The Geometric Structure of Spanning Trees and Applications to Multiobjective Optimization

By

ALLISON KELLY O'HAIR

SENIOR THESIS

Submitted in partial satisfaction of the requirements for Highest Honors for the degree of

BACHELOR OF SCIENCE

in

MATHEMATICS

in the

COLLEGE OF LETTERS AND SCIENCE

of the

UNIVERSITY OF CALIFORNIA,

DAVIS

Approved:

Jesús A. De Loera

Peter N. Malkin

June 2009

ABSTRACT. In this senior thesis, we study many different properties of spanning trees, including the graph of tree exchanges. Using this graph, we then study multiobjective optimization with regards to the edge costs. We implemented a program to enumerate all spanning trees, in order to assess the accuracy of our optimization algorithms and heuristics. We also studied the general case for matroids and wrote a program to estimate the number of bases of matroid polytopes. We consider several fast heuristics that can find the minimum spanning tree for a graph with respect to multiple sets of edge costs, particularly finding the Pareto optima. Although these heuristics could potentially only locate a local minimum, they locate the global minimum in almost every trial, and are extremely efficient. The results of this undergraduate thesis were incorporated into the research paper "Computation in Multicriteria Matroid Optimization [**8**]," coauthored by Jesús De Loera, David Haws, Jon Lee, and myself, and the software package MOCHA [**5**], where we discuss and experiment with multicriteria matroid optimization.

Contents

CHAPTER 1

The Geometric Structure of Trees

1.1. Basic Definitions

In order to motivate the research in this senior thesis, we begin with some basic definitions. For further detail and more explanation on properties of trees and graphs we refer the reader to [**12**].

The central concept of the research presented here is that of a spanning tree. A *spanning tree* of a graph G is a connected subgraph without cycles that includes every vertex of G, assuming G is connected. Otherwise, G has *spanning forests*, which are maximal subgraphs without cycles (so a $forest$ is a subgraph without cycles). Here we will discuss spanning trees since we are only concerned with connected graphs. Figure 1 gives an example of a spanning tree of a graph. The subgraph on the right is a spanning tree of the graph on the left.

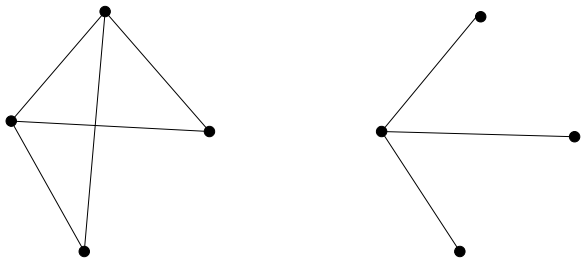


FIGURE 1. A graph(left) and a spanning tree of the graph

While there are many interesting properties of the set of spanning trees of unweighted graphs (some of which we will look at shortly), we are also very interested in properties of the spanning trees of a weighted graph. Weighted graphs are used in many different situations, where the weights can represent the costs of building a network of roads, the length of the paths between each node, or some other measure of interest. When each edge of a graph is given a cost, we can find the spanning tree with smallest cost. The minimum spanning tree of G, the tree with the smallest cost, can be found by applying Kruskal's

algorithm or Prim's algorithm, both of which find solutions to the minimum spanning tree problem [**4**]. The steps to these algorithms are as follows:

Kruskal's Algorithm:

- Step 1: Sort the edges of the graph G such that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$.
- Step 2: Set $T = (V(G), \emptyset)$.
- Step 3: For $i = 1$ to m: If $T + e_i$ contains no circuit then set $T = T + e_i$.

Prim's Algorithm:

- Step 1: Choose $v \in V(G)$. Set $T = (\{v\}, \emptyset)$.
- Step 2: While $V(T) \neq V(G)$: Choose an edge e that has only one end in $V(T)$ of minimum weight. Set $T = T + e$.

Below is an example of a weighted graph and the steps taken in both Kruskal's and Prim's algorithms when finding the minimum spanning tree of this graph. Figure 2 gives an example of a graph with edge-costs, Figure 3 demonstrates the steps of Kruskal's Algorithm, and Figure 4 demonstrates the steps of Prim's Algorithm. Although they both reach the same minimum spanning tree in this case because it is unique, it is possible for the two algorithms to arrive at two different minimum spanning trees when it is the case that the minimum spanning tree is not unique.

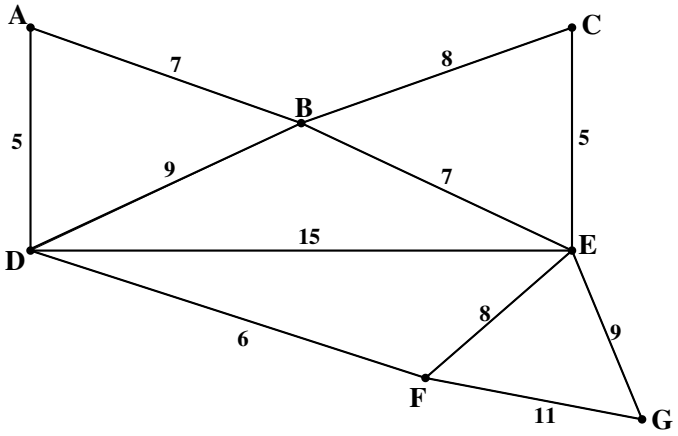


FIGURE 2. A graph with edge costs.

Furthermore, all spanning trees of G can be found by applying Kruskal's (or Prim's) algorithm to G while all variations of edge costs are considered. This can easily be seen due to the fact that the minimum spanning tree is determined by the edge costs, and if these are varied, then the minimum spanning tree will be varied as well. Thus if we vary the edge costs enough, we will find all spanning trees.

We are also interested in the number of spanning trees of a graph. In most cases there is not a simple formula for the number of spanning trees of a graph, but in the case of

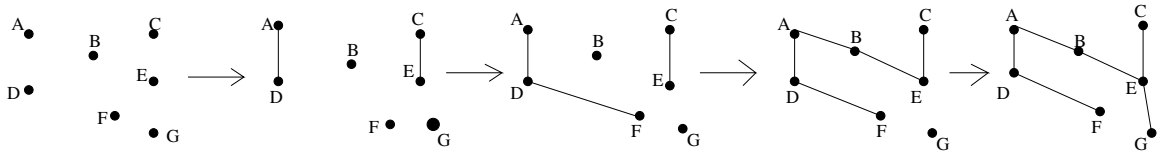


FIGURE 3. The minimum spanning tree of the above graph found using Kruskal's algorithm.

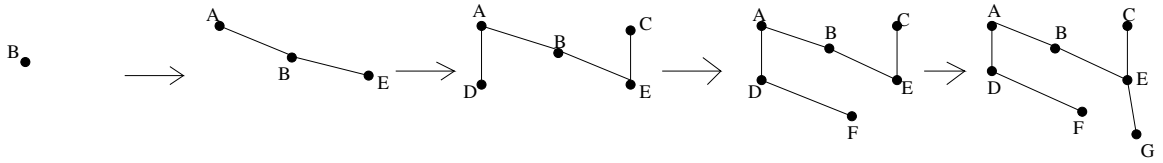


FIGURE 4. The minimum spanning tree of the above graph found using Prim's algorithm.

the complete graph we have a simple formula due to Cayley's Theorem [**9**]. The complete graph K_n is the graph on n nodes in which every pair of vertices forms an edge.

THEOREM 1. *(Cayley's Theorem) The number of spanning trees of the complete graph K_n is $n^{(n-2)}$.*

The proof of Cayley's Theorem presented here is due to Jim Pitman, who used clever counting in two ways to prove the theorem[**1**]. Cayley's theorem can also be proved using the Matrix Tree Theorem[**12**] presented below, but here we give an alternate proof to show a different way of proving the theorem. Before presenting the proof, we need to state a few definitions. A *rooted forest* on the vertex set $\{1, \ldots, n\}$ is a forest together with a choice of a root in each component tree. Additionally, a forest F is said to *contain* another forest F' if F contains F' as a directed graph. Then clearly if F properly contains F', then F has fewer components than F'. And finally, let $\mathcal{F}_{n,k}$ be the set of all rooted forests that consist of k rooted trees. Then we call a sequence $F_1, \ldots, F_k$ of forests a *refining sequence* if $F_i \in \mathcal{F}_{n,i}$ and F_i contains F_{i+1}, for all i. Now we can begin the proof.

PROOF. First we notice that $\mathcal{F}_{n,1}$ is the set of all rooted trees (since they only consist of one component). Note that $|\mathcal{F}_{n,1}| = nT_n$, since in every tree there are n choices for the root. We now regard $F_{n,k} \in \mathcal{F}_{n,k}$ as a directed graph with all of the edges directed away from the roots.

Now let F_k be a fixed forest in $\mathcal{F}_{n,k}$ and denote by $N(F_k)$ the number of rooted trees containing F_k, and by $N^*(F_k)$ the number of refining sequences ending in F_k.

We count $N^*(F_k)$ in two ways, first by starting at a tree and secondly by starting at F_k. Suppose $F_1 \in \mathcal{F}_{n,1}$ contains F_k. Since we may delete the $k-1$ edges of $F_1 \setminus F_k$ in any possible order to get a refining sequence from F_1 to F_k, we find

$$N^*(F_k) = N(F_k)(k-1)!.$$

Let us now start at the other end. To produce from F_k an F_{k-1} we have to add a directed edge, from any vertex a, to any of the $k-1$ roots of the trees that do not contain a. Thus we have $n(k-1)$ choices. Similarly, for F_{k-1} we may produce a directed edge from any vertex b to any of the $k-2$ roots of the trees not containing b. For this we have $n(k-2)$ choices. Continuing this way, we arrive at

$$N^*(F_k) = n^{k-1}(k-1)!,$$

and thus we have that

$$N(F_k) = n^{k-1} \text{ for any } F_k \in \mathcal{F}_{n,k}.$$

For $k = n$, F_n consists of just n isolated vertices. Hence $N(F_n)$ counts the number of all rooted trees, thus $|\mathcal{F}_{n,1}| = n^{n-1}$, and thus $T_n = (1/n)(|\mathcal{F}_{n,1}|) = n^{n-2}$. □

For other graphs, such as the complete bipartite graph, we can enumerate all spanning trees by finding each individually. The complete bipartite graph $K_{n,m}$ is a graph in which the vertex set can be partitioned into two independent sets of size n and m and every vertex in each set is connected by an edge to every vertex in the other set. This enumeration of spanning trees is shown in Figure 5, for $K_{2,3}$.

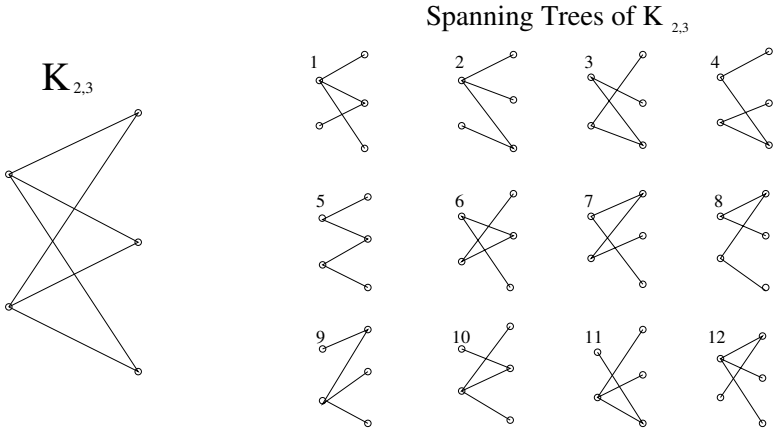


FIGURE 5. All spanning trees of $K_{2,3}$.

