{ijk} \in V$. Thus, M is a matching. Note that every x_j and $y_k \in V$ is the endpoint of some edge in M or else T is not a 3DM. Also note that every a_{ijk} and $\bar{a}_{ijk} \in V$ is the endpoint of some edge in M or else triple $(w_i, x_j, y_k) \notin \mathscr{T}$. Thus, M is a perfect matching. Each set $S_{w_i} \in \mathscr{S}$ corresponds to an element $w_i \in W$. By the construction of M, 2 edges of $\delta(S_{w_i})$ are added to M whenever there is a triple in T containing w_i. Since T is a 3DM, $|M \cap \delta(S_{w_i})| = 2, \forall S_{w_i} \in \mathscr{S}$. Thus, $\Delta(M) = 2$. By Corollary 5.1, M is an MCPM.

Conversely, consider an MCPM M of G where $\Delta(M) = 2$. By Observation 5.3, each set $S_{w_i} \in \mathscr{S}$ has exactly two edges crossing it. By Observation 5.1, the edge $(x_j, a_{ijk}) \in \delta(S_{w_i})$ is in M if and only if $(\bar{a}_{ijk}, y_k) \in \delta(S_{w_i})$ is in M. Since $\Delta(M) = 2$, there is exactly one edge $(a_{ijk}, \bar{a}_{ijk})$, with both endpoints in S_{w_i}, that is not in M, $\forall S_{w_i} \in \mathscr{S}$. When (x_j, a_{ijk}), $(\bar{a}_{ijk}) \in \delta(S_{w_i})$ and $(a_{ijk}, \bar{a}_{ijk}) \notin M$, the corresponding triple in $\mathscr{T}$ is (w_i, x_j, y_k). Choose these corresponding triples to form $T \subseteq \mathscr{T}$. Note that these triples account for every edge in M which crosses a set $S_{w_i} \in \mathscr{S}$. Each $w_i \in W$ occurs in T exactly once since there is exaclty one edge not in M with both endpoints in $S_{w_i} \in \mathscr{S}$. Each $x_j \in X$ occurs in T exactly once since M is a matching and every edge with x_j as an endpoint crosses some set $S_{w_i} \in \mathscr{S}$. Similarly, the same holds for each $y_k \in Y$. Thus, T is a 3DM. □



$W = \{w_1, w_2, w_3\}, X = \{x_1, x_2, x_3\}, Y = \{y_1, y_2, y_3\},$
$\mathscr{T} = \{(w_1, x_1, y_2), (w_1, x_2, y_3), (w_2, x_3, y_1), (w_3, x_1, y_2)\}$
$T = \{(w_1, x_2, y_3), (w_2, x_3, y_1), (w_3, x_1, y_2)\}$

Figure 5.5: An example of a 3DM and the corresponding MCPM

Note that the proof of Theorem 5.1 was based on finding a minimum crossing perfect matching M of G where $\Delta(M) = 2$. By Lemma 5.1, any approximation algorithm for the MCPM problem which finds a perfect matching of G will find a perfect matching M with $\Delta(M)$ even. The next largest even number after 2 is 4. This results in the following corollaries about the hardness of approximating this problem.

Corollary 5.2. *There is no polynomial time $(2-\varepsilon)$-approximation algorithm to the minimum crossing perfect matching problem which finds a perfect matching, unless P = NP.*

Corollary 5.3. *There is no polynomial time +1-approximation algorithm to the minimum crossing perfect matching problem which finds a perfect matching, unless P = NP.*

One can also consider the bounded crossing perfect matching problem, where the number of edges crossing each set is bounded. The proof of the theorem shows that even setting the bound to 2 is still NP-hard. Another problem to consider is the minimum crossing T-join problem.

Theorem 5.2. *The minimum crossing T-join problem is NP-hard.*

Proof. If G has a perfect matching, by setting $T = V$ we get that the T-join problem is NP-hard as well. □

In summary,

Theorem 3. *Finding an MCPM of a graph is NP-hard, even if the graph is bipartite and the family of subsets of vertices is pairwise-disjoint.*

Chapter 6

Open Problems and Further Research

The problem of approximating a general MCST to within a $\log n$ factor is a wide-open problem. Currently, there is no proof of whether such an approximation is possible or not in polynomial time.

The approximation for the special case of finding MCSTs when the family of subsets of vertices is pairwise-disjoint could possibly be reduced to an additive constant. One possible way of achieving this is to extend the methods used by Goemans for BDMSTs [13].

