PHYLOGENETIC INFERENCE USING A DISCRETE-INTEGER LINEAR PROGRAMMING MODEL

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

William Alvah Sands

May, 2017

PHYLOGENETIC INFERENCE USING A DISCRETE-INTEGER LINEAR PROGRAMMING MODEL

William Alvah Sands

Thesis

Approved:	Accepted:
Advisor Dr. Stefan Forcey	Dean of the College Dr. John Green
Faculty Reader Dr. Malena Español	Dean of the Graduate School Dr. Chand Midha
Faculty Reader Dr. J. Patrick Wilber	Date
Department Chair Dr. Timothy Norfolk	

ABSTRACT

Combinatorial methods have proved to be useful in generating relaxations of polytopes in various areas of mathematical programming. In this work, we propose a discrete-integer linear programming model for a recent version of the Phylogeny Estimation Problem (PEP), known as the Balanced Minimal Evolution Method (BME). We begin by examining an object known as the Balanced Minimal Evolution Polytope and several classes of geometric constraints that result in its relaxation. We use this information to develop the linear program and propose two Branch and Bound algorithms to solve the model. The second algorithm takes advantage of a heuristic known as a large neighborhood search. We provide experimental results for both algorithms, using perfect and noisy data, as well as suggestions for further improvement.

ACKNOWLEDGEMENTS

Firstly, I would like to provide my deepest gratitude to my advisor, mentor, and friend, Dr. Stefan Forcey. I am incredibly blessed to have had the opportunity to work with someone so supportive and encouraging during my time at the university. I would also like to extend my gratitude to Dr. Español and Dr. Wilber for their time in reviewing this work and all of their valuable suggestions.

I would also like to thank my fellow students Luke, Mackenzie, Marissa, Cody, Wiley, and Oliver, who have been instrumental in my academic and personal development. At times when I felt like a failure, they were always there to provide good humor and lift my spirits. I can only hope to find such good friends as I begin my journey in graduate school at Michigan State.

Most importantly, I would like to mention my incredible parents and Lauren for all of their incredible support and time in listening to me complain about nearly everything. Words, literally, cannot express my gratitude for how much you all have done for me. Thank you for being so understanding of my work-filled nights.

Lastly, I would like to thank coffee, without which, this work would not have been completed. Your caffeine has allowed me to be more ”productive” than could possibly be conceived.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

THE BALANCED MINIMAL EVOLUTION PROBLEM

Phylogenetics is the study of evolutionary relationships among classes of organisms, which we call *taxa*. Understanding such relationships allows us to better understand evolutionary processes, infer particular driving forces behind such changes, and predict how these relationships will change in the future. This could influence conservation efforts to maintain genetic diversity, as well as the designs for new medicine. In particular, phylogenetics has proven to be useful in the design of vaccines for viruses that mutate relatively fast. These relationships are commonly described using a weighted tree called a *phylogeny*, which indicates shared relationships among taxa; however, a given problem may involve large numbers of possible phylogenetic trees. Among the set of admissible candidates, we have to decide which one provides the best representation of the input data. This is commonly referred to as the Phylogeny Estimation Problem (PEP). Utilizing molecular data collected from DNA, RNA, amino acids, etc. allows us to determine weights, which measure *dissimilarity* between pairs of taxa. Distance-based methods are a class of techniques that use this information to help select the best tree that is represented by our molecular data. The BME method, which we consider in this problem, asserts that the best candidate has the minimum weighted path length. An advantage of the BME method is

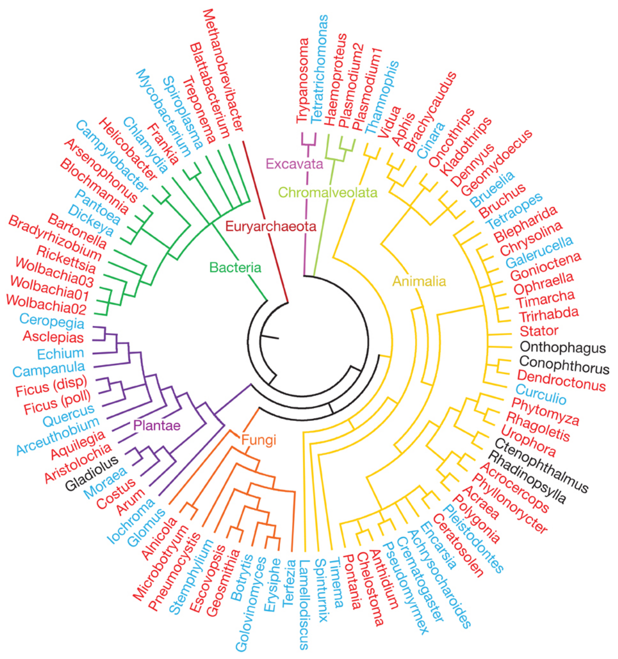
Figure 1.1: A complex example of a rooted phylogenetic tree. The connections between taxa indicate a common ancestor [11].

that it is known to be *statistically consistent*. This means that as we obtain more information related to the dissimilarity of species, then our solution approaches the true tree representing that data. Furthermore, the correct tree can be recovered even in instances of missing or corrupted data, provided that the error is within bounds.

Various methods for obtaining solutions have been proposed over the course of several decades (See [4] for a survey of current methods). In [17], Saitou and Nei suggested a popular greedy algorithm, known as Neighbor-Joining (NJ), which runs in polynomial time. The primary disadvantage is that this greedy, bottom-up approach, often obtains the incorrect tree. Developments by Gascuel and Steel [10], such as FastME, work in a similar fashion, but use more sophisticated operations called *edge moves* that make the approach more robust. While these algorithms are capable of handling a large number of taxa (e.g. 100), there is a significant trade-off between performance and accuracy. The need for faster, more accurate methods, has led us to undertake this work. Mixed-integer programming formulations, such as those presented in [5, 9], are currently being studied as a means towards greater accuracy in solving the BME problem. Catanzaro et al. proposed a model that uses binary decision variables according to whether or not an edge is present in the tree. Their model's constraints were derived using properties of the graphs being studied, while our formulation utilizes collections of facets from the BME Polytope described in [7, 8]. The particular approach used in developing the branch and bound routine for our algorithm is similar to the algorithm proposed in [13], despite significant differences in the problem structure.

In 2000, Pauplin [14] showed how to rapidly calculate the length of a phylogenetic tree without having to resort to branch length calculations. This work presented a path length function derived from bifurcations in phylogenetic trees. Recently, it was discovered that minimizing this length, which we define later, over the set of phylogenetic trees is equivalent to minimizing over a geometric object known as the Balanced Minimal Evolution Polytope [12]. Thus, the problem can be reformulated in terms of mathematical programming. While linear programming tends to be slow in comparison to these implicit enumeration schemes, the advantage is that, to a certain degree, we have the ability to control performance and accuracy. Our work seeks to provide a more balanced approach that allows the user to obtain a more accurate solution in a reasonable amount of CPU time. Later, we discuss these control options.

1.1 Definitions

Before we begin, we need to provide some basic definitions and terminology that will be useful for our purposes.

A *graph* G is a set of vertices $\mathcal{V}$, together with a set of edges $\mathcal{E}$, such that every element in $\mathcal{E}$ is of the form $\{x, y\}$ with $x, y \in \mathcal{V}$. To simplify this, we often say $G = (\mathcal{V}, \mathcal{E})$. If the pairs of vertices are unordered, we say that G is *undirected*. Generally speaking, the graphs we discuss are typically used to describe the relations between objects. In applications, graphs are used to model connections between a set of objects, called *nodes*, with a particular connection indicated by the presence of a shared *edge*. In our problem, the nodes represent taxa or branching points in an evolutionary network, while the edges represent lineages. We say that a graph has a *cycle* if there is a sequence in the edge set of the graph of the form $\{\{x_1, x_2\}, \{x_2, x_3\}, ..., \{x_N, x_1\}\}$. A graph is called a *tree* if it contains no cycles; if we moved the graph within its plane, it should resemble a tree.

Descriptions of the underlying connectivity structure of a tree, without regard to the labeling, is referred to as the tree's *topology*. We will call the vertices associated with only one edge *taxa* (or *leaves*) and the vertices associates with three edges, *internal nodes*. We say a tree is *binary* if every internal node has degree 3 (i.e. every parent node has two children). A *phylogenetic tree* is an undirected binary tree with labeled taxa. Phylogenetic trees can either be *rooted* or *unrooted*. The choice of a root is more or less arbitrary for our purposes, though it typically requires some known information about the network a priori (See Figure 1.2 for an example). To simplify our problem, we assume that we are working only with binary trees as opposed to the more general M-ary trees. It is known that any M-ary tree can be transformed into a binary tree by inserting dummy vertices and edges with null weight (See Figure 1.3). We can also describe the subtrees of a phylogenetic tree. A *clade* of a phylogenetic tree is a subset of the tree where an edge is chosen, and everything on one side of the edge is thrown away to create a subtree. Consequently, a *cherry* is a clade with 2 taxa.

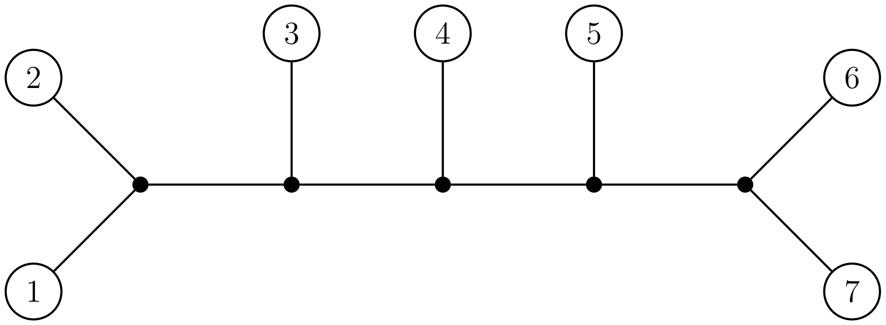

Figure 1.2: A simple example of an unrooted, binary phylogenetic tree for $n = 7$. Each taxon is associated with a labeling $1, 2, \ldots, 7$. This is a caterpillar tree because it has exactly two cherries.

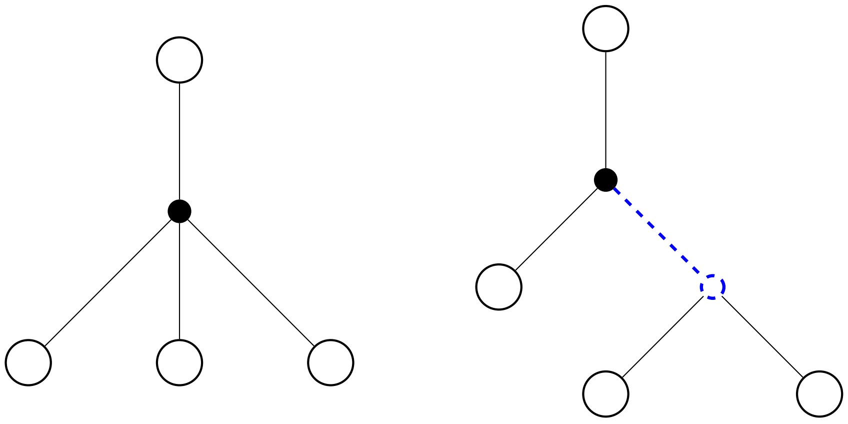

Figure 1.3: A ternary tree can be transformed into a binary tree using an insertion process. Here, the dashes indicate null edges and nodes. We avoid labeling the taxa because the ternary tree on the left has multiple binary representations.