Another useful way to count the number of spanning trees of a graph is by the Matrix Tree Theorem [**12**], of which Cayley's Theorem is a generalization. Before stating the theorem, we will state a few definitions that are important in the theorem and the proof. In everything we have discussed so far and throughout the remainder of this senior thesis, we assume we are only dealing with *simple graphs*, graphs without self-loops or multiple edges between the same two vertices. The Matrix Tree Theorem also uses the *adjacency*

matrix A of a graph, an $n \times n$ matrix where n is the number of nodes in the graph, and $A_{ij} = 1$ if there is an edge between nodes i and j and $A_{ij} = 0$ otherwise. The proof uses the *incidence matrix* M of a graph, an $n \times m$ matrix where m is the number of edges of G, with $M_{ij} = 1$ if node i is an endpoint of edge j and $M_{ij} = 0$ otherwise. If the graph is directed, then $M_{ij} = 1$ if i is the tail of edge j, $M_{ij} = -1$ if i is the head of edge j, and $M_{ij} = 0$ otherwise. Now we can state the Matrix Tree Theorem:

THEOREM 2. *(Matrix Tree Theorem) Let A be the adjacency matrix of a graph G, let D be a diagonal matrix with the diagonal entry in row i equal to the degree of vertex i, and let $Q = D - A$. Then for any s, the number of spanning trees of G equals the determinant of the matrix Q^* obtained by deleting row s and column s of Q.*

PROOF. First, we show that if G' is an orientation of G and M is the incidence matrix of G' then $Q = MM^T$. Label the now directed edges by $e_1, \ldots, e_m$. By definition of the incidence matrix, since every entry in the n by n matrix MM^T is the dot product of rows of M, diagonal entries in the product count vertex degrees and off-diagonal entries count -1 for every edge of G between two vertices.

Now we want to show that if B is an $(n-1) \times (n-1)$ submatrix of M, then $detB = 0$ if the corresponding $n-1$ edges contain a cycle, and $detB = \pm 1$ if they form a spanning tree of G. If the edges corresponding to the columns contain a cycle C, then the columns sum to the zero vector when weighted with $+1$ or -1 determined by whether the directed edge is followed forward or backward when following the cycle. This column dependency implies $detB = 0$.

For the other case, we use induction on n. For $n = 1$, by convention a 0×0 matrix has determinant 1. Now suppose $n > 1$, and let T be the spanning tree whose edges are the columns of B. Since T has at least two leaves, B contains a row corresponding to a leaf x of T. This row has only one nonzero entry in B. When computing the determinant by expanding along that row, the only submatrix B' given nonzero weight in the expansion corresponds to the spanning subtree of $G - x$ obtained by deleting x and its incident edge from T. Since B' is an $(n-2) \times (n-2)$ submatrix of the incidence matrix for an orientation of $G-x$, the induction hypothesis implies that the determinant of B' is ± 1, and multiplying it by ± 1 gives the same result for B.

Finally, we need to compute $detQ^*$. Let M^* be the matrix obtained by deleting row t of M, so $Q^* = M^*(M^*)^T$. We may assume $m \geq n-1$, else both sides have determinant 0 and there are no spanning subtrees. The Binet-Cauchy formula expresses the determinant of a product of matrices, not necessarily square, in terms of the determinants of submatrices of the factors. In particular, if $m \geq p$, A is a $p \times m$ matrix, and B is an $m \times p$ matrix, then $detAB = \sum_S detA_S detB_S$, where the summation runs over all $S \subseteq [m]$ consisting of p indices, A_S is the submatrix of A having the columns indexed by S, and B_S is the submatrix of B having the rows indexed by S. When we apply the Binet-Cauchy formula to $Q^* = M^*(M^*)^T$, the submatrix A_S is an $(n-1) \times (n-1)$ submatrix of M as discussed

before, and $B_S = A_S^T$. Hence the summation counts $1 = (\pm 1)^2$ for each set of $n-1$ edges corresponding to a spanning tree and 0 for each other set of $n-1$ edges. □

EXAMPLE 1. Consider the graph in Figure 2. The adjacency matrix A for this graph is the 7×7 matrix

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

And the diagonal matrix D is the 7×7 matrix

$$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

Then the matrix $Q = D - A$ is the 7×7 matrix

$$\begin{pmatrix} 2 & -1 & 0 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 & 0 \\ 0 & -1 & 2 & 0 & -1 & 0 & 0 \\ -1 & -1 & 0 & 4 & -1 & -1 & 0 \\ 0 & -1 & -1 & -1 & 5 & -1 & -1 \\ 0 & 0 & 0 & -1 & -1 & 3 & -1 \\ 0 & 0 & 0 & 0 & -1 & -1 & 2 \end{pmatrix}$$

Now let $s = 3$. By deleting the sth row and sth column of Q we have the 6×6 matrix

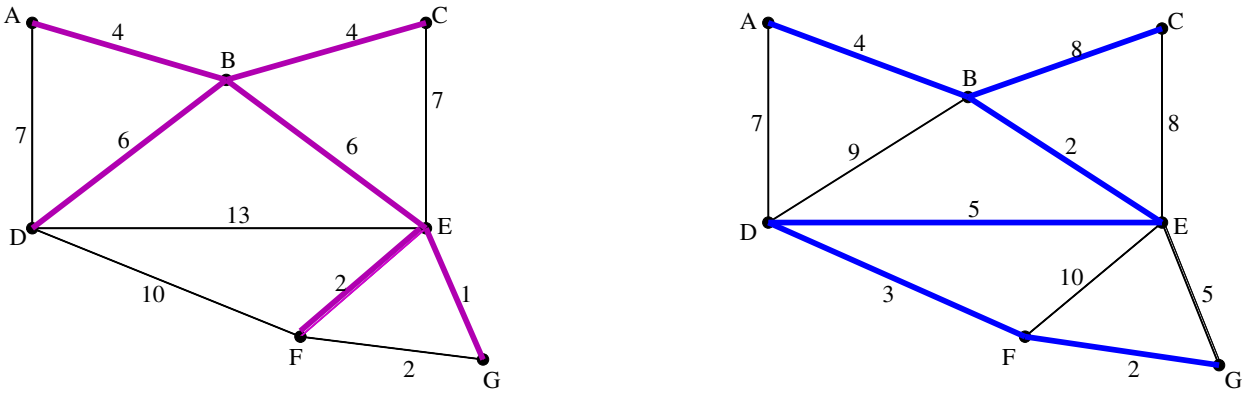
$$\begin{pmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & 0 & 0 \\ -1 & -1 & 4 & -1 & 0 & 0 \\ 0 & -1 & -1 & 5 & -1 & -1 \\ 0 & 0 & -1 & -1 & 3 & -1 \\ 0 & 0 & 0 & -1 & -1 & 2 \end{pmatrix}$$

The determinant of this matrix is 141. Thus the Matrix Tree Theorem states that the number of spanning trees for this graph is 141. These computations were performed by using Maple^{TM}.

This theorem is used in Maple^{TM} to quickly count the number of spanning trees of any graph, which was used many times throughout the research in this senior thesis.

1.2. The Graph of Tree Exchanges

Consider now the case when a graph has more than one set of edge weights. This situation occurs in numerous practical applications. The one that motivated this research is as follows. Suppose that a new network of roads is to be built that will connect a certain number of cities. We know all of the possible connections that can be built, but we just want a network of roads so that it is possible to reach any city from any other. In other words, we want a spanning tree. The government is interested in building the cheapest network of roads possible, so they let each edge weight equal the cost of building that road segment. But on the other hand, an environmental organization is concerned with the environmental impact of building each road. So they have a completely different set of edge costs. Now, given these two conflicting sets of edge weights, we need to find a spanning tree that satisfies both parties. This situation is illustrated in Figure 6.



Cost of Minimum Spanning Tree: 23

Cost of Minimum Spanning Tree : 24

FIGURE 6. Two different sets of edge weights.

We can see from the figure that we have a problem. There is a different minimum spanning tree for each set of edge weights. We need one spanning tree that is optimal or close to optimal for both sets of edge weights. In this case, we can no longer use Prim's or Kruskal's algorithms since we need to simultaneously optimize the edge costs of both parties. We can also see that if we continue to add more edge weights, like a third party

in the example above, the problem will only get worse. Thus we need some way to find the best possible spanning tree with respect to all of the edge weights. In order to improve the efficiency of this task, we will use a graph called the Graph of Tree Exchanges, which we explain now. In the next chapter, we will explain the basic notions of multiobjective optimization.

DEFINITION 1. The *graph of tree exchanges*, $T(\Gamma)$ for a graph Γ, has the set of nodes $N = \{n_t \in ST\}$, where ST is the set of spanning trees of Γ, and two nodes n_i and n_j are adjacent if the spanning trees n_i and n_j have only one differing edge in a common cycle.

Figure 7 gives an example of two nodes that would be adjacent and two nodes that would not be adjacent in the graph of tree exchanges for the complete bipartite graph $K_{2,3}$.

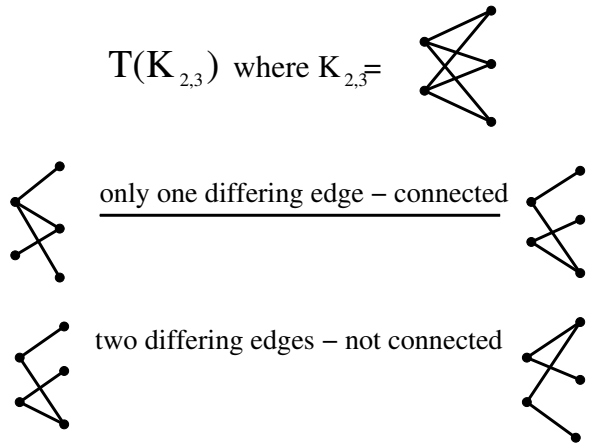


FIGURE 7. Example of connected and unconnected nodes in $T(K_{2,3})$

Figure 8 is an example of the graph of tree exchanges for the complete graph K_4. The graph itself is shown in the upper left corner, and the spanning tree that each node represents is next to the node. Even for a graph this small, it can be seen that the graph of tree exchanges has many connections, thus it appears to be a dense graph.

We are interested in the graph of tree exchanges not only for multiobjective optimization, but also because we can determine interesting properties of the set of all spanning trees by investigating this graph. Thus in the next section we will investigate some properties of the graph of tree exchanges.

1.3. Properties of the Graph of Tree Exchanges

In this section we will investigate some properties and characteristics of the graph of tree exchanges. We first look at the connection between the graph of tree exchanges and

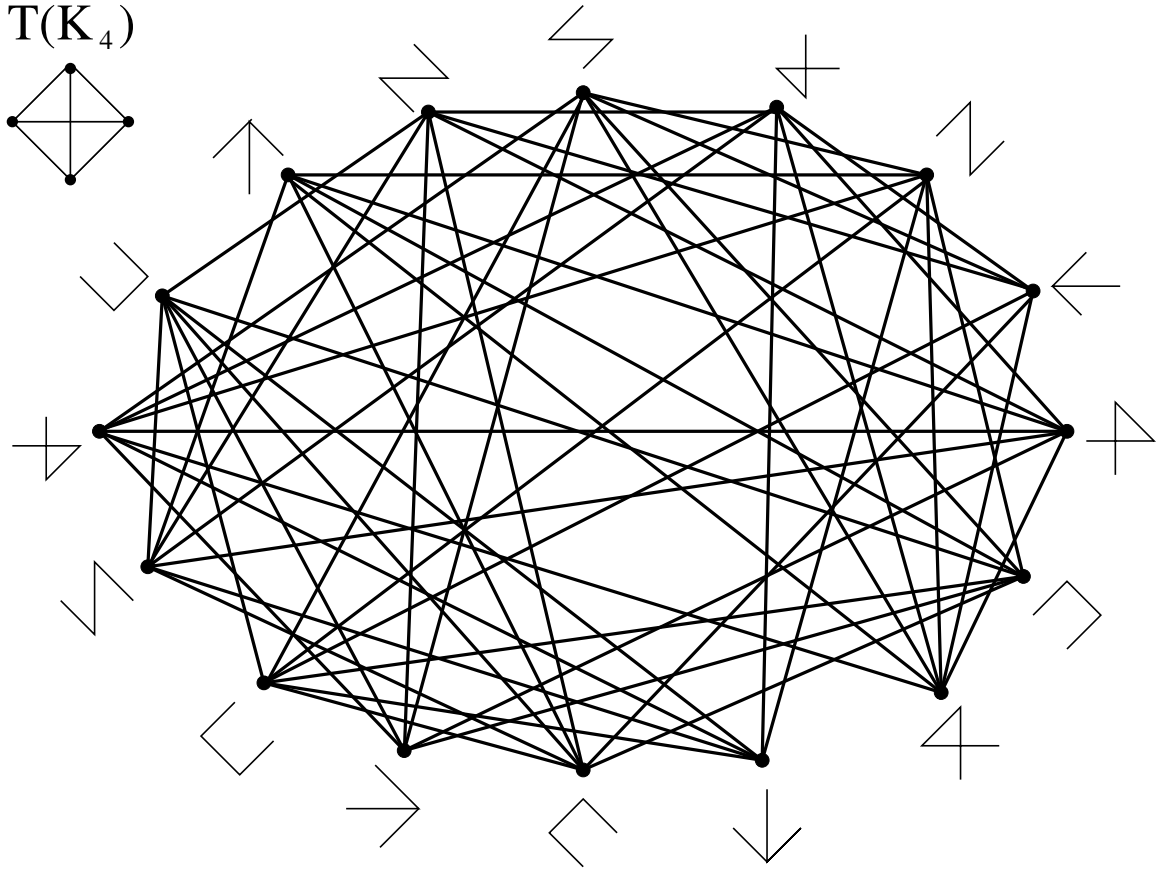


FIGURE 8. The graph of tree exchanges of the complete graph K_4

linear programming, then define some properties of graphs that we will investigate for the graph of tree exchanges, and finally, in the next section, we will give the results from our investigations.