The algorithm for MCSTs on a laminar family of subsets of vertices is dependent on an approximation algorithm for multicommodity flows on trees. The algorithm requires $\log n$ iterations in order to construct the spanning tree. An open problem is finding some way of building the spanning tree in a constant number of iterations. This would eliminate the $\log n$ factor in the approximation guarantee. Of special interest is how approximation algorithms for weighted multicommodity flows on trees [4] could be applied to MCSTs and other related problems.

The algorithm for MCSTs on a laminar family of subsets of vertices could possibly be extended further to finding a $O(f(i) \cdot \log n)$-approximation for the case where the family of subsets of vertices could be partitioned into i laminar families.

This thesis has displayed how ideas for some problems can be applied to approximating MCSTs. Recent ideas like push-relabel [3] and matroid and polyhedral theory [13] being applied to MDMSTs may also be applicable to MCSTs. Even improving special cases of MCSTs such as the pairwise-disjoint sets case may be possible.

A natural extension where little is known is when a cost function is added to the edges. As some algorithms for MDSTs generalize to MDMSTs and BDMSTs, algorithms for MCSTs may extend to approximate minimum crossing minimum spanning trees and bounded crossing minimum spanning trees. Goemans work [13] may apply to the cost version of finding MCSTs where the family of subsets of vertices is pairwise-disjoint. Weighted multicommodity flows on trees [4] may apply to the cost version of finding MCSTs where the family of subsets of vertices is laminar.

Minimum crossing perfect matchings is not a problem that has been considered in detail. Future research could look at proofs on approximation hardness or finding any polynomial time approximation algorithms, even for the special case where the graph is bipartite and the family of subsets of vertices consists of pairwise-disjoint sets. Since perfect matchings are generally difficult to alter, focus on the case of the graph being complete seems be a logical starting point.

Bibliography

[1] V. Bilò and M. Flammini. On the IP routing tables minimization with addresses reassignments. In *Proceedings, 18th International Parallel and Distributed Processing Symposium*, 2004.

[2] V. Bilò, V. Goyal, R. Ravi, and M. Singh. On the crossing spanning tree problem. In *Proceedings, International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, pages 51–60, 2004.

[3] K. Chaudhuri, S. Rao, S. Riesenfeld, and K. Talwar. A push-relabel algorithm for approximating degree bounded MSTs. In *Proceedings, 33rd International Colloquium on Automata, Languages and Programming, Part I*, Lecture Notes in Computer Science, pages 191–201. Springer, 2006.

[4] C. Chekuri, M. Mydlarz, and F. B. Shepherd. Multicommodity demand flow in a tree. In *Proceedings, 30th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 410–425. Springer, 2003.

[5] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. John Wiley & Sons Inc., New York, 1998.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 1989.

[7] T. Fischer. Optimizing the degree of minimum weight sapnning trees. Technical Report TR 93-1338, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, 1993.

[8] M. Fürer and B. Raghavachari. An NC approximation algorithm for the minimum degree spanning tree problem. In *Proceedings of the 28th Annual Allerton Conference on Communication, Control and Computing*, pages 274–281, 1990.

[9] M. Fürer and B. Raghavachari. Approximating the minimum degree spanning tree to within one from the optimal degree. In *Proceedings, Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 317–324, New York, 1992. ACM.

[10] M. Fürer and B. Raghavachari. Approximating the minimum-degree Steiner tree to within one of optimal. *Journal of Algorithms*, 17(3):409–423, November 1994.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[12] N. Garg, V. V. Vazirani, and M. Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18:3–20, 1997.

[13] M. Goemans. Bounded degree minimum spanning trees. In *47th Annual IEEE Symposium on Foundations of Computer Science*, Berkeley, 2006.

[14] A. V. Goldberg. A new max-flow algorithm. Technical Report MIT/LCS/TM-291, Massachussets Institute of Technology, 1985.

[15] D. Greenberg and S. Istrail. Physical mapping by STS hybridization: Algorithmic strategies and the challenge of software evaluation. *Journal of Computational Biology*, 2(2):219–273, 1995.

[16] J. Könemann and R. Ravi. A matter of degree: Improved approximation algorithms for degree-bounded minimum spanning trees. *SIAM J. Comput.*, 31(6):1783–1793, 2002.

[17] J. Könemann and R. Ravi. Primal-dual meets local search: Approximating MSTs with nonuniform degree bounds. *SIAM J. Comput.*, 34(3):763–773, 2005.

[18] R. Krishnan and B. Raghavachari. The directed minimum-degree spanning tree problem. In *Proceedings, Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 2001.

[19] R. Ravi, M. V. Marathe, S. S. Ravi, D. J. Rosenkrantz, and H. B. Hunt. Many birds with one stone: Multi-objective approximation algorithms. In *Proceedings, ACM Symposium on Theory of Computing*, pages 438–447, 1993.