Structure and Applications of Totally Decomposable Metrics//Doctoral Thesis, GSIA, Carnegie Mellon

George Christopher

May, 1997

Chapter 1

Introduction

1.1 Overview

Many interesting questions in operations research can be formulated as optimization problems on a complete graph G. Each edge in the graph is assigned a weight $D[i, j]$ and these weights are stored as an $n \times n$ matrix. To solve many of these problems, researchers must find a minimum (maximum) weight set of edges satisfying a set of structural conditions. Well known examples of these questions include the minimum cost spanning tree problem, the maximum weight b-matching problem and the traveling salesman problem.

Over the last 50 years, a great deal of effort has been devoted to finding efficient ways to solve these problems. Although efficient algorithms have been found for many problems, a large framework has been developed to classify those problems for which no efficient algorithm is known. Unfortunately, many combinatorial optimization problems fall into this framework. Even worse, many algorithms which are efficient from a theoretical point of view are only practical when G has relatively few vertices.

Faced with these problems, researchers are often forced to design algorithms that only work when D satisfies a set of structural constraints. Many of these constraints are quite natural. For example, when D represents an amount of time, non-negativity constraints are placed on the entries of D. When D represents a physical distance, its entries are non-negative and satisfy the triangle inequality. Most algorithms implicitly place some restriction on D.

Since it is widely believed that no polynomial time algorithm exists for solving any NP-complete problem, many algorithms have been developed to solve special cases of NP-complete problems. For example many algorithms exist for solving special cases of the traveling salesman problem. For an overview of these special cases, see Lawler et al. [30] or Burkard et al. [18].

Researchers also design algorithms that exploit the structure of a metric to speed up polynomial time algorithms. For example, consider the development of single source shortest path algorithms. The Bellman-Ford algorithm developed from Bellman's 1958 paper [12]. This algorithm finds shortest paths from a fixed source to all vertices in a graph with no negative weight cycles in $O(nm)$ time where n is the number of vertices in the graph and m is the number of edges. Dijkstra's single source shortest path algorithm was published in 1959. Although Dijkstra's algorithm will only work when D contains no negative entries, it can solve the single source shortest paths problem in $O(n \log n + m)$ time.

When dealing with special case algorithms, there is always a compromise involved. The speed of the algorithm usually depends on the strength of the restrictions placed on D: the stronger the restrictions, the faster the algorithm. Unfortunately, real world problem instances do not always meet the stringent restrictions of special case algorithms. This leaves the researcher with two options, run a much slower algorithm to solve the problem, or consider how to relax the assumptions while preserving some of the fast algorithm's time savings.

As understanding of a problem grows, a hierarchy of classes of cost functions develops. These hierarchies typically contain many small, highly restricted classes of cost functions associated with blazingly fast algorithms, and larger, less restricted classes associated with slightly slower algorithms. The restrictions placed on these classes can often be exploited to solve other problems. This makes the hierarchies interesting in their own right. This thesis considers one hierarchy of weight functions, totally decomposable metrics.

The rest of this chapter introduces background concepts needed in the discussion of the totally decomposable hierarchy of metrics. In chapter 2 various classes in the hierarchy are defined, examples of each class are given, and relationships between the classes are discussed.

In chapters 5 and 6, algorithms are developed which exploit the structure of metrics in our hierarchy. In most cases, the discussion of an algorithm is followed by pseudo-code for the algorithm and an example. Chapter 5 explains how to exploit the structure of totally decomposable metrics when solving the maximum weight b-matching problem. Chapter 6 discusses the hierarchy's relationship with the traveling salesman problem.

Although a special case algorithm may be more efficient that a general algorithm, there are some drawbacks. If the special case algorithm is used when D does not have the necessary structure, it may not work as expected. When a special case algorithm runs on a metric without the appropriate structure, it may return a suboptimal solution. To avoid this problem, either the algorithm or the researcher using it must verify that D satisfies the restrictions placed upon it. Chapters 3 and 4 address these issues. Chapter 3 discusses the structure of Circular Decomposable Metrics and chapter 4 presents recognition algorithms for the classes in the totally decomposable hierarchy.

The remainder of this chapter introduces the most basic concepts underlying this thesis. While many of these are standard ideas in combinatorial optimization, some non-standard notation and conventions are developed here. The reader is encouraged to at least skim this section.

1.2 Definitions and Conventions

1.2.1 Metrics and Matrices

A symmetric $n \times n$ matrix D is called a **metric** if the following hold for each distinct $i, j, k \in V$.

$$D[i,j] + D[j,k] \geq D[i,k] \tag{1.1}$$
$$D[i,j] \geq 0 \tag{1.2}$$
$$D[i,i] = 0 \tag{1.3}$$

Astute readers may have notice that these conditions actually define pseudo metrics (since $D[i,j]$ may be 0 even though $i \neq j$). In order to simplify the discussion in the following chapters, both pseudo-metrics and metrics will be referred to as metrics. Unless otherwise noted, we will assume that all matrices are symmetric and have zeros on the main diagonal.