There is an interesting connection between spanning trees and linear programming. Namely, for the following linear program every minimum spanning tree provides an optimal solution with respect to the edges costs c [**4**]:

$$\textit{Minimize } c^T x$$
$$\textit{subject to}$$
$$\sum x_{ij} = |N| - 1,$$
$$\sum_{i,j \in S} x_{ij} \leq |S| - 1, \quad S \subseteq V(\Gamma),$$
$$x_{ij} \geq 0.$$

The following theorem is a result of closely examining the above linear program:

THEOREM 3. *Let x^0 be the characteristic vector of a minimum spanning tree with respect to costs c. Then x^0 is an optimal solution of the linear program above. Furthermore, the characteristic vectors of all spanning trees are feasible solutions to the linear program* [**4**].

From this theorem we can conclude that if the edge costs are varied, we can find all of the spanning trees of a graph by repeatedly solving this linear program. Additionally, from this theorem we have the following corollary:

COROLLARY 1. $T(\Gamma)$ is the graph of a polytope, which is the convex hull of the characteristic vectors of the spanning trees of Γ[**4**].

We investigated several properties of the Graph of Tree Exchanges, all of which are defined here. We give examples of each of the properties on a simple graph of tree exchanges, that for the graph $K_{2,3}$. This graph is shown in Figure 9, where the numbers next to the vertices correspond to the numbers next to the spanning trees in Figure 5 of Chapter 1.

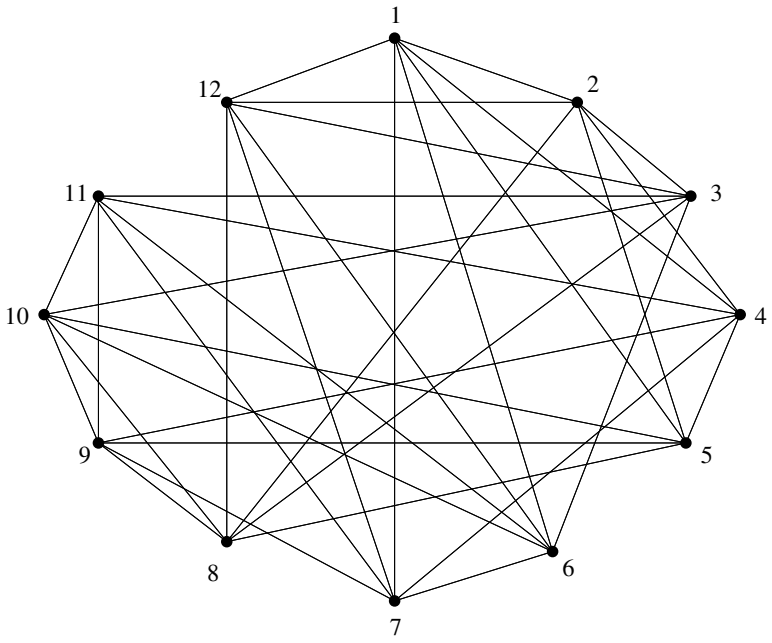


FIGURE 9. The graph of tree exchanges for $K_{2,3}$.

DEFINITION 2. A graph is *Hamiltonian* if it contains a cycle that visits every vertex of the graph exactly once.

EXAMPLE 2. Figure 10 shows a Hamiltonian cycle of the graph above. The hamiltonian cycle is outlined in red.

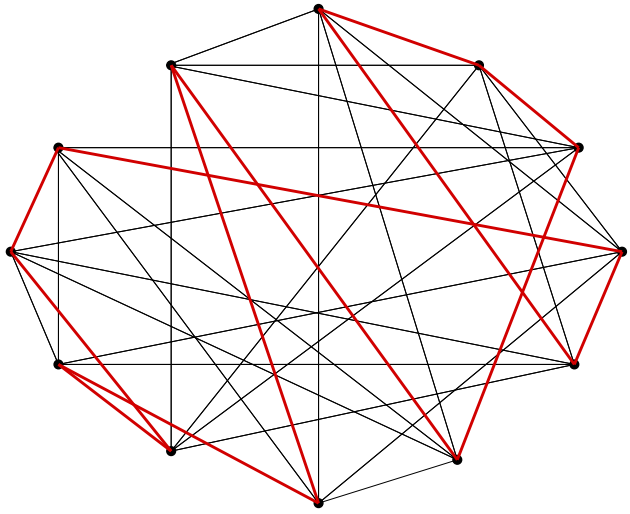


FIGURE 10. Hamiltonian cycle of a graph.

DEFINITION 3. A graph is *Eulerian* if it contains a cycle that visits every edge of the graph exactly once.

EXAMPLE 3. Figure 11 shows an Eulerian cycle of the graph above. The cycle starts at the vertex labeled Start and cycles through the edges by number.

DEFINITION 4. The *diameter* of a graph is defined as the maximum of the set of all shortest walks joining any two vertices.

EXAMPLE 4. The diameter of the graph above is 2. This is because for any vertex not directly connected to another, there is a path of length two connecting the two. By looking at the graph, it can be seen that this is true.

DEFINITION 5. The *Maximum Independent Set* of a graph is the maximum subset of vertices where no two of the vertices define an edge in the graph.

EXAMPLE 5. The maximum independent set of the graph above is 3. This is a difficult number to compute (it is known to be NP-complete to find the maximum independent set of a graph[**12**]) and was determined by examining all possible independent sets.

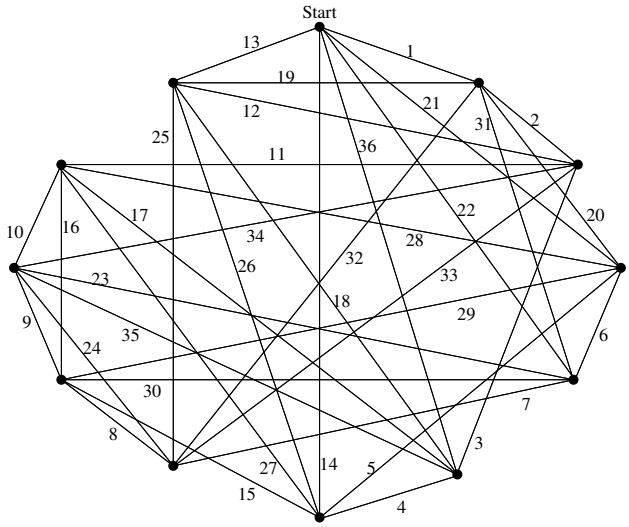


FIGURE 11. Eulerian cycle of a graph.

DEFINITION 6. A graph contains a *Perfect Matching* if it has a set of pairwise disjoint edges where every node belongs to one of the edges.

EXAMPLE 6. Figure 12 shows a perfect matching of the graph above. The blue edges are those that define the matching.

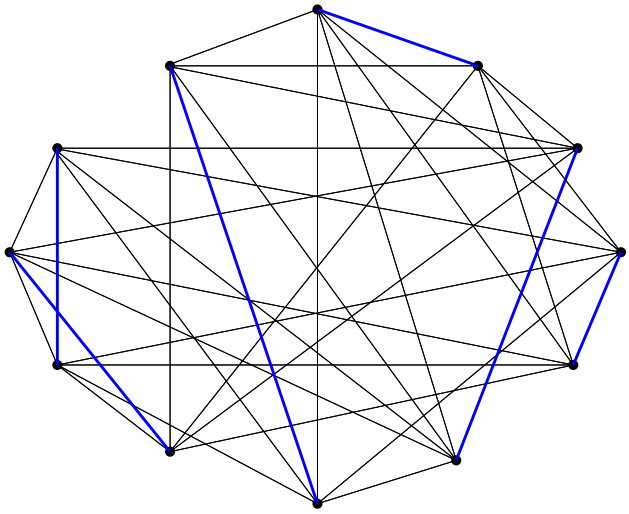


FIGURE 12. A perfect matching of a graph.

1.4. Results

With the assistance of Maple^{TM}, we were able to determine the above properties of various graphs of tree exchanges: the diameter, the maximum independent set, and the existence of a Hamiltonian cycle, an Eulerian cycle, or a perfect matching. We have collected the results in Tables 1 and 2.

The left most column gives the graph that was studied, i.e. $T(K_4)$ is the graph of tree exchanges for the complete graph on 4 nodes. The notation for the degree sequence is as follows: $[6_4, 7_{12}]$ means the graph has 4 nodes of degree 6 and 12 nodes of degree 7. All of the values for the maximum independent set are not included in the table since we did not obtain all of them, due to the problem being NP-complete, as mentioned earlier. The last four graphs, $\Gamma_1, \Gamma_2, \Gamma_3$, and Γ_4 are shown in Figure 13.

TABLE 1. Number of nodes, degree sequence, and diameter of each graph of tree exchanges.

$T(\Gamma)$	**Num. of nodes**	**Degree Sequence**	**Diameter**
$T(K_{2,3})$	12	$[6_{12}]$	2
$T(K_{2,4})$	32	$[9_{32}]$	3
$T(K_{2,5})$	80	$[12_{80}]$	4
$T(K_{3,3})$	81	$[12_{45}, 14_{36}]$	4
$T(K_4)$	16	$[6_4, 7_{12}]$	3
$T(K_5)$	125	$[12_5, 14_{60}, 16_{60}]$	4
$T(\Gamma_1)$	15	$[6_9, 8_6]$	2
$T(\Gamma_2)$	21	$[6_4, 7_{12}, 8, 9_4]$	3
$T(\Gamma_3)$	56	$[9_{18}, 11_{30}, 13_2, 15_6]$	3
$T(\Gamma_4)$	55	$[8_4, 9_{20}, 10_{13}, 11_{12}, 12_2, 14_4]$	4

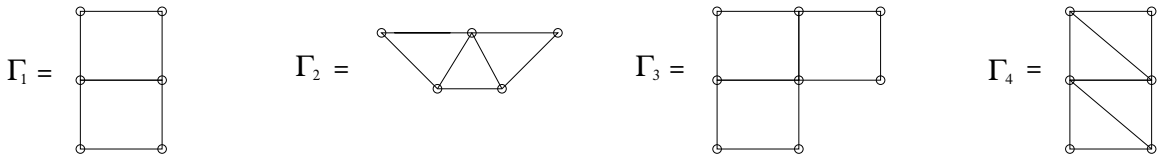


FIGURE 13. The four special graphs considered above.