1.2 Notation

Much of the notation used in the existing literature on the BME method tends to vary, despite the underlying similarities. Our convention is as follows:

Define $\mathcal{S} = \{1, 2, ..., n\}$ to be a list of distinct taxa that we wish to consider. Each element in $\mathcal{S}$ is a natural number, which corresponds to an individual taxon. Let the *dissimilarity vector* $\mathbf{d}$ with $\binom{n}{2}$ components be given, where each entry d_{ij} is positive and represents the dissimilarity between taxa i and j for each pair $\{i, j\} \subset \mathcal{S}$. This vector is obtained using molecular data, typically provided by biologists and geneticists. Let $\mathcal{T}$ be the set of all binary phylogenetic trees without edge weights. Then, for each tree $t \in \mathcal{T}$, there is a corresponding vector $\mathbf{x}(t)$ with $\binom{n}{2}$ components $x_{ij}(t)$ for each pair $\{i, j\} \subset \mathcal{S}$. Following Pauplin [14], we define

$$x_{ij}(t) := 2^{n-2-l_{ij}(t)}, \tag{1.1}$$

where $l_{ij}(t)$ is the number of internal nodes (degree 3 vertices) in the path connecting i and j in t. The additional factor of 2^{n-2} is used to rescale Pauplin's original coordinates, so that floating point errors are eliminated in the numerical computations. We use a lexicographic ordering of the entries

$$\mathbf{b} = \left(b_{12}, b_{13}, ..., b_{1n}, b_{23}, b_{24}, ...b_{2n}, ..., b_{(n-1)n}\right)$$

for vectors $\mathbf{d}$ and $\mathbf{x}$. If we are provided a binary tree T with non-negative weights on the edges, then we can calculate $\mathbf{d}$ by adding the weights on each of the edges connecting the path from i to j in T. Once we have calculated $\mathbf{d}$, we can determine the rescaled length of T using the path-length functional

$$\mathcal{L}(T) := \sum_{\substack{i,j \\ i<j}} d_{ij} 2^{n-2-l_{ij}(T)}. \tag{1.2}$$

Using our definition in (1.1), we can rewrite (1.2) as

$$\mathcal{L}(T) = \mathbf{d}_T \cdot \mathbf{x}(T). \tag{1.3}$$

Since T is its own BME tree, we are left to compute a single dot product. However, our task here is to find the BME tree represented by an arbitrary dissimilarity vector $\mathbf{d}$ using data that is potentially missing or corrupted. Therefore, we must extend this definition to handle any tree $t \in \mathcal{T}$ using a slight modification of (1.3). The penultimate form of our functional is

$$\mathcal{L}(t) = \mathbf{d} \cdot \mathbf{x}(t). \tag{1.4}$$

Here, $\mathcal{L}(t)$ will be minimized precisely when the minimizer t^* has the same topology as T and will be larger, otherwise. Using (1.1), we can equivalently describe the structure of a tree t by its unique vector representation $\mathbf{x}(t)$. This allows us to minimize (1.4) over $\mathbf{x}(t)$, with the minimizer, now, being $\mathbf{x}(t^*)$. Observe that our rescaling will not affect the solution obtained through the minimization procedure. In general, the minimizer is unique, provided it does not contain edge weights that are identically equal to zero. In the latter case, we will have a finite collection of trees that minimize $\mathcal{L}(t)$ simultaneously [8]. More detail on the minimization strategy and the technicalities of the problem is provided later in this work.

CHAPTER II

THE RELAXATION

2.1 The BME Polytope

An important aspect of our formulation involves particular restrictions on the solutions we seek. These requirements can be developed by studying the topological properties of the trees belonging to $\mathcal{T}$. Pair-wise distances on phylogenetic trees have a structure that allows us to treat them as points in $\mathbb{R}^{\binom{n}{2}}$. However, for a set $\mathcal{S}$, containing n taxa, one can show that there are exactly $(2n-5)!!$ potential trees that belong to $\mathcal{T}$ (See Table 2.1 for details). Minimizing our path length functional over each of these points would require us to construct the vector $\mathbf{x}$ that describes the topology of every tree in $\mathcal{T}$, in addition to their corresponding path lengths. A useful result presented in [12], states that minimizing over the set of trees in $\mathcal{T}$ is equivalent to minimizing over the convex hull of $\mathcal{T}$. Thus, taking the convex hull of these points, denoted $Conv(\mathcal{T})$, we arrive at the BME Polytope, here after denoted as BME(n). Informally, the convex hull can be thought of as a shrink-wrapping operation that encloses each of the points representing admissible candidates in hyperplanes. Equivalently, we can define a polytope as the intersection of half-spaces, using linear inequalities that describe the structure of the outermost faces called *facets.* If

Taxa	**Dimension**	**Vertices**	**Facets**
3	0	1	0
4	2	3	3
5	5	15	52
6	9	105	90262
n	$\binom{n}{2} - n$	$(2n-5)!!$	?

Table 2.1: This table contains some statistics for BME(n). The number of facets for BME(n) remains an open problem. Notice that the total number of facets from $n = 5$ to $n = 6$ increases drastically.

an inequality can be used to describe a facet, we call it a *facet inequality.* Facet inequalities allow us to implicitly maintain some of the geometric information of the trees without the need to explicitly determine $\mathbf{x}$ directly from a tree.

Polytopes have rich applications in mathematical programming problems, where the constraints take the form of inequalities. This allows us to reformulate the BME Problem as a *linear programming* problem where both the objective function and the constraint functions are linear. In this context, the BME Polytope represents the *feasible region* that is described by these inequalities. Consequently, obtaining a *complete* description of a polytope is often difficult. By complete, we mean that for every facet of the polytope, there exists a corresponding facet inequality that describes it. Furthermore, collections of facets for polytopes are often exponential or factorial in size, which are impractical for mathematical programming formulations. To circumvent this, we can consider a subset of the inequalities defining the complete polytope, which we call a *relaxation*.

2.2 The Splitohedron

The $\mathcal{NP}$ nature of the BME problem, as noted in [5], suggests that a complete description of the polytope is unlikely. Fortunately, we can develop relaxations of the BME Polytope using various combinations of the known facet inequalities. The performance of an algorithm can be greatly influenced by both the number of variables, commonly called *decision variables*, as well as the number of constraints defining the feasible region. Therefore, it is highly desirable to maintain a polynomial-sized formulation, meaning the input to the algorithm we use depends polynomially on the number of taxa being considered. To this end, we propose several inequalities used to construct the relaxation of BME(n).

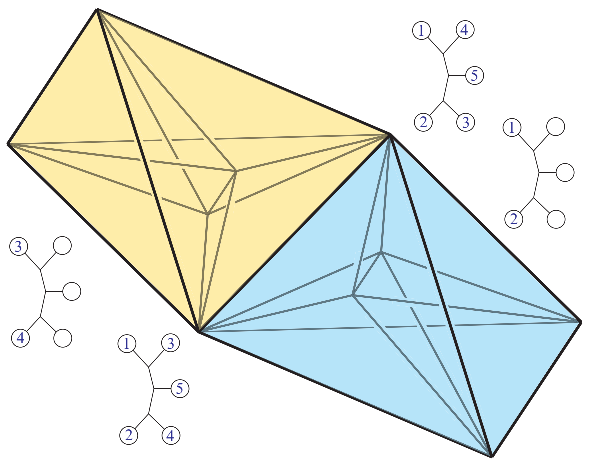

Figure 2.1: Two Caterpillar Faces for BME(5). These constraints force a lower bound on x_{ij}, which corresponds to an upper bound on the topological distance l_{ij}. These particular faces were studied in [7].

Proposition 1 (Caterpillar and Cherry Faces)**.** *For every* $i, j \in \mathcal{S}$*, with* $i \neq j$*,*

$$1 \leq x_{ij} \leq 2^{n-3}. \tag{2.1}$$

This inequality provides both a lower bound and an upper bound on each of the decision variables used in the model. These inequalities suffice to guarantee that our polytope is bounded, since it is contained within the hypercube $[1, 2^{n-3}]^{\binom{n}{2}}$. The right-hand side of the inequality follows immediately using the definition of x_{ij} and noting that every leaf must be separated using at least one internal node. These constraints are called the *Cherry Faces*. Similarly, on the left-hand side, the *Caterpillar Facets* follow because the distance between any two taxon in a tree is at most $n-2$ internal nodes away.

Proposition 2 (Kraft Equalities)**.** *Let $i, j \in \mathcal{S}$. Then for every $i \in \mathcal{S}$,*

$$\sum_{j;j\neq i} x_{ij} = 2^{n-2}. \tag{2.2}$$

The Kraft Equality is a necessary condition for a path length sequence to represent a phylogeny. These equalities are commonly encountered in information theory, specifically in *Huffman trees*, which are rooted, binary trees used to represent symbols in a coding alphabet. Interestingly, Huffman trees can be described using a path length sequence [16]. Therefore, we can think of a phylogeny as a Huffman tree encoded in a binary alphabet using the taxa as symbols in the code [5, 9]. We do not provide a proof of this inequality here, but one can derive this property using an inductive edge collapsing argument and an appropriate relabeling of the taxa.

Proposition 3 (Intersecting-Cherry Facets)**.** *Let $i, j, k \in \mathcal{S}$ be distinct. Then, for any collection of phylogenetic trees with either $\{i, j\}$ or $\{j, k\}$ as cherries, we have*

$$x_{ij} + x_{jk} - x_{ik} \leq 2^{n-3}. \tag{2.3}$$

This inequality will become strictly less than when a graph contains neither $\{i, j\}$ or $\{j, k\}$ as cherries. The proof that this inequality forms a facet of BME(n) can be found in [7] (See Theorem 4.7). In [5], an equivalent form of this inequality was proposed. The authors referred to these constraints as the *triangular inequalities.* In Figure 2.2, we provide the Schlegel diagram displaying a particular Intersecting-Cherry Facet for BME(5).

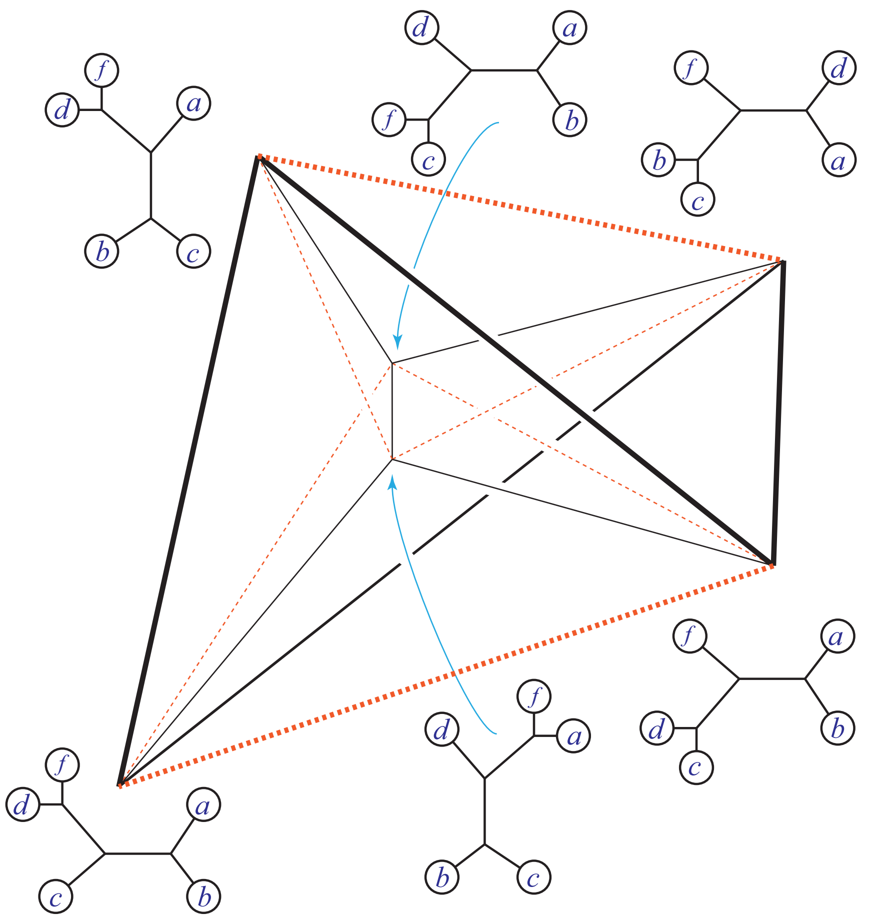

Figure 2.2: A visualization of an intersecting cherry facet for BME(5), presented in [7]. The vertices in this facet include any tree which contains $\{a, b\}$ or $\{b, c\}$ as cherries.