With these results, we independently made the conjecture that any graph of tree exchanges is Hamiltonian and thus will have a perfect matching if the number of spanning trees is even. This is due to the fact that you can just take every other edge in the Hamiltonian cycle to form the perfect matching. This result was also discovered by Takahiko Kamae in [**7**] and stated in the following theorem.

TABLE 2. Existence of a Hamiltonian cycle, existence of an eulerian cycle, the maximum independent set, and existence of a perfect matching for each graph of tree exchanges.

$T(\Gamma)$	**Hamiltonian**	**Eulerian**	**Max. Ind. Set**	**Per. Mat.**
$T(K_{2,3})$	Yes	Yes	3	Yes
$T(K_{2,4})$	Yes	No	6	Yes
$T(K_{2,5})$	Yes	Yes		Yes
$T(K_{3,3})$	Yes	Yes		No
$T(K_4)$	Yes	No	4	Yes
$T(K_5)$	Yes	Yes		No
$T(\Gamma_1)$	Yes	Yes	3	No
$T(\Gamma_2)$	Yes	No	5	No
$T(\Gamma_3)$	Yes	No		Yes
$T(\Gamma_4)$	Yes	No		No

THEOREM 4. *Let e (or v) be the number of edges (or vertices) in a given graph. Let t_1 and t_2 be adjacent trees. Then the tree graph associated with any graph with $e \geq 3$ has a Hamiltonian cycle with respect to t_1 and t_2 (a Hamiltonian cycle in which trees t_1 and t_2 are adjacently connected).*

Through my extensive computational experiments, I also made the following conjecture:

CONJECTURE 1. *The diameter of the graph of tree exchanges for the complete graph on n nodes is $(n-1)$ and the diameter of the graph of tree exchanges for the complete bipartite graph $K_{2,m}$ is also $(n-1)$.*

I did not discover a proof for this conjecture, but it holds for each of the graphs that I studied. Through these results we were able to learn more about the graph of tree exchanges to better understand its use in multiobjective optimization. In the next chapter we will investigate multiobjective optimization while still focusing on spanning trees and using the graph of tree exchanges.

CHAPTER 2

Enumeration of Trees and Bases and Applications to Multiobjective Optimization

2.1. Introduction

In this chapter, we will focus on multiobjective optimization while discussing spanning trees, as well as enumeration of spanning trees. I will also make the connection between graphs, spanning trees, and matroids while introducing enumeration of matroid bases. While the focus of this senior thesis is on ideas relating to spanning trees, more general results are in the paper [**8**]. I will begin by introducing the basic notions and definitions needed for multiobjective optimization.

The multiobjective optimization methods used in [**8**] use the adjacency structure inherent in the graph of tree exchanges to pivot to feasible solutions which may or may not be optimal. The paper [**8**] presents a modified breadth-first-search heuristic that uses tree adjacency to enumerate a subset of feasible solutions, other heuristics, and computational evidence supporting these new techniques. We implemented all of our algorithms in the software package MOCHA [**5**]. In this senior thesis, I will just discuss the method of finding the Pareto optima.

Here is the setup for multiobjective optimization: We consider the case where we give d weightings $w_1, ..., w_d \in \mathbf{R}^n$ to the n edges of the graph. That is, every w_i assigns a real-value to each element of $[n]$. We let $W \in \mathbf{R}^{d\times n}$ be the matrix with rows $w_1, ..., w_d$. For each spanning tree S of the graph, we define the *incidence vector* of S as $e_S := \sum_{i\in S} e_i \in \mathbf{R}^n$. Thus by taking the inner product of any weighting vector and the incidence vector of a spanning tree we can find the cost of that spanning tree with respect to one of the edge weightings. In order to clarify this idea, the following is a simple example.

Example 7. Consider the graph with two sets of edge weights introduced in Chapter 1. In this case $d = 2$, and $n = 11$ since the graph has 11 edges.

Let us order the edges $(AB, AD, BC, BD, BE, CE, DE, DF, EF, EG, FG)$. Then the first weighting w_1 is $(4, 7, 4, 6, 6, 7, 13, 10, 2, 1, 2)$ and the second weighting w_2 is $(4, 7, 8, 9, 2, 8, 5, 3, 10, 5, 2)$. So the matrix W is this case is 2×11 matrix

$$\begin{pmatrix} 4 & 7 & 4 & 6 & 6 & 7 & 13 & 10 & 2 & 1 & 2 \\ 4 & 7 & 8 & 9 & 2 & 8 & 5 & 3 & 10 & 5 & 2 \end{pmatrix}$$

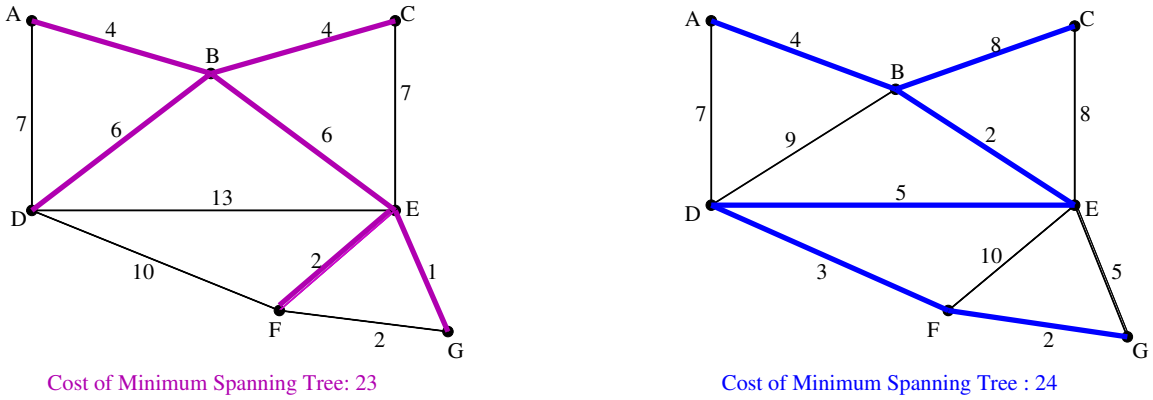


FIGURE 1. An example of a graph with two sets of edge weights.

The incidence vector for the minimum spanning tree on the left, S_1, is $e_{S_1} = (1,0,1,1,1,0,0,0,1,1,0)$, and the incidence vector for the minimum spanning tree on the right, S_2, is $e_{S_2} = (1,0,1,0,1,0,1,1,0,0,1)$. Thus the cost of S_1 with respect to the weighting w_1 is the dot product of e_{S_1} and w_1, $e_{S_1} \bullet\ w_1 = 23$. Similarly, the cost of S_1 with respect to w_2 is $e_{S_1} \bullet\ w_2 = 38$, the cost of S_2 with respect to w_1 is $e_{S_2} \bullet\ w_1 = 39$, and the cost of S_2 with respect to w_2 is $e_{S_2} \bullet\ w_2 = 24$.

Now, let the set of all spanning trees of the graph be S_M. We also define $P_M := conv(e_S | S \in S_M) \subseteq \mathbf{R}^n$, thus P_M is the polytope previously discussed formed from all of the incidence vectors of the spanning trees. And then using the d weightings, we can define $WP_M := \{We_S | S \in S_M\} \subseteq \mathbf{R}^d$, which is the polytope P_M projected to dimension d. As discussed in Chapter 1 for the figure above, one can think of the rows of W as the set of criteria that (possibly conflicting) parties may bring to a discussion.

There are many different techniques to select the optimal spanning tree or set of optimal spanning trees using different tie-breaking criteria. In the next section we will discuss Pareto optimization, which is one of these possible techniques. It is necessary to use some tie-breaking criteria because given the different weightings, a different spanning tree might be best for each edge weighting. This is the problem discovered in Chapter 1 while looking at the example with just two different weightings.

In the following sections I will introduce Pareto optimization, an algorithm for enumerating all spanning trees of a graph which we use to check our calculations, and an algorithm for estimating the number of bases of a matrix.

2.2. Pareto Optimum

The definition of Pareto Multi-criteria Optimization for graphs is as follows.

Pareto Multi-criteria Graphical Optimization: Given a graph G with n edges and set of spanning trees S_M, $W \in \mathbf{R}^{d \times n}$, find all spanning trees $S \in S_M$ such that $S = argmin_{Pareto}((We_{S'})|S' \in S_M)$.

In this statement, $\min_{Pareto}$ is understood in the sense of Pareto optimality for problems with multiple objective functions, namely, we adopt the convention that for vectors $a, b \in \mathbf{R}^d$, we have $a \leq b$ if and only if $a_i \leq b_i$ for all entries of the vectors. Furthermore, we say that $a < b$ if $a \leq b$ and $a \neq b$. The Pareto multi-criteria matroid optimization problem has been studied by several authors before. For example, Ehrgott [**6**] investigated two optimization problems for matroids with multiple objective functions, and he pioneered a study of Pareto bases via the base-exchange property of matroids.

EXAMPLE 8. Consider the following simple graph:

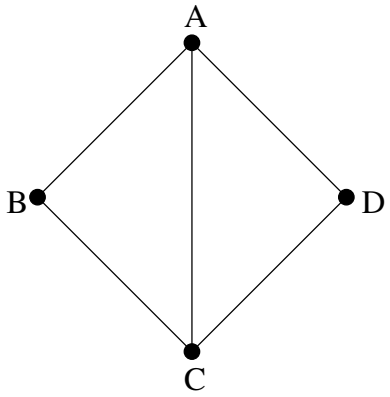


FIGURE 2. A simple graph.

Let the edges be ordered (AB, AC, AD, BC, CD). There are eight spanning trees, $\{S_1, S_2, \ldots, S_8\}$, with incidence vectors $e_{S_1} = (1, 1, 1, 0, 0)$, $e_{S_2} = (1, 1, 0, 0, 1)$, $e_{S_3} = (1, 0, 1, 1, 0)$, $e_{S_4} = (1, 0, 1, 0, 1)$, $e_{S_5} = (1, 0, 0, 1, 1)$, $e_{S_6} = (0, 1, 1, 1, 0)$, $e_{S_7} = (0, 1, 0, 1, 1)$, and $e_{S_8} = (0, 0, 1, 1, 1)$. If we let two edge weightings be $w_1 = (1, 3, 1, 2, 1)$ and $w_2 = (3, 1, 1, 1, 2)$, and let c_{S_i} be the cost vector for spanning tree S_i where the first entry is the cost of spanning tree S_i with respect to w_1 and the second entry is the cost of S_i with respect to w_2, then we have the following cost vectors: $c_{S_1} = (5, 5)$, $c_{S_2} = (5, 6)$, $c_{S_3} = (4, 5)$, $c_{S_4} = (3, 6)$, $c_{S_5} = (4, 6)$, $c_{S_6} = (6, 3)$, $c_{S_7} = (6, 4)$, and $c_{S_8} = (4, 4)$. Now we can plot these points in 2 dimensions, as shown in Figure 3.

Each point is marked with the spanning tree that it corresponds to. The Pareto optima are those where at least one of the coordinates is less than the corresponding coordinate for all other points. Thus from the graph we can see that the Pareto optima are S_4, S_6, and S_8.

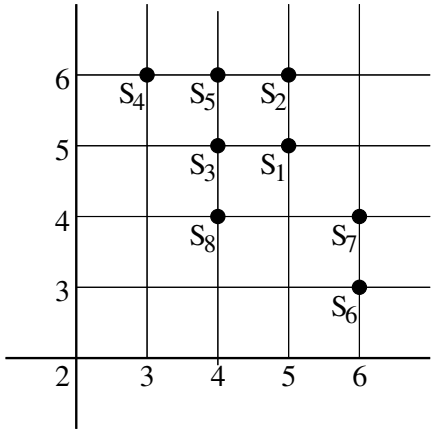


FIGURE 3. A simple example of Pareto Optima. The points marked with blue circles are the Pareto Optima.

We wanted to implement the Pareto optima optimization because it is a widely used method for finding the optimal spanning tree in regards to multiobjective optimization. The points that are Pareto optima points have the smallest cost in regards to at least one of the weights. Thus they are all optimal in some way. By finding the set of Pareto optima, we are limiting the spanning trees that need to be considered. This could be very helpful in practical applications. For example, suppose that we go back to our example discussed in Section 1.2. Instead of the government and environmental agency having to consider all possible spanning trees, we could find all the Pareto optima and we know that the one they choose will be among these spanning trees.

In our software package, I implemented a Pareto optimum code that when given a set of points, the Pareto optima are found. The code is in Appendix A.
Figure 4 gives an example of the output with our code.

The black dots are all of the projected points, and the points marked with blue circles signify the ones that are Pareto optima. Thus it can be seen that even in a simple case the number of Pareto optima is a very small percentage of the total number of projected points.

2.3. Enumerating All Spanning Trees of a Graph

One of the reasons that we developed our software package is because enumerating all the spanning trees of a graph is not efficient. We wanted to find a method that could enable us to perform multiobjective optimization without having to enumerate all spanning trees. But even though we did not want to enumerate all spanning trees in all cases, we needed

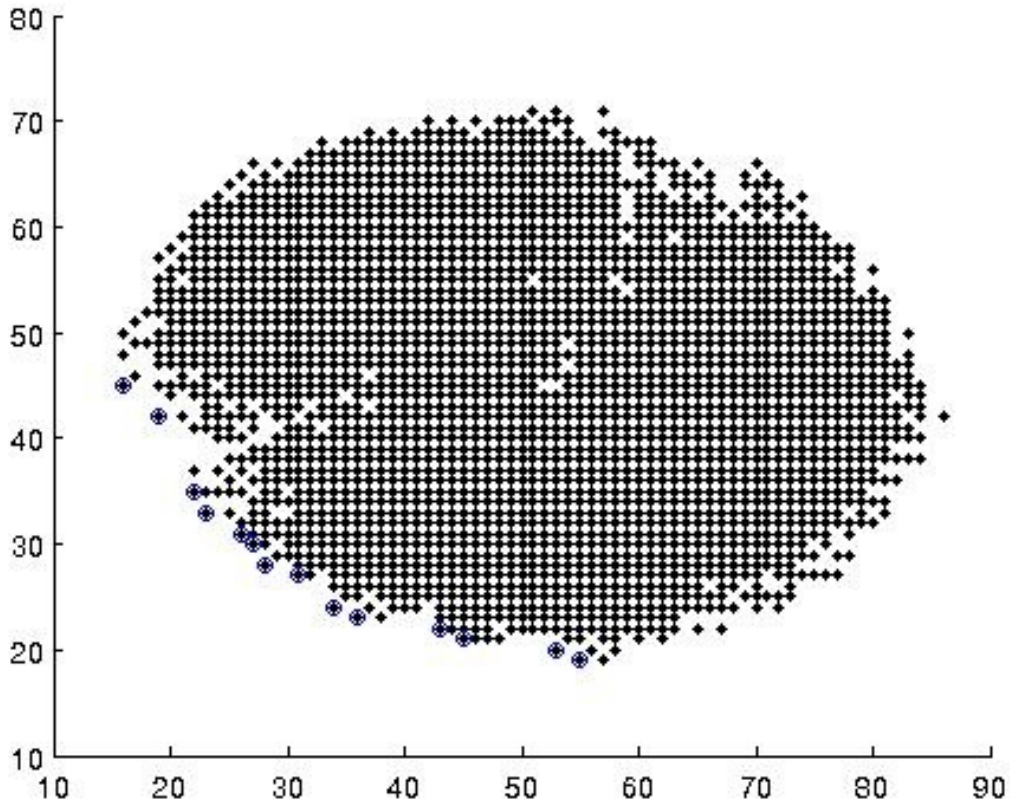


FIGURE 4. An Example of Pareto Optima.

some way to check our methods for correctness.

We ran tests of our heuristics on several different graphs, and in order to test for correctness, we compare the number of projected spanning trees found using our methods versus the real total number of projected spanning trees.

We used an algorithm for generating all of the spanning trees in undirected graphs presented by Matsui [**10**]. Matsui proves the following regarding the running time of his algorithm.

THEOREM 5. *The algorithm requires $O(n + m + \tau n)$ time when the given graph has n vertices, m edges, and τ spanning trees. For outputting all of the spanning trees explicitly, this time complexity is optimal.*

The algorithm generates all the spanning trees by traversing a tree structure on the graph of tree exchanges. The algorithm can also generate all of the spanning trees in nondecreasing order of weight when a weighted graph is given.

This algorithm requires an important pre-procedure, which we will explain in detail here. Before explaining the pre-procedure, we need to establish some notation. Assume we assign a linear ordering on the edge-set E by setting $E = \{e_1, e_2, ..., e_m\}$, where $|E| = m$. Then for any edge e_j, we say that the *index* of the edge e_j is j, and we denote the index of an edge e by $index(e)$. Ultimately, this pre-procedure partitions all of the edges into a specialized set of forests, which satisfies the following condition:

ASSUMPTION 1. *There exists a sequence of integers* $(j_0, j_1, \ldots, j_r)$ *satisfying that* $(1) 0 = j_0 < j_1 < \ldots < j_r = m$, *and* (2) *the edge partition* $(F_1, F_2, \ldots, F_r)$,
where $F_s = \{e_{j_{s-1}+1}, e_{j_{s-1}+2}, \ldots, e_{j_s}\}$, *satisfies the conditions that* F_1 *is a maximal forest of* G *and:*

$\forall s \in \{2, \ldots, r\}, F_s$ *is a maximal forest of the graph* $(V, E \setminus (F_1 \cup F_2 \cup \ldots \cup F_{s-1}))$.

This partitioning is done by employing the algorithm proposed by Nagamochi and Ibaraki [**11**] for generating a sparse k-connected graph. In the paper describing this algorithm, "A Linear-Time Algorithm for Finding a Sparse k-Connected Spanning Subgraph of a k-Connected Graph," Nagamochi and Ibaraki claim that by using their algorithm as preprocessing, the time complexity of algorithms for solving other graph problems can be improved, which is what Matsui does with his algorithm[**10**]. Based on this set of forests, Matsui's algorithm is able to decide the order of edges to exchange. In order to use our implementation of Matsui's algorithm, I implemented the partitioning algorithm[**11**]. This procedure requires the following property:

LEMMA 1. *For a graph* $G = (V, E)$, *simple or multiple, let* $F_i = (V, E_i)$ *be a maximal spanning forest in* $G - E_1 \cup E_2 \cup ... \cup E_{i-1}$, *for* $i = 1, 2, ..., |E|$, *where possibly* $E_i = E_{i+1} = ... = E_{|E|} = \emptyset$ *for some* i. *Then each spanning subgraph* $G_i = (V, E_1 \cup E_2 \cup ... \cup E_i)$ *satisfies*

$$\lambda(x, y; G_i) \geq min\{\lambda(x, y; G), i\} \text{ for all } x, y \in V,$$

where $\lambda(x, y; H)$ *denotes the local edge-connectivity between* x *and* y *in graph* H.

This lemma shows that $G_k = (V, E')$, where $E' = E_1 \cup E_2 \cup ... \cup E_k$, is k-edge- connected if $k \leq$ the edge-connectivity $\lambda(G)$. Also, G_k satisfies $|E'| \leq k(|V| - 1)$ since $|E_i| \leq |V| - 1$ for all i. Thus it is easy to see that G_k can be obtained in $O(k(|V| + |E|))$ time by repeating the graph search procedure k times. However, this time complexity can be reduced to $O(|V| + |E|)$ by constructing all $E_1, E_2, ..., E_{|E|}$ in a single scan. During the graph search

we compute, for each edge e being scanned, the i satisfying $e \in E_i$. Such i can be defined to be the smallest i such that $E_i \cup \{e\}$ does not contain a cycle, where these E_i denote the edge sets constructed so far. In general, checking whether $E_i \cup \{e\}$ contains a cycle requires $O(|E_i|)$ time, but to reduce this to $O(1)$, we always chose an unscanned edge e that is adjacent to an edge $e' \in E_i$ with the largest i. This graph search procedure is described as follows[**11**]:

Procedure FOREST; { input: $G(V, E)$, output: $E_1, E_2, ..., E_{|E|}$}
{ Let $r(v) := i$ denote that v has been reached by an edge of the forest
$F_i = (V, E_i)$.}
begin
1 $E_1 := E_2 := ... := E_{|E|} := \emptyset$
2 Label all nodes $v \in V$ and all edges $e \in E$ "unscanned";
3 $r(v) := 0$ for all $v \in V$;
4 **while** there exist "unscanned" nodes **do**
begin
5 Choose an "unscanned" node $x \in V$ with the largest r;
6 **for** each "unscanned" edge e incident to x **do**
begin
7 $E_{r(y)+1} := E_{r(y)+1} \cup \{e\}$ {y is the other end node ($\neq x$) of e}
8 **if** $r(x) = r(y)$ **then** $r(x) := r(x) + 1$;
9 $r(y) := r(y) + 1$;
10 Mark e "scanned"
end;
11 Mark x "scanned"
end;
end.

Figure 5 and Figure 6 show an example of the above procedure on a simple graph. Figure 5 shows the graph, and Figure 6 shows the partitioning of the edges. The edges in different partitions have different line types, thus there are four edge partitions, E_1, E_2, E_3 and E_4. In Figure 6 all of the edges and vertices are labeled, where x_i represents the ith node scanned by FOREST, and e_j represents the j^{th} edge scanned by FOREST. The edges are directed to show the path that FOREST takes.

In our implementation of this procedure, we need some way to find an unscanned node x with the largest $r(x)$ efficiently. To do this we prepare $|V|$ buckets such that each unscanned node v is contained in the $r(v)$th bucket. All of the nonempty buckets are doubly linked by pointers so that an unscanned node x with the largest $r(x)$ can be found in $O(1)$ time and the link update after increasing the label r of a node by one can also be done in $O(1)$ time. The entire time required to update bucket links is therefore $O(|V| + |E|)$

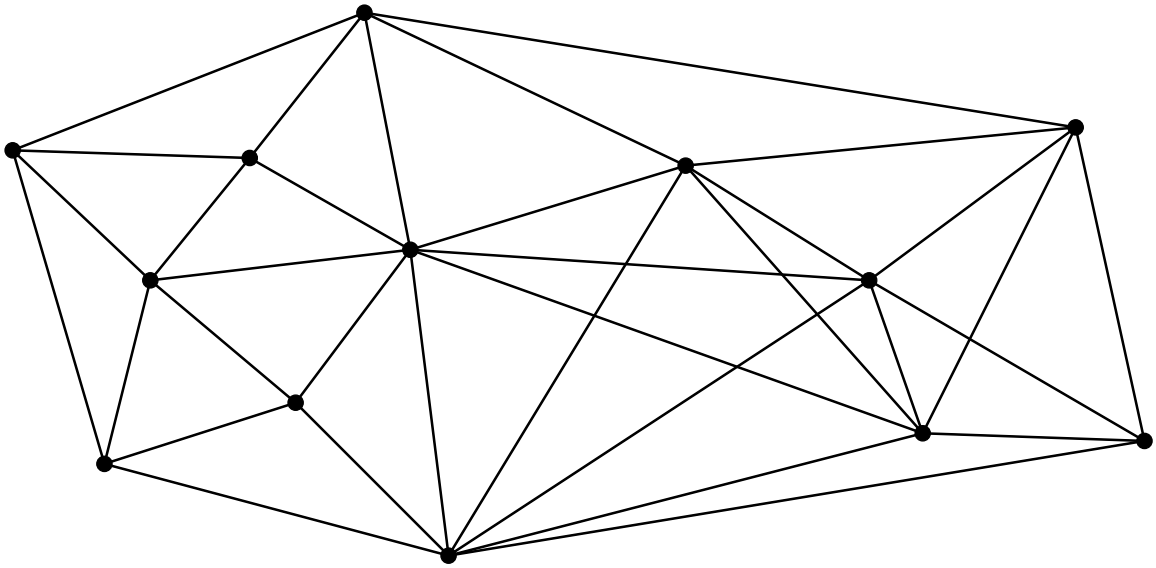


FIGURE 5. A simple graph.