Proposition 4 (Split Facets)**.** *Consider* $\pi = \{S_1, S_2\}$, *a partition of* $\mathcal{S}$*. Let* $|S_1| := k \geq 3$ *and* $|S_2| := m \geq 3$*. Then for* $i, j \in S_1$

$$\sum_{\substack{i,j \\ i<j}} x_{ij} \leq (k-1)2^{n-3} \tag{2.4}$$

Here, $|\cdot|$ denotes the cardinality. For convenience, we will refer to types of splits using the cardinality of their partitions, e.g., we say a tree exhibits a (k, m)-split. This inequality allows us to have some control on the positioning of the taxa within a subgraph of a tree t. The split inequality is an equality for any tree that displays the split, and is a strict inequality for all others. In [8], we proved that this inequality, indeed, forms a facet of BME(n). We also showed that this inequality grows on the order of $\mathcal{O}(2^n)$, which will be relevant to our discussion on the performance of our proposed algorithm in later sections.

We define our relaxation of the BME Polytope as the intersection of half-spaces given by Propositions 1-4. This operation forms a new polytope, which we call the *Splitohedron*, denoted as *Sp*(n). Some properties regarding faces of Sp(n) are provided in Table 2.2. We now state a theorem obtained in [8] relating the vertices of Sp(n) and BME(n).

Classification	**Size of Collection**	**Vertices in Faces**
Caterpillar Faces	$\binom{n}{2}$	$(n-2)!$
Cherry Faces	$\binom{n}{2}$	$(2n-7)!!$
Intersecting Cherry Facets	$\binom{n}{2}(n-2)$	$2(2n-7)!!$
Kraft Equalities	n	-
Split-Facets	$2^{n-1}-\binom{n}{2}-n-1$	$(2m-3)!!(2k-3)!!$

Table 2.2: We provide some statistics for the inequalities used in our relaxation, Sp(n), based on [8]. Notice that the facets of first three classes of inequalities grow polynomially in n, while the facets for (k,m)-splits grow exponentially in n. The Kraft Equalities appear in all faces of the polytope, so they trivially contain all of the vertices.

Theorem 1. *Let t be a phylogenetic tree with n taxa. If the number of cherries of t is at least $\lceil \frac{n}{4} \rceil$, then $\mathbf{x}(t)$ is a vertex in both BME(n) and Sp(n). For $n \leq 11$ the statement holds regardless of the number of cherries.*

Theorem 1 allows us to estimate when we begin losing information under the relaxation. As long as $n \leq 11$, the Splitohedron will contain all the vertices of BME(n). Otherwise, we begin losing some of the vertices of the BME Polytope.

CHAPTER III

MODELING WITH LINEAR PROGRAMMING

3.1 Conventions for Linear Programming Models

Many problems in applications involve optimizing a particular quantity, which is subject to a set of constraints. This is called *constrained optimization.* These constraints can either take the form of inequalities, equalities, or combinations of the two. We say that a mathematical programming problem is a *linear programming* problem if it can be expressed in the form

$$\begin{aligned} \underset{\mathbf{x}}{\text{argmax}} \quad & \mathbf{c}^T \cdot \mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}, \end{aligned}$$

where $\mathbf{c}^T$, $\mathbf{x}$, and $\mathbf{b}$ are vectors. Here, A is an $m \times n$ matrix that consists of real numbers. Each entry of the vector $\mathbf{x}$ is called a *decision variable.* We call the space contained by matrix-vector inequality the *feasible region* for the problem because solutions must be contained in this space. This convention used for formulating our model generalizes other types of problems we might encounter. For example, if we wish to minimize an objective, rather than maximize, we simply multiply the objective function by a negative constant. Furthermore, the inclusion of equality constraints is redundant in the standard form because every equality constraint can be represented as a conjunction of two inequalities, one of which is multiplied by a negative to flip the direction of the inequality. Finally, if the right-hand side of the non-negativity constraint is not satisfied, we can simply make a substitution in the problem that shifts the lower bound of the new decision variable.

Some additional descriptive terms for a linear programming problem are often related to the range of values that decision variables can take, which often influences the approach used to solve the problem. In this current setup, each decision variable takes the value of a real number. For many problems, involving integer quantities, a decision variable with a continuous value does not make sense, e.g., sizes of bolts available in a hardware store. If the problem involves values from a discrete list of available values, we call that problem a *Discrete Programming Problem* (DP). Problems involving integer restrictions result in either a *Mixed-Integer Programming Problem* (MIP) or an *Integer Programming Problem* (IP). The latter case arises if all of the decision variables are restricted to be integers, while the former case involves only a few of the decision variables being required to take integer values.

In practice, formulating a problem as a continuous linear programming problem is advantageous in that these types of problems are well-studied, and we often have efficient methods to solve them. These formulations are particularly important because they are used in a variety of applications in prominent research areas such as game theory, transportation theory, and resource allocation. In stark contrast to the continuous setting, problems formulated as MIPs, IPs, and DP's are much more difficult to solve, despite their deceivingly simple appearance. Today, a substantial amount of modern research is devoted to the design of more efficient algorithms to handle these types of problems.

3.2 The Discrete Integer Linear Programming Model

The linearity of the half-spaces defining the Splitohedron and the underlying linear objective function suggest a linear programming based approach. Recall that, in Chapter 1, we defined the path-length functional that describes the length of a tree $t \in \mathcal{T}$. As previously noted, we seek a minimizer $\mathbf{x}(t^{\star})$ of this functional, but we delayed defining the region containing admissible solutions. Now that we have defined our relaxation of the BME Polytope, we can use it as the feasible region for our linear programming model. Our model is as follows.

Formulation (Discrete ILP)**.**

$$\underset{\mathbf{x}}{\text{argmin}} \quad \mathbf{d} \cdot \mathbf{x}$$

$$\textit{subject to} \qquad \sum_{j;j\neq i} x_{ij} = 2^{n-2}, \quad \forall i \in \mathcal{S} \tag{3.1}$$

$$x_{ij} + x_{jk} - x_{ik} \leq 2^{n-3}, \quad i,j,k \in \mathcal{S}, i \neq j \neq k \tag{3.2}$$

$$\sum_{\substack{i,j \in S_1 \\ i<j}} x_{ij} \leq (k-1)2^{n-3}, \quad k \geq 3, m \geq 3 \tag{3.3}$$

$$1 \leq x_{ij} \leq 2^{n-3}, \quad \forall i,j \in \mathcal{S}, i \neq j \tag{3.4}$$

$$x_{ij} \in \{2^k : k \in \mathbb{Z}_+ \cup \{0\}\}, \quad i,j \in \mathcal{S}, i \neq j \tag{3.5}$$

Here we use *argmin* because we want to return the argument $\mathbf{x}$ that minimizes the scaled path length $\mathcal{L}(\mathbf{x})$. This form of our functional considers $\mathbf{x}$ as a general vector without the dependence on t. This is a direct consequence of our relaxation, which introduces non-tree realizable vectors into the feasible region. Notice that (3.1)-(3.4) are the *Kraft Equalities*, *Intersecting Cherry Facets*, *Split Facets*, and the *Caterpillar and Cherry Faces*, respectively. These are the same inequalities we used to define our polytope $\mathrm{Sp}(n)$. The last constraint, (3.5), states that each of the decision variables belongs to the set of powers of two. This allows us to avoid encountering many of the potential solutions that might not belong to the BME Polytope that are present in our relaxation.

3.3 Excluding the Four-Point Condition

An interesting question related to the options for model constraints is whether or not particular inequalities are necessary to produce a phylogenetic tree. Many existing models such as those presented in [5, 9], include what is commonly called the *Four-Point Condition.*

Proposition 5 (Four-Point Condition)**.** *Let $i, j, k, l \in \mathcal{S}$ be distinct. Then any binary phylogenetic tree in $\mathcal{T}$ satisfies*

$$x_{ij}x_{kl} \geq min\{x_{ik}x_{jl}, x_{il}x_{jk}\}. \tag{3.6}$$

This particular form can be derived by transforming the version of the Four-Point Condition presented in [3], using our definition of x_{ij}. In this form, the inequalities are *fully* nonlinear, as opposed to the non-transformed inequalities, which can be linearized. As noted in [5], the Four-Point Condition and Kraft Equalities are necessary to completely characterize the path length sequences in $\mathcal{T}$.

We chose to exclude the Four-Point Condition so that we preserve the linearity in our formulation. At present, it is known that these conditions are necessary, but not sufficient for binary phylogenetic trees. Our experimental observations of test cases lead us to conjecture that our algorithm, in conjunction with (3.1)-(3.5), is sufficient to provide a tree realization for $\mathbf{x}$. In practice, the inclusion of the Four-Point Condition is troublesome because the total number of inequalities generated by this constraint, alone, is exponential. This results in a notable increase in the time required to solve the problem. One can circumvent this problem, of course, by dynamically adding the exponential constraints as needed, rather than including them all at once.

CHAPTER IV

THE BRANCH AND BOUND ALGORITHM

4.1 Structure of the Algorithm

Solving $\mathcal{NP}$-hard optimization problems to optimality is an incredibly difficult process. One of the most common techniques for solving this class of problems is called *Branch and Bound*. This process is recursive, breaking the original problem into *subproblems*, which are easier to solve. We sometimes describe the recursive nature of this process as traversing a binary search tree, where each node represents an individual problem. We provide an example of a search tree in Figure 4.1. To begin, the discrete valued constraints on the decision variables are relaxed. This allows us to utilize Linear Programming algorithms because the decision variables, now, admit a continuum of values. The initial Linear Programming problem that results from this relaxation is called the *root LP*. Computing its solution allows us to determine the feasibility of the original problem. If the solution to the root LP meets our original restrictions for the decision variables, then the branch and bound routine terminates. Otherwise, a *branching rule* is used to determine how to divide the solution space, resulting in two subproblems (or nodes). After solving each of these problems, a *selection strategy* is used to determine which nodes to explore in the search tree. If

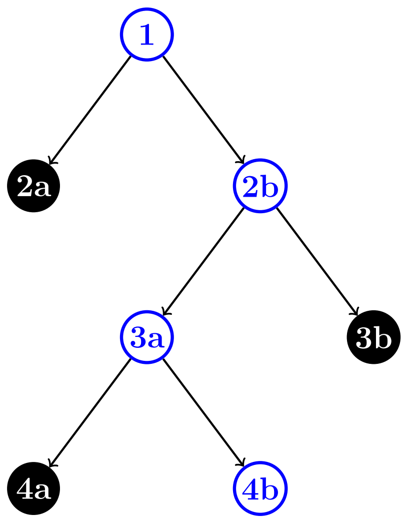

Figure 4.1: An example of a Branch and Bound binary search tree. Active nodes follow along the indicated blue path, while fathomed nodes are labeled in black.

a node is not explored, we say the node was *fathomed* or *pruned*. Once a node is selected for exploration, the process is repeated. The inequalities used in the creation of individual problems, along the path, are maintained throughout the search. Once we obtain a feasible solution satisfying the constraints on the decision variables, we can update the global bound on the objective and use it to prune subproblems, which provide a less optimal objective value. We are permitted to prune subproblems in this manner, even if the discrete constraints on the decision variables are not satisfied. We call the best, current solution the *incumbent solution*. This pruning process allows us to eliminate significant portions of the search space, which effectively reduces the algorithm's running time. Repeatedly applying this process allows us to eventually obtain the optimal solution to our original Discrete Programming problem.