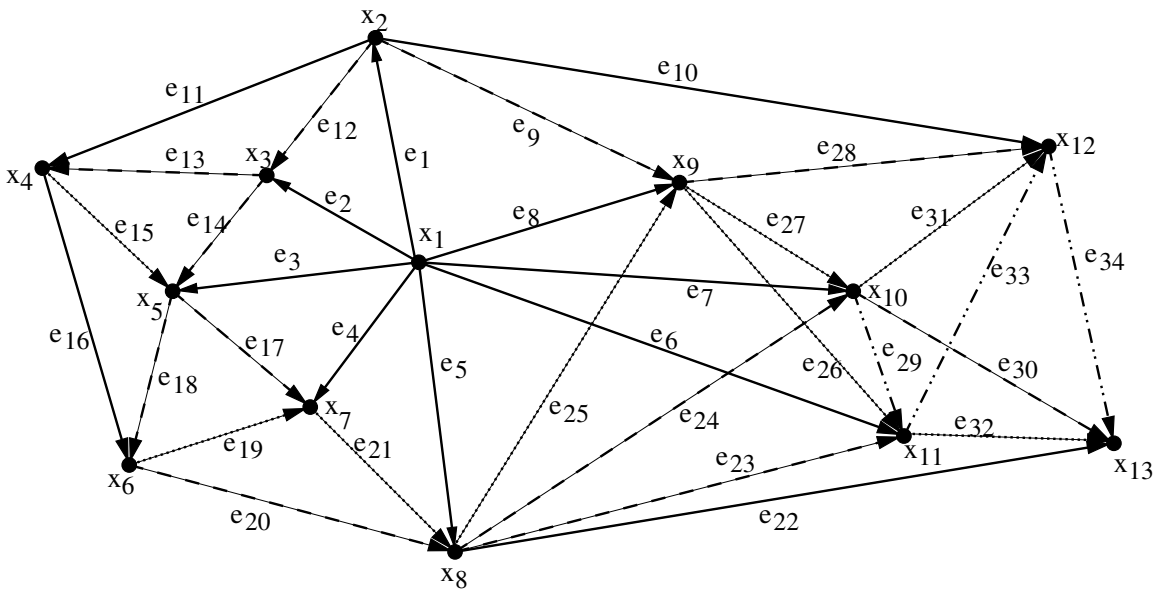


FIGURE 6. An example of the partitioning algorithm on the previous graph.

because labels are changed $O(|E|)$ times. This shows that the time complexity of FOREST is $O(|V| + |E|)$. The code for our implementation of the FOREST procedure can be found

in Appendix A.

As previously mentioned, with the set of forests produced by the procedure FOREST, Matsui's algorithm is able to efficiently decide the order of edges to exchange. Thus Matsui's algorithm can be seen as an application of the above partitioning technique of Nagamochi and Ibaraki. In Matsui's paper, he makes the assumption that the edges are partitioned in the way achieved by the FOREST procedure. With this assumption, he is able to prove a critical lemma, which is described below.

First we need to establish a necessary property achieved from the FOREST procedure. For any edge e in the edge set E that connects vertices u and v in the vertex set V, denote e by $\{u, v\}$. Additionally, as before we denote the ith edge partition by E_i. Then the FOREST procedure directly implies the following property:

Claim: For any edge $\{u, v\} \in E_i$, the vertices u and v are connected in the graph (V, E_{i-1}).

Given an edge-subset $E' \subseteq E$, the edge in E' with the smallest index is called the *top edge* of E'. Similarly, we call the edge in E' with the largest index the *bottom edge* of E'. Also, for a spanning tree T, we let $\phi(T)$ denote the spanning tree $(T \setminus \{f\}) \cup \{g\}$ where f is the bottom edge of T and g is the top edge of the cut-set of T when f is deleted. Then for any spanning tree T, we say that T is a *child* of $\phi(T)$ and $\phi(T)$ is the *parent* of T. Furthermore, we say that an edge f is a *pivot edge* of T if there exists a child T' of T such that $T' \setminus T = \{f\}$. Then from the claim above, Matsui proves the following Lemma:

LEMMA 2. *Let T be a spanning tree of the graph G and let k be the index of the bottom edge of T. For any pivot edge f of T, either $index(f) \leq k + 2n - 3$ or there exists a pivot edge f' satisfying the condition that $index(f) - 2n + 3 \leq index(f') < index(f)$.*

This lemma gives an algorithm for finding all of the pivot edges efficiently. Then with an algorithm for finding all of the children of a spanning tree (described in the paper by Matsui), the algorithm for enumerating all of the spanning trees is able to be constructed.

This implementation was necessary because our first goal was to perform experiments on matroids for which we can compute all bases in order to better understand our heuristics and algorithms. We generated fifteen connected random graphs: *gn9e18, gn9e27, gn10e22, gn10e28, gn10e33, gn11e13, gn11e20, gn11e27, gn11e41, gn12e16, gn12e24, gn12e33, gn13e19, gn13e29, gn13e39* which we will refer to as our *calibration set*. The names of our graphs follow the simple nomenclature *gn*[#**nodes**]*e*[#**edges**]. We consider two, three and five criteria, i.e. number of weightings. We further consider three different ranges of integral weights for each criteria. For the calibration set we adopt the following nomenclature

$$gn[\#\textbf{nodes}]e[\#\textbf{edges}]d[\#\textbf{criteria}]w[\textbf{low weight}]w[\textbf{high weight}]$$

where we generated random integral weightings between [**low weight**] and [**high weight**]. First we simply compare the number of spanning trees of our calibration set to the number of projected spanning trees.

- **Table 1** shows the calibration set with two weightings (criteria) and integral weights $0-20$, $0-100$, and $0-1000$.
- **Table 2** shows the calibration set with three weightings (criteria) and integral weights $0-20$, $0-100$, and $0-1000$.
- **Table 3** contains the calibration set with five weightings (criteria) and integral weights $0-1$, $0-2$, and $0-5$.

We give the exact number of projected spanning trees and compare versus the exact number of spanning trees.

TABLE 1. Calibration test set: Exact #Spanning trees vs. #projected spanning trees in 2 criteria.

Name	Nodes	Edges	Weight range	#Spanning Trees	#Projected Trees	Percent
gn9e18d2w0w20	9	18	$0-20$	$2,981$	$1,742$	58.44
gn9e27d2w0w20	9	27	$0-20$	$372,320$	$6,346$	01.70
gn10e22d2w0w20	10	22	$0-20$	$53,357$	$3,957$	07.42
gn10e28d2w0w20	10	28	$0-20$	$800,948$	$7,131$	00.89
gn10e33d2w0w20	10	33	$0-20$	$3,584,016$	$10,833$	00.30
gn11e13d2w0w20	11	13	$0-20$	41	41	100.00
gn11e20d2w0w20	11	20	$0-20$	$6,939$	$2,173$	31.32
gn11e27d2w0w20	11	27	$0-20$	$284,730$	$7,515$	02.64
gn11e41d2w0w20	11	41	$0-20$	$90,922,271$	$16,457$	00.02
gn12e16d2w0w20	12	16	$0-20$	162	154	95.06
gn12e24d2w0w20	12	24	$0-20$	$208,380$	$5,647$	02.71
gn12e33d2w0w20	12	33	$0-20$	$4,741,624$	$9,364$	00.20
gn13e19d2w0w20	13	19	$0-20$	$3,401$	$1,608$	47.28
gn13e29d2w0w20	13	29	$0-20$	$1,543,340$	$8,474$	00.55
gn13e39d2w0w20	13	39	$0-20$	$131,807,934$	$18,468$	00.01
gn9e18d2w0w100	9	18	$0-100$	$2,981$	$2,858$	95.87
gn9e27d2w0w100	9	27	$0-100$	$372,320$	$89,092$	23.93
gn10e22d2w0w100	10	22	$0-100$	$53,357$	$37,204$	69.73
gn10e28d2w0w100	10	28	$0-100$	$800,948$	$101,334$	12.65
gn10e33d2w0w100	10	33	$0-100$	$3,584,016$	$166,427$	04.64
gn11e13d2w0w100	11	13	$0-100$	41	41	100.00
gn11e20d2w0w100	11	20	$0-100$	$6,939$	$6,580$	94.83
gn11e27d2w0w100	11	27	$0-100$	$284,730$	$81,803$	28.73
gn11e41d2w0w100	11	41	$0-100$	$90,922,271$	$309,961$	00.34
gn12e16d2w0w100	12	16	$0-100$	162	162	100.00
gn12e24d2w0w100	12	24	$0-100$	$208,380$	$92,813$	44.54
gn12e33d2w0w100	12	33	$0-100$	$4,741,624$	$192,122$	04.05
gn13e19d2w0w100	13	19	$0-100$	$3,401$	$3,255$	95.71
gn13e29d2w0w100	13	29	$0-100$	$1,543,340$	$164,617$	10.67
gn13e39d2w0w100	13	39	$0-100$	$131,807,934$	$315,881$	00.24
gn9e18d2w0w1000	9	18	$0-1000$	$2,981$	$2,981$	100.00
gn9e27d2w0w1000	9	27	$0-1000$	$372,320$	$364,382$	97.87
gn10e22d2w0w1000	10	22	$0-1000$	$53,357$	$52,990$	99.31
gn10e28d2w0w1000	10	28	$0-1000$	$800,948$	$756,013$	94.39
gn10e33d2w0w1000	10	33	$0-1000$	$3,584,016$	$2,726,287$	76.07
gn11e13d2w0w1000	11	13	$0-1000$	41	41	100.00
gn11e20d2w0w1000	11	20	$0-1000$	$6,939$	$6,921$	99.74
gn11e27d2w0w1000	11	27	$0-1000$	$284,730$	$279,308$	98.10
gn11e41d2w0w1000	11	41	$0-1000$	$90,922,271$	$13,884,793$	15.27
gn12e16d2w0w1000	12	16	$0-1000$	162	162	100.00
gn12e24d2w0w1000	12	24	$0-1000$	$208,380$	$205,690$	98.71
gn12e33d2w0w1000	12	33	$0-1000$	$4,741,624$	$3,680,313$	77.62
gn13e19d2w0w1000	13	19	$0-1000$	$3,401$	$3,401$	100.00
gn13e29d2w0w1000	13	29	$0-1000$	$1,543,340$	$1,396,180$	90.46
gn13e39d2w0w1000	13	39	$0-1000$	$131,807,934$	$15,037,589$	11.41

TABLE 2. Calibration test set: Exact #Spanning trees vs. #projected spanning trees in 3 criteria.