Designing a Branch and Bound algorithm involves a great deal of experimentation. One has a large amount of flexibility among choices for bounding functions, selection strategies, and the branching rules available for implementation. An additional degree of freedom with Branch and Bound algorithms, often used to improve performance, are *heuristics*. Heuristics, on a general level, involve incorporating problem-dependent experimental information into an algorithm to eliminate unlikely candidates in the branch and bound process and promote faster convergence. Such techniques allow for fast computation of often intractable, $\mathcal{NP}$-hard optimization problems. However, approaches that utilize heuristics cannot guarantee that the solution obtained is the global minimizer (or maximizer) for the problem.

We propose two branching methods to handle the Discrete Integer Linear Program developed in Chapter 3. We provide pseudo-codes for the main algorithm in Figure 4.2 and the branching routines for pure Branch and Bound and a heuristic Branch and Bound techniques in Figures 4.3 and 4.4, respectively. Both routines are part of the main script, which is used to evaluate the feasibility of the relaxed problem. The main difference between the two methods is that one method utilizes a heuristic, while the other relies exclusively on the Branch and Bound process.

Algorithm 1: The Discrete ILP Main Algorithm

```
Require: Identification of minimizer $\mathbf{x}(t^\star)$
Input: $\mathbf{d}$, $A$, $\mathbf{b}$, $A_{eq}$, $\mathbf{b}_{eq}$, $lb$, $ub$, $n$, maxiter0, maxiter1, $\epsilon$, heuristic
Output: $\mathbf{x}^\star$, $\mathcal{L}^\star$, status
 1: Initialization: set bound = $+\infty$ and iter = 0
 2: Solve the relaxation at the root node $\to \mathbf{x}_0$, $\mathcal{L}_0$, status0
 3: if status = infeasible then
 4:    Return $\mathbf{x}^\star, \mathcal{L}^\star = \emptyset$
 5: else
 6:    if heuristic = 0 then
 7:       Call branch0 $\to \mathbf{x}^\star, \mathcal{L}^\star$, status
 8:    else if heuristic = 1 then
 9:       Find all entries s.t. $|x_0(i) - 2^{n-3}| < \epsilon$ for $i = 1, \ldots, \binom{n}{2}$
10:       Fix positions and update $A_{eq}, \mathbf{b}_{eq}$, $ub$
11:       Call branch1 $\to \mathbf{x}^\star, \mathcal{L}^\star$, status
12:    end if
13: end if
```

Figure 4.2: Main algorithm for the Discrete ILP problem. The algorithm decides to call a particular branching function depending on whether or not we use a heuristic.

Algorithm 2: Branch0 Algorithm

Input: $\mathbf{d}$, A, $\mathbf{b}$, A_{eq}, $\mathbf{b}_{eq}$, lb, ub, $\mathbf{x}_t, \mathcal{L}_t$, ϵ, bound

Output: $\tilde{\mathbf{x}}, \tilde{\mathcal{L}}$, status, bb

```
1:  Solve the relaxation at the current node $\rightarrow \mathbf{x}_0, \mathcal{L}_0$, status0
2:  **if** status0 = infeasible or $\mathcal{L}_0 >$ bound **then**
3:     Return input, bb $\leftarrow$ bound
4:  **else**
5:     Compute $\mathcal{E} = \max_i\{\min\{|x_0(i) - 2^{\lfloor log_2(x_0(i)) \rfloor}|, |x_0(i) - 2^{\lceil log_2(x_0(i)) \rceil}|\}\}$
6:     **if** $\mathcal{E} < \epsilon$ or iter $>$ maxiter1 **then**
7:        **if** $\mathcal{L}_0 <$ bound **then**
8:           $\tilde{\mathbf{x}} \leftarrow \mathbf{x}_0, \tilde{\mathcal{L}} \leftarrow \mathcal{L}_0$, bb $\leftarrow \mathcal{L}_0$
9:        **else**
10:          Return input, bb $\leftarrow$ bound
11:       **end if**
12:       Return
13:    **end if**
14:    Select a branching variable $x_0(j)$
15:    Build subproblem $\mathcal{P}_1$ : Set $x_0(j) \leq 2^{\lfloor log_2(x_0(j)) \rfloor}$ in $\{A, \mathbf{b}\} \rightarrow \{A_1, \mathbf{b}_1\}$
16:    Build subproblem $\mathcal{P}_2$ : Set $x_0(j) \geq 2^{\lceil log_2(x_0(j)) \rceil}$ in $\{A, \mathbf{b}\} \rightarrow \{A_2, \mathbf{b}_2\}$
17:    iter $\leftarrow$ iter + 2
18:    Call branching routine for $\mathcal{P}_1 \rightarrow \mathbf{x}_1, \mathcal{L}_1$, status1, bound1
19:    **if** bound1 < bound and status1 = feasible **then**
20:       $\tilde{\mathbf{x}} \leftarrow \mathbf{x}_1, \tilde{\mathcal{L}} \leftarrow \mathcal{L}_1$, bound $\leftarrow$ bound1, bb $\leftarrow$ bound1, status $\leftarrow$ status1
21:    **else**
22:       Return $\mathcal{P}_1$ input data, bb $\leftarrow$ bound
23:    **end if**
24:    Call branching routine for $\mathcal{P}_2 \rightarrow \mathbf{x}_2, \mathcal{L}_2$, bound2, status2
25:    **if** bound2 < bound and status2 = feasible **then**
26:       $\tilde{\mathbf{x}} \leftarrow \mathbf{x}_2, \tilde{\mathcal{L}} \leftarrow \mathcal{L}_2$, bb $\leftarrow$ bound2, status $\leftarrow$ status2
27:    **end if**
28: **end if**
```

Figure 4.3: Branching algorithm for pure Branch and Bound.

Algorithm 3: Branch1 Algorithm

```
Input: $\mathbf{d}$, $A$, $\mathbf{b}$, $A_{eq}$, $\mathbf{b}_{eq}$, $lb$, $ub$, $\mathbf{x}_t$, $\mathcal{L}_t$, $\epsilon$, bound
Output: $\tilde{\mathbf{x}}, \tilde{\mathcal{L}}$, status, bb
1: Solve the relaxation at the current node $\rightarrow \mathbf{x}_0, \mathcal{L}_0$, status0
2: if status0 = infeasible or $\mathcal{L}_0 >$ bound then
3:    Return input, bb $\leftarrow$ bound
4: else
5:    Compute $\mathcal{E} = \max_i \{\min\{|x_0(i) - 2^{\lfloor log_2(x_0(i)) \rfloor}|, |x_0(i) - 2^{\lceil log_2(x_0(i)) \rceil}|\}\}$
6:    if $\mathcal{E} < \epsilon$ or iter $>$ maxiter1 then
7:       if $\mathcal{L}_0 <$ bound then
8:          $\tilde{\mathbf{x}} \leftarrow \mathbf{x}_0, \tilde{\mathcal{L}} \leftarrow \mathcal{L}_0$, bb $\leftarrow \mathcal{L}_0$
9:       else
10:         Return input, bb $\leftarrow$ bound
11:      end if
12:      Return
13:   end if
14:   if iter $>$ maxiter0 then
15:      Find an entry $k = \operatorname{argmin}_i |x_0(i) - 2^{[log_2(x_0(i))]}| < \epsilon$ for $i = 1, \ldots, \binom{n}{2}$
16:      Set $x_0(k) = 2^{[log_2(x_0(k))]}$ and update $A_{eq}, \mathbf{b}_{eq}$
17:   end if
18:   Select a branching variable $x_0(j)$
19:   Build subproblem $\mathcal{P}_1$ : Set $x_0(j) \leq 2^{\lfloor log_2(x_0(j)) \rfloor}$ in $\{A, \mathbf{b}\} \rightarrow \{A_1, \mathbf{b}_1\}$
20:   Build subproblem $\mathcal{P}_2$ : Set $x_0(j) \geq 2^{\lceil log_2(x_0(j)) \rceil}$ in $\{A, \mathbf{b}\} \rightarrow \{A_2, \mathbf{b}_2\}$
21:   iter $\leftarrow$ iter + 2
22:   Call branching routine for $\mathcal{P}_1 \rightarrow \mathbf{x}_1, \mathcal{L}_1$, status1, bound1
23:   if bound1 $<$ bound and status1 = feasible then
24:      $\tilde{\mathbf{x}} \leftarrow \mathbf{x}_1, \tilde{\mathcal{L}} \leftarrow \mathcal{L}_1$, bound $\leftarrow$ bound1, bb $\leftarrow$ bound1, status $\leftarrow$ status1
25:   else
26:      Return $\mathcal{P}_1$ input data, bb $\leftarrow$ bound
27:   end if
28:   Call branching routine for $\mathcal{P}_2 \rightarrow \mathbf{x}_2, \mathcal{L}_2$, bound2, status2
29:   if bound2 $<$ bound and status2 = feasible then
30:      $\tilde{\mathbf{x}} \leftarrow \mathbf{x}_2, \tilde{\mathcal{L}} \leftarrow \mathcal{L}_2$, bb $\leftarrow$ bound2, status $\leftarrow$ status2
31:   end if
32: end if
```

Figure 4.4: Branching algorithm using a LNS heuristic. Once we specify a tolerance, the algorithm determines if a decision variable can be fixed before branching.

4.2 Branch Selection Strategy

During the branching process, we would like to know how to select the branching variable used to build the subproblems. Various selection strategies currently exist, which allow us to make these decisions. In our approach, we select the branching variable by examining each decision variable's distance to its adjoining powers of two. We first select the minimum distance between each decision variable and its adjoining powers of two. Then, find the entry that maximizes this result. For a vector $\mathbf{x}_0$, this is the entry i satisfying

$$\underset{i}{\operatorname{argmax}}\{\min\{|x_0(i) - 2^{\lfloor log_2 x_0(i) \rfloor}|, |x_0(i) - 2^{\lceil log_2 x_0(i) \rceil}|\}\}. \tag{4.1}$$

Rather than utilizing our lexicographic ordering convention described earlier, it is preferable to think of $\mathbf{x}_0$ as a vector of length $\binom{n}{2}$ with entries $(x_0(1), x_0(2), \ldots, x_0(N))$. This mapping between the orderings allows for a simpler numerical implementation. Given the structure for the sequence of powers of two, our branching strategy is more likely to select variables whose distance is close to that of a cherry. The distances separating powers of two greatly increase as we consider more terms. Therefore, we think of this as a bottom-up selection strategy. Our particular choice for a branching strategy was guided by examining output from subproblems within the search tree. Specifically, we observed that the linear programming algorithm used in the relaxations tended to prioritize finding cherries during the branch and bound process before clustering nearby items.

Alternatively, we could have devised a more complex selection strategy that incorporates the effects from previous choices for branching variables into subsequent selections. These types of strategies are commonly used in Branch and Bound algorithms, but we opted for a simpler approach for an ease of implementation. It is well known that selection strategies greatly influence the performance of Branch and Bound algorithms. Existing strategies can be improved using sophisticated techniques such as machine-learning and statistics to guide the exploration (See for example [1]). One could adopt this framework for our problem, but an in-depth discussion of advanced strategies is beyond the scope of this work.

4.3 Large Neighborhood Search and the Tolerance Parameter

In some cases, pure Branch and Bound strategies might not be practical in terms of CPU time. The problem with pure Branch and Bound strategies is that they can produce solutions that are invalid, yet feasible, due to the introduction of floating point error. On the other hand, careless rounding schemes can often lead to instabilities in the Branch and Bound process. The algorithm may converge to the optimal solution slowly, so additional user options, such as heuristics become necessary to improve numerical convergence. Once we begin making assumptions about the solutions, we eliminate a large portion of available answers. While one can no longer guarantee that the solution obtained from the Branch and Bound process is optimal, using carefully designed heuristics can allow one to obtain qualitatively good solutions. Here, "good" refers to a solution that is valid (discrete constraints are satisfied), feasible, and either optimal or quasi-optimal.

After developing a pure Branch and Bound algorithm to handle our model, we performed numerical experiments to determine possible improvements for tuning our algorithm. An examination of the root LP and intermediate solutions for subproblems suggest that cherries are determined first and once an entry becomes relatively close to a power of two, its changes in subsequent problems are minimal. Guided by this observation, we decided to utilize a rounding scheme, which adjusts and fixes an entry of the solution to a subproblem while maintaining feasibility. We refer to this fixing process as a *Large Neighborhood Search* (LNS). This operation is performed only if the entry is within an allowable distance ϵ to its nearest power of two neighbor. We call ϵ the *tolerance parameter*, which characterizes how radical or conservative we wish to be with the fixing process. This approach differs from existing Branch and Bound algorithms for the BME Problem in that users have an option to control the frequency of rounding used in the search tree. As $\epsilon \to 0$, we rely less on rounding and the solution intuitively becomes closer to the solution from a pure Branch and Bound approach.

The heuristic option for the Discrete ILP algorithm uses the fixing procedure in two places: the root LP and inside the Branch and Bound routine. During the feasibility check for the root LP, the heuristic option determines the number of cherries present by examining its solution. If it finds that there are at least two cherries in $\mathbf{x}_0$, the algorithm fixes their entries to the values given by their respective upper bounds, inside A_{eq} and $\mathbf{b}_{eq}$. Then, assuming all the cherries have been found, we can reset the upper bounds for the remaining decision variables. We send these arguments to the Branch and Bound function to begin the search. The algorithm performs pure Branch and Bound until we reach maxiter0, then it switches to a combined feature of LNS with Branch and Bound to solve the remaining subproblems. Before the algorithm branches, it determines if entries in the solution at the current node are eligible for rounding. We select a candidate for fixing if an entry of $\mathbf{x}_0$ satisfies

$$\underset{i}{\operatorname{argmin}} |x_0(i) - 2^{[log_2(x_0(i))]}| < \epsilon, \tag{4.2}$$

where $\epsilon > 0$ is a prescribed rounding tolerance and $[\cdot]$ is the nearest integer function. If the algorithm finds an entry k, satisfying (4.2), it rounds $x_0(k)$ to its nearest discrete-allowed value by setting

$$x_0(k) = 2^{[log_2(x_0(k))]} \tag{4.3}$$

in the equality constraints $\{A_{eq}, \mathbf{b}_{eq}\}$ to be maintained indefinitely in the search.

Currently, our approach does not use previous information from the branching process to decide how to pick variables to fix. In future developments, we would like to experiment with more intelligent heuristics. In particular, we would like to consider utilizing approaches such as a *Relaxation Induced Neighborhood Search* (RINS) and *guided dives*, which are better equipped to handle infeasibilities and unproductive searches, specifically related to rounding, using sophisticated backtracking processes. Authors in [6] develop these methods, exploring how each of the methods define, search, and diversify neighborhoods to improve incumbent solutions.

CHAPTER V

NUMERICAL RESULTS AND CONCLUSIONS

5.1 Numerical Implementation

For simplicity, we chose to implement the algorithms developed in Chapter 4 using MATLAB – the code for the algorithms can be found in the appendix. First, the function BMEineq generates the constraints for BME(n), which are passed as arguments, along with the dissimilarity data, to the main algorithm – this is where we begin the Branch and Bound process. To solve the relaxations at each node, in our search tree, we use MATLAB's linprog solver. In particular, we chose the dual-simplex algorithm because it is designed to handle large-scale optimization problems involving many decision variables and constraints. In the options structure for the solver, we specified a constraint tolerance of 1×10^{-10}. We found that this choice allows us to maintain a considerable amount of accuracy without significantly hindering performance.

In our algorithms, we defined two variables, maxiter0 and maxiter1, as stopping criterion. Pure Branch and Bound uses only the former, while the heuristic option utilizes both. In either case, maxiter0 was set to 1.5×10^4. The value of maxiter1 in the heuristic option was set to twice the value of maxiter0. The idea, here, is that we let the program run using pure Branch and Bound initially, then switch to the heuristic, to try again, if the initial run was unproductive. Our algorithms terminate if we exceed the maximum number of allowable iterations or the solution at the current node has entries within a distance ϵ from the nearest, power of two. In our simulations, we chose ϵ to be 1×10^{-4}. For our purposes, we used a single tolerance constraint in our branching algorithms, although, in principle, we could have defined different tolerances, say ϵ_0, ϵ_1, and ϵ_2, to use for stopping criterion, branching variable selection, and the LNS search, respectively. Once the algorithm terminates, one can draw the phylogenetic tree by passing the solution to the distance function, which infers the topological distances l_{ij} of the phylogenetic tree.

5.2 Error Metrics

To test the effectiveness of our algorithm, we can analyze the "distance" between a tree returned by the algorithm $t^\star$ and the true solution T. One might be tempted to use standard norms, such as

$$\|\mathbf{x}\| = \sum_{i=1}^{N} |x_i|, \tag{5.1}$$

and

$$\|\mathbf{x}\|_2 = \left(\sum_{i=1}^{N} x_i^2 \right)^{1/2}. \tag{5.2}$$

Standard norms, such as (5.1) and (5.2), display no preference in the ordering among the entries of a vector. Therefore, it is possible to obtain a completely different tree by carefully interchanging the entries in $\mathbf{x}$. In other words, the order is important because it inherently contains topological information related to the phylogenetic tree we are studying. Therefore, we will need a more sophisticated metric that considers this ordering.

In [15], Robinson and Foulds proposed a new distance, which, in the same paper, they proved was a metric. This distance is commonly called the *Robinson-Foulds metric*, which we denote as $d(t_1, t_2)$. Given two trees, t_1 and t_2, the metric describes the total number of "moves" required to transform t_1 into t_2 or vice-versa. More specifically, these moves are referred to as edge contractions and expansions. A *contraction* is an operation on a tree that collapses an edge between two internal nodes, while an *expansion* inserts an edge between two internal nodes. It should be noted that these moves are restricted to adjoining internal nodes and they are inverse operations of each other. The Robinson-Foulds distance has the properties of a metric space, namely:

(i) $d(t_1, t_2) > 0$, if $t_1 \neq t_2$,

(ii) $d(t_1, t_2) = 0$ if and only if $t_1 = t_2$,

(iii) $d(t_1, t_2) = d(t_2, t_1)$, and

(iv) $d(t_1, t_3) = d(t_1, t_2) + d(t_2, t_3)$, for $t_1, t_2, t_3 \in \mathcal{T}$.

Proofs of these properties are presented in [15]. We pay particular attention to properties (i) and (ii) in our error analysis. In using this metric, we make an underlying assumption that we are comparing two trees. For our algorithm, we have not yet proved that our process returns a binary tree, so this step must be verified by inspection. We expect $d(\cdot,\cdot)$ to take non-negative, integer values, given the interpretation of the edge moves used in the transformation process.

Interestingly, this metric is related to the partitions on the list of taxa, which we denoted as $\pi = \{S_1, S_2\}$. We referred to these partitions of $\mathcal{S}$ as splits. Recall that any tree displaying a split S_1 satisfies

$$\sum_{\substack{i,j \\ i<j}} x_{ij} = (k-1)2^{n-3}, \tag{5.3}$$

where k is the cardinality of S_1. Using this fact, we define the characteristic function for splits as

$$\mathcal{X}_\pi(t_1,t_2) := \begin{cases} 1, & \text{if either } t_1 \text{ or } t_2 \text{ displays } \pi\text{, while the other does not,} \\ 0, & \text{otherwise.} \end{cases}$$

The above definition considers a partition of $\mathcal{S}$ and takes a logical role in differentiating splits among the two trees. Using (5.3), we can determine whether or not a split is displayed in a tree. Combining this information we can now define $d(t_1,t_2)$ in terms of characteristic functions, leading to

$$d(t_1,t_2) := \sum_{\pi=\{S_1,S_2\}} \mathcal{X}_\pi(t_1,t_2). \tag{5.4}$$

The sum is implicitly taken over all partitions of $\mathcal{S}$ with the additional requirement that each split have a cardinality of at least two. This ensures that problems do not arise in the calculation of the characteristic function from (5.3).

5.3 Results

We examined the performance of our algorithms using both perfect and noisy initial data. The test data was obtained from a collection of trees, with $7 \leq n \leq 10$, having various topologies and edge weights. We also considered heavily unbalanced trees, which contain edge weight(s) that are considerably larger than others, in an attempt to fool the algorithm. Additionally, we observed that the algorithms perform well on trees that contain large numbers of cherries. It has been conjectured that these topologies promote the greatest computational difficulties for solving the BME problem. To this end, we chose to study cases where the underlying topology is either a caterpillar tree or a tree that contains few cherries. Specific information for the test cases used in the simulation is presented in the appendices.

Results from both simulations were compared to the true solutions using the Robinson-Foulds metric defined above. In the instances of noisy data, we assumed the data in $\mathbf{d}$ was perfect, then perturbed it by a vector $\boldsymbol{\delta}$, whose entries are normally distributed, i.e., $\delta_{ij} \sim \mathcal{N}(0, \sigma^2)$. We chose $\sigma = 1$ in our experiments, i.e., the variables obey a standard normal distribution. This perturbation creates a modified objective function

$$(\mathbf{d} + \boldsymbol{\delta}) \cdot \mathbf{x}.$$

Ideally, our algorithm should be adept at handling noise in the input data. More specifically, the algorithm should be able to return the same solution using corrupted or noisy data, provided the noise is contained within the *safety radius*. For distance based methods, the safety radius for BME(n) is denoted as $\rho_n \hat{T}$, where $\hat{T}$ is the smallest edge length in T. For experients involving the safety radius, each tree in the test suite is known. Thus, we denote it by T. Following [18], we say that we are within the safety radius if

$$\|\boldsymbol{\delta}\|_\infty < \rho_n \hat{T}$$

holds. For our experiments, $\hat{T} = 1$, so the above condition simplifies. While the safety radius for our approach has not been identified, we include this information so that we could quantify the amount of noise added in the experiments and observe how this noise affects the solutions obtained by the algorithm.