Name	Nodes	Edges	Weight range	#Spanning Trees	#Projected Trees	Percent
gn9e18d3w0w20	9	18	$0-20$	$2,981$	$2,905$	97.45
gn9e27d3w0w20	9	27	$0-20$	$372,320$	$134,039$	36.00
gn10e22d3w0w20	10	22	$0-20$	$53,357$	$42,887$	80.38
gn10e28d3w0w20	10	28	$0-20$	$800,948$	$238,529$	29.78
gn10e33d3w0w20	10	33	$0-20$	$3,584,016$	$411,730$	11.49
gn11e13d3w0w20	11	13	$0-20$	41	41	100.00
gn11e20d3w0w20	11	20	$0-20$	$6,939$	$6,603$	95.16
gn11e27d3w0w20	11	27	$0-20$	$284,730$	$146,672$	51.51
gn11e41d3w0w20	11	41	$0-20$	$90,922,271$	$959,469$	01.06
gn12e16d3w0w20	12	16	$0-20$	162	162	100.00
gn12e24d3w0w20	12	24	$0-20$	$208,380$	$111,201$	53.36
gn12e33d3w0w20	12	33	$0-20$	$4,741,624$	$470,609$	09.93
gn13e19d3w0w20	13	19	$0-20$	$3,401$	$3,358$	98.74
gn13e29d3w0w20	13	29	$0-20$	$1,543,340$	$264,949$	17.18
gn13e39d3w0w20	13	39	$0-20$	$131,807,934$	$930,322$	00.71
gn9e18d3w0w100	9	18	$0-100$	$2,981$	$2,981$	100.00
gn9e27d3w0w100	9	27	$0-100$	$372,320$	$367,313$	98.66
gn10e22d3w0w100	10	22	$0-100$	$53,357$	$53,289$	99.87
gn10e28d3w0w100	10	28	$0-100$	$800,948$	$786,781$	98.23
gn10e33d3w0w100	10	33	$0-100$	$3,584,016$	$3,351,096$	93.01
gn11e13d3w0w100	11	13	$0-100$	41	41	100.00
gn11e20d3w0w100	11	20	$0-100$	$6,939$	$6,939$	100.00
gn11e27d3w0w100	11	27	$0-100$	$284,730$	$281,303$	98.80
gn11e41d3w0w100	11	41	$0-100$	$90,922,271$	$35,943,327$	39.53
gn12e16d3w0w100	12	16	$0-100$	162	162	100.00
gn12e24d3w0w100	12	24	$0-100$	$208,380$	$207,143$	99.41
gn12e33d3w0w100	12	33	$0-100$	$4,741,624$	$4,740,880$	99.98
gn13e19d3w0w100	13	19	$0-100$	$3,401$	$3,401$	100.00
gn13e29d3w0w100	13	29	$0-100$	$1,543,340$	$1,484,719$	96.20
gn13e39d3w0w100	13	39	$0-100$	$131,807,934$	$44,757,592$	33.96
gn9e18d3w0w1000	9	18	$0-1000$	$2,981$	$2,981$	100.00
gn9e27d3w0w1000	9	27	$0-1000$	$372,320$	$372,320$	100.00
gn10e22d3w0w1000	10	22	$0-1000$	$53,357$	$53,357$	100.00
gn10e28d3w0w1000	10	28	$0-1000$	$800,948$	$800,946$	99.99
gn10e33d3w0w1000	10	33	$0-1000$	$3,584,016$	$3,583,757$	99.99
gn11e13d3w0w1000	11	13	$0-1000$	41	41	100.00
gn11e20d3w0w1000	11	20	$0-1000$	$6,939$	$6,939$	100.00
gn11e27d3w0w1000	11	27	$0-1000$	$284,730$	$284,730$	100.00
gn11e41d3w0w1000	11	41	$0-1000$	$90,922,271$	$90,699,181$	99.75
gn12e16d3w0w1000	12	16	$0-1000$	162	162	100.00
gn12e24d3w0w1000	12	24	$0-1000$	$208,380$	$208,356$	99.99
gn12e33d3w0w1000	12	33	$0-1000$	$4,741,624$	$4,740,880$	99.98
gn13e19d3w0w1000	13	19	$0-1000$	$3,401$	$3,401$	100.00
gn13e29d3w0w1000	13	29	$0-1000$	$1,543,340$	$1,543,304$	99.99
gn13e39d3w0w1000	13	39	$0-1000$	$131,807,934$	$131,464,478$	99.74

TABLE 3. Calibration test set: Exact #Spanning trees vs. #projected spanning trees in 5 criteria.

Name	Nodes	Edges	Weight range	#Spanning Trees	#Projected Trees	Percent
gn9e18d5w0w1	9	18	$0-1$	2,981	1,099	36.87
gn9e27d5w0w1	9	27	$0-1$	372,320	8,205	2.20
gn10e22d5w0w1	10	22	$0-1$	53,357	4,746	8.89
gn10e28d5w0w1	10	28	$0-1$	800,948	10,898	1.36
gn10e33d5w0w1	10	33	$0-1$	3,584,016	15,482	0.43
gn11e13d5w0w1	11	13	$0-1$	41	27	65.85
gn11e20d5w0w1	11	20	$0-1$	6,939	1,866	26.89
gn11e27d5w0w1	11	27	$0-1$	284,730	5,828	2.05
gn11e41d5w0w1	11	41	$0-1$	90,922,271	36,847	0.04
gn12e16d5w0w1	12	16	$0-1$	162	116	71.61
gn12e24d5w0w1	12	24	$0-1$	208,380	6,738	3.23
gn12e33d5w0w1	12	33	$0-1$	4,741,624	20,857	0.44
gn13e19d5w0w1	13	19	$0-1$	3,401	1,149	33.78
gn13e29d5w0w1	13	29	$0-1$	1,543,340	111,116	7.20
gn13e39d5w0w1	13	39	$0-1$	131,807,934	34,392	0.03
gn9e18d5w0w2	9	18	$0-2$	2,981	2,276	76.35
gn9e27d5w0w2	9	27	$0-2$	372,320	43,029	11.56
gn10e22d5w0w2	10	22	$0-2$	53,357	14,623	27.41
gn10e28d5w0w2	10	28	$0-2$	800,948	66,190	8.26
gn10e33d5w0w2	10	33	$0-2$	3,584,016	105,309	2.94
gn11e13d5w0w2	11	13	$0-2$	41	41	100.00
gn11e20d5w0w2	11	20	$0-2$	6,939	3,761	54.20
gn11e27d5w0w2	11	27	$0-2$	284,730	39,292	13.80
gn11e41d5w0w2	11	41	$0-2$	90,922,271	290,555	0.32
gn12e16d5w0w2	12	16	$0-2$	162	160	98.77
gn12e24d5w0w2	12	24	$0-2$	208,380	34,057	16.34
gn12e33d5w0w2	12	33	$0-2$	4,741,624	120,211	2.54
gn13e19d5w0w2	13	19	$0-2$	3,401	2,613	76.83
gn13e29d5w0w2	13	29	$0-2$	1,543,340	98,285	6.37
gn13e39d5w0w2	13	39	$0-2$	131,807,934	348,703	0.26
gn9e18d5w0w5	9	18	$0-5$	2,981	2,960	99.30
gn9e27d5w0w5	9	27	$0-5$	372,320	271,048	72.80
gn10e22d5w0w5	10	22	$0-5$	53,357	49,463	92.70
gn10e28d5w0w5	10	28	$0-5$	800,948	493,565	61.62
gn10e33d5w0w5	10	33	$0-5$	3,584,016	1,294,875	36.13
gn11e13d5w0w5	11	13	$0-5$	41	41	100.00
gn11e20d5w0w5	11	20	$0-5$	6,939	5,975	86.11
gn11e27d5w0w5	11	27	$0-5$	284,730	222,974	78.31
gn11e41d5w0w5	11	41	$0-5$	90,922,271	4,711,354	5.18
gn12e16d5w0w5	12	16	$0-5$	162	162	100.00
gn12e24d5w0w5	12	24	$0-5$	208,380	177,845	85.35
gn12e33d5w0w5	12	33	$0-5$	4,741,624	1,489,751	31.42
gn13e19d5w0w5	13	19	$0-5$	3,401	3,381	99.41
gn13e29d5w0w5	13	29	$0-5$	1,543,340	787,196	51.01
gn13e39d5w0w5	13	39	$0-5$	131,807,934	7,737,684	5.87

2.4. Estimation of Bases

In this section we will give a short explanation of matroids, and then introduce a method for enumerating all bases of a matroid. We begin with one of the many equivalent definitions of a matroid.

DEFINITION 7. A non-empty collection B of subsets of $[n] := 1, ..., n$ is the set of bases of a *matroid* M if and only if

- If $B_1, B_2 \in B$ and $x \in B_1 \setminus B_2, \exists y \in B_2 \setminus B_1$ such that $(B_1 \cup y) \setminus x \in B$.

From this definition we can conclude that in a graphical matroid, which are the ones that we have been looking at throughout this senior thesis, the set $[n]$ is the set of all edges of the graph, and a subset of $[n]$ is in B if it is a spanning tree.

To enumerate all bases of a matroid, we implemented the asymptotic $0\backslash1$ polytope vertex-estimation presented in [**3**]. This gives us the ability to estimate the number of bases of matroid polytopes in order to better understand the ratio of bases to projected bases for problems where full enumeration is intractable. In this section, I will introduce the ideas presented by Barvinok and Samorodnitsky in [**2**] and [**3**].

In [**2**] Barvinok and Samorodnitsky develop general methods to obtain fast (polynomial time) estimates of the cardinality of a combinatorially defined set via solving some randomly generated optimization problems on the set. A general problem of combinatorial counting can be stated as follows: given a family $F \subset 2^X$ of subsets of the ground set X, compute or estimate the cardinality $|F|$ of the family. To clarify what "given" means, since in most interesting cases $|F|$ is exponentially large in the cardinality of $|X|$, we assume that the family is defined by its *Optimization Oracle*:

Optimization Oracle defining a family $F \subset 2^X$
Input: A set of integer weights $\gamma_x : x \in X$.
Output: The number $min \sum_{x \in Y} \gamma_x$ over all $Y \in F$.

That is, for any given integer weighting $\{\gamma_x\}$ on the set X, we should be able to produce the minimum weight of a subset $Y \in F$. The main motivation of Barvinok and Samorodnitsky was the example of finding perfect matchings in a graph, and they give several other examples. Here, I will discuss the example of bases in matroids given in [**3**], since this is the problem we are concerned with.

EXAMPLE 9. *(Bases in Matroids)* Let A be a $k \times n$ matrix of rank k over a field $\mathbf{F}$. We assume that $k < n$. Let $X = X(A)$ be the set of all k-subsets x of $\{1, ..., n\}$ such that the columns of A indexed by the elements of x are linearly independent. Thus X is the set of all

non-zero $k \times k$ minors of A, or, in other words, the set of bases of the matroid represented by A. According to Barvinok and Samorodnitsky it is an interesting and apparently hard problem to compute or to approximate the cardinality of X. On the contrary, it is easy to construct the Optimization Oracle for X. Given real weights $\gamma_1, ..., \gamma_n$, we construct a linearly independent set $a_{i_1}, ..., a_{i_k}$ of columns of the largest total weight one-by-one: first we choose a_{i_1} to be a non-zero column of A with the largest possible weight γ_{i_1}, then we choose a_{i_2} to be a column of the maximum possible weight such that a_{i_1} and a_{i_2} are linearly independent, etc. Particular cases of this problem include counting forests and spanning subgraphs in a given graph. It should also be noted that this is a generalization of Kruskal's algorithm.

The algorithms given by Barvinok and Samorodnitsky are randomized, that is the outcome is a random variable, which, with high probability satisfies the desired properties. They are able to make the probability of success arbitrarily close to 1 by running the algorithm several times and averaging the outcomes.

In [**3**], they start by fixing a Borel probability measure μ in $\mathbf{R}$. They require μ to be symmetric, that is, $\mu(A) = \mu(-A)$ for any Borel set $A \subset \mathbf{R}$, and to have finite variance. They relate two quantities associated with X: the cardinality, $|X|$, of X, and $\Gamma(X, \mu)$, which is defined as follows. Let us fix a measure μ as above and let $\gamma_1, ..., \gamma_n$ be independent random variables having the distribution μ. Then

$$\Gamma(X, \mu) = \mathbf{E} \max \textstyle\sum_{i \in x} \gamma_i \text{ over all } x \in X.$$

In other words, we sample weights of $1, ..., n$ independently at random from the distribution μ, define the weight of a subset $x \in X$ as the sum of the weights of its elements and let $\Gamma(X, \mu)$ be the expected maximum weight of a subset from X. In a sense, $\Gamma(X, \mu)$ measures how large X is, and in some respects, $\Gamma(X, \mu)$ behaves rather like $ln|X|$. Then the goal of [**3**] is stated as follows:

Goal: Find a measure μ for which $\Gamma(X, \mu)$ gives the best estimate of $ln|X|$.

The value of $\Gamma(X, \mu)$ can be efficiently computed through the averaging of several sample maxima for randomly chosen weights $\gamma_1, ..., \gamma_n$. Counting the elements in X can be a hard and interesting problem, thus $\Gamma(X, \mu)$ provides a quick estimate for $ln|X|$. One of the main results of [**3**] is that there are measures μ for which $\Gamma(X, \mu)$ gives an asymptotically tight estimate for $ln|X|$ provided $ln|X|$ grows faster than a linear function of k, where k is the number of elements in the subsets that X is composed of. The best estimates are obtained when μ is the logistic distribution. The approach to the combinatorial counting problem by Barvinok and Samorodnitsky provides very crude bounds, but it is insensitive

to the varied structure of the family of subsets, so it is ready to handle a broad class of problems.