The results for our numerical experiments, presented in Tables 5.1 and 5.2, demonstrate the performance our algorithms. For smaller, more manageable values of n, we see that pure Branch and Bound is preferable to the heuristic option in terms of CPU time. However, as we consider larger cases, the problems become increasingly difficult and pure Branch and Bound begins to fail. Cases where the algorithm failed to produce a tree-realizable solution were indicated by "—." For many of the trials, the L_∞-norm for the noise was relatively large. Based on the measured error, it appears that the noise had little to no qualitative affect on the solutions obtained by the algorithm. This could indicate a possible insensitivity among a subset of decision variables. Alternatively, given the way we devised the experimental noise, it is possible that more sensitive variables had negligible perturbations. In order to examine this more closely, a complete statistical analysis of the algorithm needs to be considered.

n	**Test Case**	**CPU Time** (s)	$d(t^\star, T)$
7	N7T1	0.20	0
	N7T2	0.11	0
8	N8T1	1.328	0
	N8T2	0.88	0
9	N9T1	—	—
	N9T2	1.59	0
10	N10T1	—	—
	N10T2	32.94	0

n	**Test Case**	**CPU Time** (s)	$d(t^\star, T)$
7	N7T1	1.39	0
	N7T2	1.50	0
8	N8T1	2.11	0
	N8T2	1.81	0
9	N9T1	3.77	0
	N9T2	2.95	0
10	N10T1	192.73	0
	N10T2	41.55	0

Table 5.1: Results for both the pure Branch and Bound algorithm (top) and Branch and Bound using the LNS heuristic (bottom), using perfect data.

n	**Test Case**	**CPU Time** (s)	$\lVert\boldsymbol{\delta}\rVert_\infty$	$d(t^\star_\delta, T)$
7	N7T1	0.20	2.04	0
	N7T2	0.31	1.22	0
8	N8T1	122.36	2.02	0
	N8T2	59.73	2.51	0
9	N9T1	12.08	1.68	0
	N9T2	345.80	0.36	0
10	N10T1	—	0.87	—
	N10T2	—	0.97	—

n	**Test Case**	**CPU Time** (s)	$\lVert\boldsymbol{\delta}\rVert_\infty$	$d(t^\star_\delta, T)$
7	N7T1	1.61	2.04	0
	N7T2	1.28	1.22	0
8	N8T1	129.25	2.02	0
	N8T2	1.95	2.51	0
9	N9T1	36.94	1.68	0
	N9T2	5.48	0.36	0
10	N10T1	348.51	0.87	0
	N10T2	75.41	0.97	0

Table 5.2: Results for both the pure Branch and Bound algorithm (top) and Branch and Bound using the LNS heuristic (bottom), using noisy data.

One of the primary considerations of our algorithms is CPU time. However, the interpretation might be misleading. One might be tempted to begin comparing our results to those presented in [5] and [9], but, first, one needs to consider other factors such as computing platforms, implementation in various coding languages and professional software, and the test cases used in the experiments. For instance, algorithms designed in a high-performance computing environment, or in the framework of commercial optimization software, such as CPLEX, are better equipped to handle the computational workload required to solve large scale problems. Therefore, the performance results serve only as a benchmark for relative comparison.

5.4 Future Considerations

In the future, we would like to obtain qualitative information regarding the solutions returned by our algorithm. For instance, can we guarantee that our algorithm, in conjunction with current constraints, returns a minimizer $\mathbf{x}^\star$ that can be realized by a phylogenetic tree t? Additionally, we plan to add a verification feature to our code that determines if the solution returned by the algorithm has a tree representation. We would also like to understand how the use of the LNS heuristic, and heuristics, in general, influence convergence. This aspect could be further explored using more statistically guided selection strategies and branching rules mentioned in Chapter 4. By further studying the geometric structure of BME(n), we could include more polynomial-sized classes of facets to obtain refinements of our relaxation. This could possibly allow us to extend results on the accuracy of Sp(n) while maintaining a manageable-sized formulation.

An advantage of our approach is the independent structure of the computations in the Branch and Bound search tree. Such problems are excellent candidates for parallel processing. Exploring parallel computing options allows us to improve the diversity of our search space, for example, by simultaneously branching on multiple variables, in an effort to obtain good bounds earlier in the search. This would be highly beneficial to the pruning process because we could potentially eliminate larger portions of the feasible region, which effectively reduces the CPU time; however, great care must be taken to design parallel algorithms, so that the jobs on processors are evenly distributed. Otherwise, it is likely that a subset of the processors will become idle during the computations, which significantly slows performance. In some cases, the performance can be so poor that sequential processing is favorable to parallel processing. For this reason, we do not consider this development here. More detail on parallelization of Branch and Bound can be found in [2, 6].

In Chapter 3, as we developed the Discrete Integer Linear Programming model, we noted that our formulation size is $\mathcal{O}(2^n)$ because we chose to include all of our Split Facets. In terms of computational performance, this inclusion is not ideal because the resulting constraint matrix for inequalities becomes rather large. The time required to solve the large system of linear equations greatly increases. To circumvent this in future developments, we would like to formulate a polynomial-sized version of our current model, using a process that dynamically adds the split facets, as needed. This separation device will manage our split inequalities, forming *local refinements* of $\mathrm{Sp}(n)$. Local refinements are reasonable if the initial solution is placed in a neighborhood of the minimizer. Therefore it may be necessary to investigate more sophisticated linear programming solvers.

While MATLAB is considerably easy to use, in terms of implementing our algorithms, we believe performance would benefit greatly if compiled languages, such as C++ or Fortran, were used to perform the dense calculations. In this sense, a cross-platform setup would be ideal, where the constraints can be easily generated in MATLAB, or Julia, and passed to the Branch and Bound code, written in a high-performance environment.

BIBLIOGRAPHY

[1] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, (33):42–54, 2005.

[2] D. Bader, W. Hart, and C. Phillips. Parallel algorithm design for branch and bound. In H. Greenberg, editor, *Tutorials on Emerging Methodologies and Applications in Operations Research*, chapter 5, pages 1–44. Kluwer Academic Press, 2004.

[3] P. Buneman. A note on the metric properties of trees. *Journal of Combinatorial Theory*, (17):48–50, 1974.

[4] D. Catanzaro. The minimal evolution problem: Overview and classification. *Networks*, 53(2):112–125, 2007.

[5] D. Catanzaro, M. Labbé, R. Pesenti, and J.-J. Salazar-González. The balanced minimal evolution problem. *INFORMS Journal on Computing*, 24(2):276–294, 2012.

[6] E. Danna, E. Rothberg, and C. Le Pape. Exploring relaxation induced neighborhoods to improve mip solutions. *Mathematical Programming*, 102(1):71–90, 2005.

[7] S. Forcey, L. Keefe, and W. Sands. Facets of the balanced minimal evolution polytope. *Journal of Mathematical Biology*, 73(2):447–468, 2016.

[8] S. Forcey, L. Keefe, and W. Sands. Split facets of balanced minimal evolution polytopes and the permutoassociahedron. *arXiv:1608.01622*, (pre-print), 2016.

[9] B. Fortz, O. Oliveira, and C. Requejo. Compact mixed integer linear programming models to the minimum weighted tree reconstruction problem. *European Journal of Operations Research*, 256:242–251, 2017.

[10] O. Gascuel and M. Steel. Neighbor-joining revealed. *Molecular Biology and Evolution*, 23:1997–2000, 2006.

[11] J. M. Gómez, M. Verdú, and F. Perfectti. Ecological interactions are evolutionarily conserved across the entire tree of life. *Nature*, 465:918–921, 2010.

[12] D. Haws, T. Hodge, and R. Yoshida. Optimality of the neighbor joining algorithm and faces of the balanced minimum evolution polytope. *Bulletin of Mathematical Biology*, 73(11):2627–2648, 2011.

[13] J. Moreira, E. Miguez, C. Vilachá, and A. Otero. A parallel branch and bound approach to optimal power flow with discrete variables. *Przeglad Elektrotechniczny*, 89(3):47–52, 2013.

[14] Y. Pauplin. Direct calculation of a tree length using a distance matrix. *Journal of Molecular Evolution*, 51(1):41–47, 2000.

[15] D. Robinson and L. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53:131–147, 1981.

[16] W. Rytter. Trees with minimum weighted path length. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, chapter 10, pages 1–22. Chapman and Hall/CRC, 2004.

[17] N. Saitou and M. Nei. The neighbor joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4:406–425, 1987.

[18] J. Xi, J. Xie, R. Yoshida, and S. Forcey. Stochastic safety radius on neighbor-joining method and balanced minimal evolution on small trees. *arXiv:1507.08734*, (pre-print), 2015.

APPENDIX

Listing A.1: MATLAB BNB Test Data

```
clear all
close all

n = 9;  %Number of taxa
[A,b,Aeq,beq,lb,ub] = BMEineq(n); %Generate BME(n)
k = nchoosek(n,2); %Number of decision variables
M=1:k; %Number of variables required to be discrete
e=1e-4; %Tolerance parameter
maxiteration = 15000; %Maximum number of iterations
noise = 1; %Determines if we want to add noise

%% Cases
%Provide a true solution and distance data for experiments

%n = 6:
    %Test: N6T1

%      xtrue = [2 4 4 4 2 1 1 4 8 8 2 1 2 1 4];
%
%      d = [4 3 3 3 4 5 5 3 2 2 4 5 4 5 3];

    %Test: N6T2

%      xtrue = [8 1 4 2 1 1 4 2 1 2 4 8 4 2 4];
%
%      d = [2 14 3 13 14 14 3 13 14 13 3 2 12 13 3];


%n = 7:
    %Test: N7T1

%      xtrue = [1 1 2 16 4 8 16 8 1 4 2 8 1 4 2 2 8 4 4 8 8];
%
```

```
%       d = [17  18  5  3  5  3  3  14  18  16  16  15  19  17  17  6  4  4  6  4
    4];

     %Test:  N7T2

%       xtrue  =  [4  8  8  4  4  4  2  2  4  4  16  16  2  2  2  2  2  2  16  4  4];
%
%       d = [11  7  7  13  14  11  10  10  12  13  8  2  12  13  10  12  13  10
    3  12  13];


%n = 8:
     %Test:  N8T1

%       xtrue  =  [8  8  16  8  4  16  4  32  4  2  1  16  1  4  2  1  16  1  16  8
    8  8  16  4  ...
%       16  2  32  2];
%
%       d = [4  4  3  4  5  3  5  2  5  6  7  3  5  5  6  7  3  7  3  4  4  4  3  5  3
    6  2  6];

     %Test:  N8T2

%       xtrue  =  [32  16  4  4  4  4  4  16  4  4  4  2  2  8  8  8  4  4  32  8  4
    4  8  4  4  ...
%       16  16  32];
%
%       d = [5  9  15  16  15  18  19  8  14  15  14  17  18  10  11  10  13  14
     9  12  15  16  ...
%       13  16  17  9  10  9];


%n = 9:
     %Test:  N9T1

      xtrue  =  [16  8  32  64  4  2  1  1  32  32  16  16  8  4  4  16  8  32  16
          8  8  32  8  4  2  2 ...
      4  2  1  1  32  16  16  32  32  64];

      d = [4  5  3  2  6  7  8  8  3  3  4  4  5  6  6  4  5  3  4  5  5  3  5  6  7  7
          6  7  8  8  3  4  4 ...
      3  3  2];

    %Test: N9T2

%     xtrue = [32 8 8 8 4 2 64 2 16 16 16 8 4 32 4 16 64 8 4
    8 4 16 32 16 ...
%     8 16 8 4 8 4 32 4 32 2 64 2];
%
%     d = [4 8 9 9 11 16 4 13 6 7 7 9 14 4 11 5 3 7 12 8 9 6
    4 9 9 6 ...
%     8 13 9 10 9 11 6 16 5 13];


%n = 10:
    %Test: N10T1

%     xtrue = [128 64 32 16 8 4 2 1 1 64 32 16 8 4 2 1 1 64
    32 16 8 4 2 2 ...
%     64 32 16 8 4 4 64 32 16 8 8 64 32 16 16 64 32 32 64 64
    128];
%
%     d = [2 3 4 5 6 7 8 9 9 3 4 5 6 7 8 9 9 3 4 5 6 7 8 8 3
    4 5 6 7 7 ...
%     3 4 5 6 6 3 4 5 5 3 4 4 3 3 2];

    %Test: N10T2

%     xtrue = [64 128 8 32 8 8 2 4 2 64 16 64 16 16 4 8 4 8
    32 8 8 2 4 2 ...
%     32 128 32 8 16 8 32 32 8 16 8 32 8 16 8 32 64 32 64 128
     64];
%
%     d = [3 3 11 9 11 12 16 15 17 4 10 8 10 11 15 14 16 ...
%     12 10 12 13 17 16 18 6 2 5 9 8 10 6 7 11 10 12 5 9 8 10
     8 7 9 3 3 4];


%% Solve the BME Problem and calculate topological distances
if noise == 0 %No noise added, so we have perfect data

    %Set aside a pool of 2 cores
    %parpool(2)

    spmd
    %Algorithm 2 (heuristic = 1)
    t2s = cputime;
    [x2,val2,status2]=DILP1(d,A,b,Aeq,beq,lb,ub,M,e,
        maxiteration,1);
    t2f = cputime - t2s;
    RF2 = RFmetric(x2,xtrue,n);

    %Algorithm 1 (heuristic = 0)
    t1s = cputime;
    [x1,val1,status1]=DILP1(d,A,b,Aeq,beq,lb,ub,M,e,
        maxiteration,0);
    t1f = cputime - t1s;
    RF1 = RFmetric(x1,xtrue,n);

    %l = distance(x,n); %if we want to draw the graph
    end

%% Solve the same problem, but add some noise. Do this if
    noise = 1
else %noise == 1

    %Create the distribution
    mu = 0; %Mean
    sigma = 1; %Standard Deviation
    h = 0.4; %Scales the perturbations

    delta = h*normrnd(mu,sigma,1,k); %Construct the
        perturbation vector

    L_inf = max(abs(delta)); %How much noise we add

    dpert = d + delta; %Perturb the objective function

    %Set aside a pool of 2 cores
    %parpool(2)

    spmd
    %Algorithm 2 (heuristic = 1)
    t2s = cputime;
    [x2pert,val2pert,pert2_status]=DILP1(dpert,A,b,Aeq,beq,lb
        ,ub,M,e,maxiteration,1);
    t2f = cputime - t2s;
    RF2 = RFmetric(x2pert,xtrue,n);


    %Algorithm 1 (heuristic = 0)
    t1s = cputime;
    [x1pert, val1pert, pert1_status]=DILP1(dpert,A,b,Aeq,beq,lb
        ,ub,M,e,maxiteration,0);
    t1f = cputime - t1s;
    RF1 = RFmetric(x1pert,xtrue,n);
    end
    %l_pert = distance(xpert,n); %if we want to draw the
        graph
end
```