To implement this code, I used the function GreedyAlgorithmMax() in MOCHA [**5**] and I also used the logarithmic distribution functions created by Alexander Yong on the webpage "C++ codes for estimating permanents, hafnians and the number of forests in a graph [**13**]." The code used in this implementation is included in Appendix A.

APPENDIX A

Selected Pieces of Our Code in the Package MOCHA

Pareto Optimum

The following code is a simple program to find the Pareto optima when given a set of points inputPoints in $\mathbf{R}^n$. This code can be found in the file mathprog.cpp in the software package MOCHA [**5**].

```
set <Matrix, ltcolvec> ProjBalMatroidOpt::ParetoOptimum( set <Matrix, ltcolvec> &inputPoints)
{
    // This is the set of pareto optimum we will calculate.
    // This is a set structure from the standard library which
    // holds Matrix classes that we defined. ltcolvec is a bool
    // function that compares two Matrix classes with one column.

    set <Matrix, ltcolvec> popt = inputPoints;

    // Perform some error checking
    if (inputPoints.empty())
    {
        return popt;
    }

    // An iterator to go through all inputPoints.
    set <Matrix, ltcolvec>::iterator msetit = popt.begin();

    //An iterator to go in front of points so when we erase we can get back to where we
    //started
    set <Matrix, ltcolvec>::iterator msetit2 = popt.begin();
    msetit2++;

    //An iterator to compare to
    set <Matrix, ltcolvec>::iterator stepit = popt.begin();
```

```
    int counter = 0;

    for( ; msetit != popt.end(); msetit++)
    {

        // Set this temp matrix to the current matrix msetit is refering to
        Matrix tempM = *msetit;
        for(stepit = popt.begin(); stepit != popt.end(); stepit++)
        {
            if (stepit != msetit)
            {
                counter = 0;
                Matrix tempM2 = *stepit;
                for(int i = 0; i < tempM.rows; i++)
                {
                    //if all coordinates are larger than another point
                    if(tempM(i,0) ≤ tempM2(i,0))
                        counter++;
                }
                if(counter == tempM.rows)
                {
                    popt.erase(msetit);
                    msetit = msetit2;
                    msetit−−;
                    break;
                }
            }
        }
        msetit2++;
    }

    return popt;
}
```

FOREST Implementation

The following code implements the algorithm presented by Nagamochi and Ibaraki in [**11**]. This code can be found in the file graph.cpp in the software package MOCHA [**5**].

```
list <set <unsigned>> Graph::NagIbar ()
{

    //declare list of edge partitions to be returned
    set <unsigned> emptySet;
    list <set <unsigned>> edgePartitions(numEdges, emptySet);

    //matrix to know whether edges are scanned or not
    int Unscanned[numNodes][numNodes];

    //initialize the matrix to say none of the edges are scanned
    int i, j;
    for(i = 0; i < numNodes; i++)
    for(j = 0; j < numNodes; j++)
    {
        if(j == i)
            Unscanned[i][j] = 0;
        else
            Unscanned[i][j] = adjMatrix(i,j);
    }

    //make buckets and put all vertex numbers in 0th bucket
    VertexBucket allVertices(0);
    for(i = 0; i < numNodes; i++)
        allVertices.vertices.push_front(i);
    list <VertexBucket> Buckets(1, allVertices);

    //array to hold r-values for each vertex
    int rvalues[numNodes];

    //initialize each r-value to 0
    for(i = 0; i < numNodes; i++)
        rvalues[i] = 0;

    int v, r, node;
    list <set <unsigned>>::iterator lsit;
```

```
lsit = edgePartitions.begin();
list <VertexBucket>::iterator lsit2;
lsit2 = Buckets.begin();
list <VertexBucket>::iterator lsit3;
lsit3 = Buckets.begin();
set <unsigned> itrSet;

//as long as we still have a vertex in a bucket
while(!Buckets.empty())
{
    lsit2 = Buckets.begin();
    v = (*lsit2).vertices.front();
    (*lsit2).vertices.pop_front();
    if((*lsit2).vertices.empty())
        Buckets.erase(lsit2);
    for(node = 0; node < numNodes; node++)
    {
        if(Unscanned[v][node] == 1)
        {
            r = rvalues[node] + 1;
            lsit = edgePartitions.begin();
            advance(lsit, r-1);
            (*lsit).insert(edgeNumber(v,node));
            if(rvalues[v] == rvalues[node])
                rvalues[v] += 1;
            rvalues[node] += 1;
            //move vertex node to new bucket
            lsit3 = Buckets.begin();
            for( ; (*lsit3).r != rvalues[node] && lsit3 != Buckets.end() ; lsit3++);
                if((*lsit3).r == rvalues[node] && lsit3 != Buckets.end())
                {
                    (*lsit3).vertices.push_front(node);
                    ++lsit3;
                    (*lsit3).vertices.remove(node);
                    if((*lsit3).vertices.empty())
                        Buckets.erase(lsit3);
                }
                else
                {
                    lsit3 = Buckets.begin();
                    for( ; (*lsit3).r > rvalues[node]; lsit3++);
```

```
                    VertexBucket newVertexBucket(rvalues[node]);
                newVertexBucket.vertices.push_front(node);
                Buckets.insert(lsit3, newVertexBucket);
                (*lsit3).vertices.remove(node);
                if((*lsit3).vertices.empty())
                    Buckets.erase(lsit3);
            }
            Unscanned[v][node] = 0;
            Unscanned[node][v] = 0;
        }
    }
}
return edgePartitions;
}
```

Barvinok's Estimation

The following code implements the algorithm for estimating the number of bases for matriods in [**3**]. This code can be found in the file estimatebases.cpp in the software package MOCHA [**5**].

```
int main (int argc, char *argv[])
{
    int m, num_rows, num_cols, i, j, k, max_weight;
    float upper_bound, lower_bound, GAMMA, avg_GAMMA;
    string filename;
    string matroidType;
    ifstream inFile;
    set <unsigned> max_weight_basis;

    cout << "argc = " << argc << endl;

    cout << "This program estimates the number of bases of a matrix by assigning\n";
    cout << "random weights to each column of the matrix and finding the maximal\n";
    cout << "weight basis. This is done m times and the resulting average estimates\n";
    cout << "the function GAMMA. Then upper and lower bounds for the number of
    bases\n";
    cout << "are calculated.\n\n";
    if (argc > 1)
    {
        sscanf(argv[1],"%d",&m);
        cout << "m = " << m << endl;
    }
    else
    {
        cout << "Enter the number of times m to sample: \n";
        cin >> m;
        cout << "\n";
    }
    if (m <= 0)
    {
        cout << "m <= 0 exiting" << endl;
        exit(0);
    }
```

```
Matroid *M;

if (argc > 2)
{
    filename = argv[2];
}
else
{
    cout << "Enter the name of the input file:\n";
    cin >> filename;
}

inFile.open(filename.c_str(), ios::in);

while(!inFile)
{
    cout << "File open error: '" << filename << "' ";
    cout << "Try again.\n";
    inFile.close();
    cout << "Enter the name of the input file:\n";
    cin >> filename;
    inFile.open(filename.c_str(), ios::in);
}
if (argc > 3)
{
    matroidType = argv[3];
}
else
{
    cout << "Matroid type (1) Vector, (2) Graphical: ";
    cin >> matroidType;
}
if (matroidType == "1")
{
    M = new VectorMatroid(inFile);
}
else if (matroidType == "2")
{
    M = new GraphicalMatroid(inFile);
}
cout << *M;
```

```
num_rows = M->getNumElements();
Matrix weights(num_rows,1);

GAMMA=0;
srand((unsigned)time(NULL));

int ten_percent = floor(m*0.10);
ten_percent = (int)max((double)1,(double)ten_percent);
cout << "ten_percent = " << ten_percent << endl;
for(k=0; k<m; k++)
{
    if (k % ten_percent == 0 && k != 0)
    {
        cout << " " << (k/ten_percent)*10 << "%" << endl;
    }
    max_weight_basis.clear();

    for(i = 0; i < num_rows; i++){
        weights(i,0)=0;
    }

    for(i = 0; i < num_rows; i++){
        weights(i,0)=M->random_weight_logistic((float) ((rand() % 99999) + 1) / 100000);
    }

    max_weight_basis = M- >GreedyAlgorithmMax(weights);

    set<unsigned>::iterator it;
    max_weight = 0;

    for(it = max_weight_basis.begin(); it != max_weight_basis.end(); it++)
    {
        max_weight += weights(*it,0);
    }
    GAMMA=GAMMA+max_weight;
}
cout << " 100%" << endl;

//compute upper and lower estimates
avg_GAMMA = (float)GAMMA/m;
upper_bound = M->upper_logistic(avg_GAMMA, num_rows);
```

```
    lower_bound = M->lower_logistic(avg_GAMMA, num_rows);

    //Output results
    cout << "\n";
    cout << "\n";
    cout << "Bounds for Log(X) are:\n";
    cout << "Upper bound:\n";
    printf("%f\n",upper_bound);
    cout << "Lower bound:\n";
    printf("%f\n",lower_bound);
    cout << "GAMMA(X):\n";
    printf("%f\n",avg_GAMMA);

}
```

Bibliography

1. Martin Aigner and Günter M. Ziegler, *Proofs from The Book*, second ed., Springer-Verlag, Berlin, 2001, Including illustrations by Karl H. Hofmann. MR MR1801937 (2001j:00001)
2. Alexander Barvinok and Alex Samorodnitsky, *The distance approach to approximate combinatorial counting*, Geom. Funct. Anal. **11** (2001), no. 5, 871–899. MR MR1872641 (2002i:60015)
3. ———, *Random weighting, asymptotic counting, and inverse isoperimetry*, Israel J. Math. **158** (2007), 159–191. MR MR2342462 (2008j:60022)
4. William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver, *Combinatorial optimization*, Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons Inc., New York, 1998, A Wiley-Interscience Publication. MR MR1490579 (99b:90098)
5. Jesús De Loera, David C. Haws, and Allison O'Hair, *Matroids Optimization Combinatorics Heuristics and Algorithms*, Available from URL `http://math.ucdavis.edu/~haws/MOCHA/`, 2009.
6. Matthias Ehrgott, *On matroids with multiple objectives*, Optimization **38** (1996), no. 1, 73–84, Multicriteria optimization and decision theory (Holzhau, 1994). MR MR1411486 (97h:90044)
7. Takahiko Kamae, *The existence of a Hamilton circuit in a tree graph*, IEEE Trans. Circuit Theory **CT-14** (1967), 279–283. MR MR0246789 (40 #58)
8. Jesús De Loera, David Haws, Jon Lee, and Allison O'Hair, *Computation in multicriteria matroid optimization*, Not yet published, 2009.
9. László Lovász, József Pelikán, and Katalin Vesztergombi, *Discrete mathematics*, Undergraduate Texts in Mathematics, Springer-Verlag, New York, 2003, Elementary and beyond. MR MR1952453 (2004a:05001)
10. Tomomi Matsui, *A flexible algorithm for generating all the spanning trees in undirected graphs*, Algorithmica **18** (1997), no. 4, 530–543. MR MR1453415 (98a:68149)
11. Hiroshi Nagamochi and Toshihide Ibaraki, *A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph*, Algorithmica **7** (1992), no. 5-6, 583–596. MR MR1154589 (93d:05096)
12. Douglas B. West, *Introduction to graph theory*, Prentice Hall Inc., Upper Saddle River, NJ, 1996. MR MR1367739 (96i:05001)
13. Alexander Yong, *C++ codes for estimating permanents, hafnians, and the number of forests in a graph*, Available from URL `http://www.math.lsa.umich.edu/~barvinok/manual.html`, 2007.