Listing A.2: MATLAB Code for Generating BME(n)

```
function [A,b,Aeq,beq,lb,ub] = BMEineq(n)
%This function determines the Matrix of vector inequalities
%used for the BME Problem. Here, LHS = A, RHS = b,
%lb is the lower bound on the x values for the problem, while
%ub is the upper bound on the x values.

%Initialize matrix to store constraints
k = nchoosek(n,2);
r = k+k*(n-2)+2*n; %+number of splits
A = zeros(r,k);
b = zeros(r,1);
Aeq = zeros(n,k); %What's the size of Aeq?
beq = zeros(n,1); %Size of beq?

%Create a vector for the lb on index:
lb = ones(k,1);

%Create a vector for the ub on index:
ub = zeros(k,1);
for jj=1:k
    ub(jj) = 2^(n-3);
end

%Intersecting Cherry Facets: x_ik + x_jk - x_ij <= 2^(n-2)
%Start a counter to index the row
t = 1;
for i = 1:n-1
    for j = 1+i:n
        for s = 1:n
            if i~=s && s~=j
                A(t,min(i,j)*(2*n-1-min(i,j))/2-n+max(i,j)) =
                    -1;
                A(t,min(i,s)*(2*n-1-min(i,s))/2-n+max(i,s)) =
                    1;
                A(t,min(s,j)*(2*n-1-min(s,j))/2-n+max(s,j)) =
                    1;
                b(t,1) = 2^(n-3);
                t = t + 1; %Increment t
            end
        end
    end
end
```

```

%Split Facets:sum(x_ij), where i,j are in S1 and i < j.
 if n>5  %Condition for a split
     fl = floor(n/2);
     row = 0;
     for ii = 3:fl
         row = row + nchoosek(n,ii);
     end
     col = fl;
     C = zeros(row,col); %Preallocates matrix
     x=1:n; %"Leaves"
     for nn = 3:fl %Sizes of subsets
         %C = combnk(x,nn); %Generates a matrix of subsets for splits
         C1 = combnk(x,nn); %Generates a matrix of subsets for splits
         [rowC1,colC1] = size(C1);
         %Begin the transfer to C
         if nn == 3
             C(1:rowC1,1:3) = C1(1:rowC1,1:3);
         else %nn>3 and we need to tack onto the end of the matrix
             C(all(~C,2),:) = [];
             [rC,cC] = size(C);  %Returns the current size of C
             C(rC+1:rC + rowC1,1:nn) = C1(1:rowC1,1:nn);
         end
     end

     %Generate the rows of A from the matrix of subsets
     d = 1; %row counter
     for rr = 1:row
         nonz = nonzeros(C(rr,:)); %counts {|non-zero entries|}
         m = length(nonz);
         if mod(n,2)~=0 %n is odd, we want all size
         %floor(n/2) subsets
             %Enter as a split:
                 for i = 1:n-1
                     for j=i+1:n
                         R1o = ismember(i,C(rr,:)); %checks for i in C
```

```
                       R2o = ismember(j,C(rr,:)); %checks for j in C
                       if R1o == 1 && R2o == 1
                           A(k+k*(n-2)+2*n+d,i*(2*n-1-i)/2-n+j) = 1;
                           b(k+k*(n-2)+2*n+d,1) = (m-1)*2^(n-3);
                       end
                   end
               end
               d = d + 1; %Increment d for next row entry
        else %n is even so subsets will be slightly different
              if m < n/2
                   for i = 1:n-1
                       for j=i+1:n
                           R1e = ismember(i,C(rr,:));
                           R2e = ismember(j,C(rr,:));
                           if R1e == 1 && R2e == 1
                               A(k+k*(n-2)+2*n+d,i*(2*n-1-i)/2-n+j) = 1;
                               b(k+k*(n-2)+2*n+d,1) = (m-1)*2^(n-3);
                           end
                       end
                   end
                   d = d + 1; %Increment d for next entry
              else %m = n/2
                  for i = 1:n-1
                       for j=i+1:n
                           R1ef = ismember(1,C(rr,:));
                           R2ef = ismember(i,C(rr,:));
                           R3ef = ismember(j,C(rr,:));
                           if R1ef == 1 && R2ef == 1 && R3ef == 1
                               A(k+k*(n-2)+2*n+d,i*(2*n-1-i)/2-n+j) = 1;
                               b(k+k*(n-2)+2*n+d,1) = (m-1)*2^(n-3);
                           end
                       end
                  end
                  d = d + 1; %Increment d for next entry
```

```
                    end
                end
            end
        end

%Kraft Equalities
%sum(x_ij) = 2^(n-2)
%Since these are equalities, put them into Aeq and beq
for i = 1:n
    for j = 1:n
        if i~=j
        Aeq(i,min(i,j)*(2*n-1-min(i,j))/2-n+max(i,j)) = 1;
        end
    end
    beq(i,1) = 2^(n-2);
end
%Clean up and filter out zero rows in A and b
     A(all(~A,2),:) = [];
     b(all(~b,2),:) = [];
     Aeq(all(~Aeq,2),:) = [];
     beq(all(~beq,2),:) = [];
end
```

Listing A.3: MATLAB Code for Discrete ILP

```
function [x,val,status] = DILP1(f,A,b,Aeq,beq,lb,ub,M,e,
    maxiteration,heuristic)
%This function solves a discrete-integer linear programming
    problem
%using the branch and bound algorithm.
%The code uses MATLAB's linear programming solver "linprog"
%to solve the LP relaxations at each node of the branch and
    bound tree.
%   min f*x
%  subject to
%        A*x <= b
%        Aeq * x = beq
%        lb <= x <= ub
%        M is a vector of indices for the discrete variables
%        e is the tolerance
%The output variables are:
% x : the solution
% val: value of the objective function at the optimal
    solution
% status =1 if successful
%        =0 if maximum number of iterations reached in the
    linprog function
%        =-1 if there is no solution
%In order to run this code, you will need the contraints from
     BMEineq.m
%and BNBtest.m.
%Author: William Sands
%This code was adapted from code provided by Kartik
    Sivaramakrishnan to solve a particular discrete-integer
    linear programming problem.

%%
global count
global maxiter0
global maxiter1
h = heuristic; %This determines the rounding scheme for the
    branching
count=2;

if h == 0
     maxiter0 = maxiteration; %We don't round any of the
         variables
```

```
elseif h == 1
    maxiter0 = maxiteration; %We round only one variable at a time
    maxiter1= 2*maxiteration;
else
    fprintf('error: heuristic must equal 0 or 1.')
    return
end

options = optimset('display','off');
options.Algorithm = 'dual-simplex';
options.ConstraintTolerance = '1e-10';

bound = inf; %The initial bound is set to +ve infinity
%Solve the LP relaxation at the root node using MATLAB's linprog function
%Type "help linprog" for help with the linprog routine

%Solve the initial LP solution to determine feasiblility
[x0,val0,exitflag] = linprog(f,A,b,Aeq,beq,lb,ub,[],options);

%If the LP is infeasible, then don't branch
if exitflag <= 0
    x = [];
    val = [];
    status = exitflag;
    return
end

if h == 1  %Determine number of cherries
    cherries = find(abs(x0(M)-ub(M)) <= e);
    newub = find(abs(x0(M)-ub(M)) > e);
    [row,~] = size(cherries);
    if row >=2
        %Initialize Aeq and Beq.
            [rq,cq] = size(Aeq);
            Aeq = [Aeq;zeros(row,cq)]; %Same column size as A
            beq = [beq;zeros(row,1)];
           %Set the cherries as equalities in Aeq and beq
            for rr = 1:row
                Aeq(rq+rr,cherries(rr)) = 1;
                beq(rq+rr,1) = ub(cherries(rr));
            end
            %Reset the upper bounds since we found 'all' of the cherries
            ub(newub) = ub(cherries(1))-0.5*ub(cherries(1)); %Set new ub
            [x,val,status,b] = branch1(f,A,b,Aeq,beq,lb,ub,x0,val0,M,e,bound);

    else %Don't fix the cherries. Cherry cannot be forced.
        [x,val,status,b] = branch1(f,A,b,Aeq,beq,lb,ub,x0,val0,M,e,bound);
    end

else %Don't use the rounding heuristic
    [x,val,status,b] = branch0(f,A,b,Aeq,beq,lb,ub,x0,val0,M,e,bound);
end
end
```

Listing A.4: MATLAB Code for branch0

```
function [xx,val,status,bb] = branch0(f,A,b,Aeq,beq,lb,ub,x,v
    ,M,e,bound)
global count
global maxiter0

options.Display = 'off';
options.Algorithm = 'dual-simplex';
options.ConstraintTolerance = '1e-8';

%Solve the LP relaxation at the current node
[x0,val0,status0] = linprog(f,A,b,Aeq,beq,lb,ub,[],options);

%If the LP relaxation is infeasible,then PRUNE THE NODE BY
    INFEASIBILITY
%If the new objective value is worse, then prune by
    optimality
if status0 <= 0 || val0 > bound
     %Return the input to linprog
     xx = x;
     val = v;
     status = status0;
     bb = bound;
     return;
end

%%
%If the solution to the LP relaxation is feasible in the DILP
     problem, then check the objective value of this
%against the objective value of the best feasible/valid
    solution that has been obtained so far for the DILP
    problem.
%If the new feasible/valid solution has a lower objective
    value then update the bound
%Else PRUNE THE NODE BY OPTIMALITY

%Calculate tolerance and find branching variables.
[E,ind] = max(min(abs(x0(M)-2.^floor(log2(x0(M)))),abs(x0(M)
    -2.^ceil(log2(x0(M))))));

if E < e || count > maxiter0 %If we have a valid solution or
    exceed maxiter
     status = 1;
    if val0 < bound %The new feasible solution is an
        improvement
         xx = x0;
         val = val0;
         bb = val0;
    else
         xx = x;  %Return the input solution and most recent
             bounds
         val = v;
         bb = bound;
    end
    return
end

%%
%If we come here this means that the solution of the LP
    relaxation is not valid in the DILP problem.
%However, the objective value of the LP relaxation is lower
    than the current bound.
%So we branch on this node to create two subproblems.
%We will solve the two subproblems recursively by calling the
     same branching function.

%Select the branching variable.
br_var = M(ind(1));
br_value = x0(br_var);
[~,c] = size(A);

%First LP problem with the added constraint that x_i <= floor
    (x_i),i=ind(1)
A1 = [A ; zeros(1,c)];
A1(end,br_var) = 1;
b1 = [b;2^floor(log2(br_value))];

%Second LP problem with the added constraint that x_i >= ceil
    (x_i),i=ind(1)
A2 = [A ;zeros(1,c)];
A2(end,br_var) = -1;
b2 = [b; -2^ceil(log2(br_value))];

%%
%Solve the first LP problem
count = count + 1; %+One for each subproblem being created
```

```
[x1,val1,status1,bound1] = branch0(f,A1,b1,Aeq,beq,lb,ub,x0,
    val0,M,e,bound);
status = status1;
if status1 > 0 && bound1 < bound %If the solution was
    successfull and gives a better bound
     xx = x1;
     val = val1;
     bound = bound1;
     bb = bound1;
else
     xx = x0;
     val = val0;
     bb = bound;
end

%Solve the second LP problem
count = count + 1; %+One for each subproblem being created
[x2,val2,status2,bound2] = branch0(f,A2,b2,Aeq,beq,lb,ub,x0,
    val0,M,e,bound);
if status2 > 0 && bound2 < bound %If the solution was
    successful and gives a better bound
     status = status2;
     xx = x2;
     val = val2;
     bb = bound2;
end
end
```

Listing A.5: MATLAB Code for branch1

```
function [xx,val,status,bb] = branch1(f,A,b,Aeq,beq,lb,ub,x,v,M,e,bound)
global count
global maxiter0
global maxiter1

options.Display = 'off';
options.Algorithm = 'dual-simplex';
options.ConstraintTolerance = '1e-8';

%Solve the LP relaxation at the current node
[x0,val0,status0] = linprog(f,A,b,Aeq,beq,lb,ub,[],options);

%If the LP relaxation is infeasible,then PRUNE THE NODE BY INFEASIBILITY
%If the new objective value is worse, then prune by optimality
if status0 <= 0 || val0 > bound
     %Return the input to linprog
     xx = x;
     val = v;
     status = status0;
     bb = bound;
     return;
end

%%
%If the solution to the LP relaxation is feasible in the DILP problem, then check the objective value of this
%against the objective value of the best feasible/valid solution that has been obtained so far for the DILP problem.
%If the new feasible/valid solution has a lower objective value then update the bound
%Else PRUNE THE NODE BY OPTIMALITY

%Calculate tolerance and find branching variables.
[E,ind]= max(min(abs(x0(M)-2.^floor(log2(x0(M)))),abs(x0(M)-2.^ceil(log2(x0(M))))));

if E < e || count > maxiter1 %If we have a valid solution or exceed maxiter
```

```
    status = 1;
    if val0 < bound %The new feasible solution is an
       improvement
        xx = x0;
        val = val0;
        bb = val0;
    else
        xx = x;  %Return the input solution and most recent
           bounds
        val = v;
        bb = bound;
    end
    return
end

%%
%If we come here this means that the solution of the LP
   relaxation is not valid in the DILP problem.
%However, the objective value of the LP relaxation is lower
   than the current bound.
%So we branch on this node to create two subproblems.
%We will solve the two subproblems recursively by calling the
    same branching function.

%Select the branching variable.
br_var = M(ind(1));
br_value = x0(br_var);
[req,ceq] = size(Aeq);
[~,c] = size(A);

%Use the rounding heuristic (optional) to check for entries
   arbitrarily close to their
%discrete values. Don't set equalities for variables that are
    already powers of 2.

%We are using a heuristic
nonz = find(abs(x(M)-2.^round(log2(x0(M)))) > 0); %nonpower
   of 2 entries

if count > maxiter0
   cRind = find(min(abs(x0(nonz)-2.^round(log2(x0(nonz))))) <
       e); %Smallest within tol
end
```

```

if count > maxiter0
    if ~isempty(cRind) %If a variable can be rounded
        Aeq = [Aeq; zeros(1,ceq)];
        beq = [beq;zeros(1)];
        Aeq(req+1,cRind(1)) = 1;
        beq(req+1) = 2^round(log2(x0(cRind(1)))); %Round that
            variable
    end
end

%First LP problem with the added constraint that x_i <= floor
    (x_i),i = ind(1)
A1 = [A ; zeros(1,c)];
A1(end,br_var) = 1;
b1 = [b;2^floor(log2(br_value))];

%Second LP problem with the added constraint that x_i >= ceil
    (x_i),i=ind(1)
A2 = [A ;zeros(1,c)];
A2(end,br_var) = -1;
b2 = [b; -2^ceil(log2(br_value))];

%%
%Solve the first LP problem
count = count+1; %+One for each subproblem being created
[x1,val1,status1,bound1] = branch1(f,A1,b1,Aeq,beq,lb,ub,x0,
    val0,M,e,bound);
status = status1;
if status1 > 0 && bound1 < bound %If the solution was
    successfull and gives a better bound
    xx = x1;
    val = val1;
    bound = bound1;
    bb = bound1;
else
    xx = x0;
    val = val0;
    bb = bound;
end

%Solve the second LP problem
count = count + 1; %+One for each subproblem being created
```

```
[x2,val2,status2,bound2] = branch1(f,A2,b2,Aeq,beq,lb,ub,x0,
   val0,M,e,bound);
if status2 > 0 && bound2 < bound %If the solution was
   successful and gives a better bound
    status = status2;
    xx = x2;
    val = val2;
    bb = bound2;
end
end
```

Listing A.6: MATLAB Code to compute the Robinson-Foulds Metric

```
function [total] = RFmetric(x1,x2,n)
%This function computes the Robinson-Foulds distance using
   characteristic
%functions to determine differences in the splits. It takes
   vector valued
%input as well as the number of taxa in the trees. Here, x1
   and x2
%represent two trees.

%%

if n < 4
   fprintf = ('Error. The number of taxa must be at least 4.'
      );
   return
end

%Initialize the sum for the RF metric

total = 0;

%% Generate the set of partitions

fl = floor(n/2);
    row = 0;
    for ii = 2:fl %Want sets of size 2 to floor(n/2)
        row = row + nchoosek(n,ii); %Calculates how many rows
            are needed
    end
    col = fl; %Columns needed
    C = zeros(row,col); %Preallocates matrix for partitions
    x = 1:n; %"Leaves"
    for nn = 2:fl %Sizes of subsets

        C1 = combnk(x,nn); %Generates a matrix of subsets for
            splits
        [rowC1,~] = size(C1); %Find dimensions of C1

        %Begin the transfer to C
        if nn == 2
            C(1:rowC1,1:2) = C1(1:rowC1,1:2);
        else %nn > 3 and we need to tack onto the end of the
           matrix
            C(all(~C,2),:) = [];
            [rC,cC] = size(C);  %Returns the current size of
               C
            C(rC+1:rC + rowC1,1:nn) = C1(1:rowC1,1:nn);
        end
    end

%% Check to see if splits are the same

%Sum over the splits

for rr = 1:row
    nonz = nonzeros(C(rr,:)); %Counts {|non-zero entries|}
    m = length(nonz);
    if mod(n,2)~=0 %n is odd, we want all size floor(n/2)
       subsets
        %Enter as a split:
        %Initialize a local sum to change for each row of C
        sum1 = 0;
        sum2 = 0;

            for i = 1:n-1
                for j=i+1:n

                    R1o = ismember(i,C(rr,:)); %Checks for i
                       in C
                    R2o = ismember(j,C(rr,:)); %Checks for j
                       in C

                    if R1o == 1 && R2o == 1 %It is in the
                       split

                        %Update the sums
                        sum1 = sum1 + x1(i*(2*n-1-i)/2-n+j);
                        sum2 = sum2 + x2(i*(2*n-1-i)/2-n+j);

                    end
                end
            end

        %Now compute the characteristic function for this
           split

        if (sum1 == (m-1)*2^(n-3) && sum2 ~= (m-1)*2^(n-3))
           || ...
           (sum2 == (m-1)*2^(n-3) && sum1 ~= (m-1)*2^(n-3))

            total = total + 1; %Characteristic is 1, so
               update the sum

        else

            total = total;

        end

     else %n is even so subsets will be slightly different
     %Enter as a split:
     %Initialize a local sum to change for each row of C
     sum1 = 0;
     sum2 = 0;

           if m < n/2
                for i = 1:n-1
                    for j=i+1:n
                        R1e = ismember(i,C(rr,:));
                        R2e = ismember(j,C(rr,:));
                        if R1e == 1 && R2e == 1

                            %Update the sums
                            sum1 = sum1 + x1(i*(2*n-1-i)/2-n+
                               j);
                            sum2 = sum2 + x2(i*(2*n-1-i)/2-n+
                               j);

                        end
                    end
                end

           else %m = n/2
               for i = 1:n-1
                   for j=i+1:n
                       R1ef = ismember(1,C(rr,:));
```

```
                        R2ef = ismember(i,C(rr,:));
                        R3ef = ismember(j,C(rr,:));
                        if R1ef == 1 && R2ef == 1 && R3ef ==
                           1

                            %Update the sums
                            sum1 = sum1 + x1(i*(2*n-1-i)/2-n+
                               j);
                            sum2 = sum2 + x2(i*(2*n-1-i)/2-n+
                               j);

                        end
                    end
               end

                      %Now compute the characteristic
                         function for this split

               if (sum1 == (m-1)*2^(n-3) && sum2 ~= (m-1)*2^(
                  n-3)) || ...
                    (sum2 == (m-1)*2^(n-3) && sum1 ~= (m-1)
                       *2^(n-3))

                       total = total + 1; %Characteristic is
                          1, so update the sum

               else

                       total = total;

               end

            end
    end
end

end
```

Listing A.7: MATLAB Code to obtain distances l_{ij}

```
function [l] = distance(x,n)
%This function takes the solution returned by the DILP.m file
    and converts the entries in the vector from pauplin
   coordinates to real distances
%using the formula: l_ij = n-2-log2(x_ij) for all entries

num_entries = length(x);
l = zeros(num_entries,1);
    for i = 1:num_entries
        l(i) = n-2-log2(x(i));
    end
end
```