For convenience, we denote the set of indices of rows and columns of D by $S_D = \{1, 2, ..., n\}$. When no confusion will arise, the subscript may be deleted from S.

A specific metric that we refer to often is the **zero metric**. The zero metric is the metric D with $D[i,j] = 0$ for all $i, j \in S$.

1.2.2 Cuts and Cut Metrics

One of the most important ideas used in this paper is the idea of a cut.

Definition 1.2.1 *A* **cut** *of a set* $S = \{1, ..., n\}$ *is a pair of subsets* $(C, \overline{C})$ *such that*

$$\begin{array}{rcl} C \cap \overline{C} & = & \emptyset \\ C \cup \overline{C} & = & S \end{array}$$

C and $\overline{C}$ are called the **shores** *of the cut. When S is the set of indices of rows and columns of a metric D, we will say that $(C, \overline{C})$ is a cut of D.*

Definition 1.2.2 *The cut $(C, \overline{C})$ is said to* **separate** *distinct indices $i, j \in S$ if, $|\{i, j\} \cap C| = 1$. $(C, \overline{C})$ is said to separate disjoint subsets S_1 and S_2 of S if it separates i and j for all $i \in S_1$ and $j \in S_2$.*

Notice the symmetry between C and $\overline{C}$ in definition 1.2.1. In the following chapters, we will exploit this symmetry by placing restrictions on C, such as requiring that $|C| \leq |\overline{C}|$ or forcing a particular node l to be in C. These restrictions are introduced to simplify the proofs of our theorems and are always chosen so that any arbitrary cut $(C, \overline{C})$ which does not satisfy them can be modified to satisfy them by exchanging the roles of C and $\overline{C}$.

The classes of metrics considered in chapter 2 can all be defined in terms of cuts and the algorithms in later chapters are designed to exploit the structure of and interaction between cuts. The definition of a cut metric translates the structure of a cut into the notation of matrices.

Definition 1.2.3 *The* **cut metric** *D_C defined by the cut $(C, \overline{C})$ on $S = \{1, ..., n\}$ is the metric satisfying*

$$\begin{array}{rcll} D_C[i, j] & = & 1 & \textit{If } (C, \overline{C}) \textit{ separates } i \textit{ and } j \\ D_C[i, j] & = & 0 & \textit{Otherwise} \end{array}$$

A weighted cut metric is a metric of the form αD_C for a positive scalar α and cut metric D_C.

Individual cut metrics are too structured to be interesting, so we consider metrics which can be written as the sum of a collection of weighted cut metrics.

Definition 1.2.4 *A metric D is called* **decomposable** *if there exists a collection $\mathcal{C}$ of weighted cut metrics such that*

$$D = \sum_{(C,\overline{C})\in\mathcal{C}} \alpha_C D_C$$

$\mathcal{C}$ is called **decomposition** *of D.*

Unfortunately, decomposable metrics can be extremely complicated. In order to reduce the complexity of the metrics we will study, we restrict the interaction of the members of $\mathcal{C}$. This is done in two ways:

Definition 1.2.5 *We say that the two cuts $(C, \overline{C})$ and $(B, \overline{B})$ are* **laminar** *when at least one of the four following conditions holds*

$$\begin{array}{rclcrcl} C \cap B & = & \emptyset & \quad & C \cap \overline{B} & = & \emptyset \\ \overline{C} \cap B & = & \emptyset & \quad & \overline{C} \cap \overline{B} & = & \emptyset \end{array}$$

A collection of cuts $\mathcal{C}$ is called **laminar** *if every pair of cuts in $\mathcal{C}$ is laminar.*

In Figure 1.1 the two cuts pictured on the left are not laminar, while the two on the right are.

Definition 1.2.6 *A collection of cuts $\mathcal{C}$ is called* **weakly compatible** *if there do not exist distinct cuts $\{(C_1, \overline{C}_1), (C_2, \overline{C}_2), (C_3, \overline{C}_3)\} \in \mathcal{C}$ and distinct indices a, b, c, d of S such that*

$$\begin{array}{l} (C_1, \overline{C}_1) \text{ separates } \{a, b\} \text{ and } \{c, d\} \\ (C_2, \overline{C}_2) \text{ separates } \{a, c\} \text{ and } \{b, d\} \\ (C_3, \overline{C}_3) \text{ separates } \{a, d\} \text{ and } \{b, c\} \end{array}$$

See Figure 1.2 for the simplest example of a collection of non-weakly compatible cuts.

Given a laminar or weakly compatible collection of cuts $\mathcal{C}$, we will often consider when a cut $(C, \overline{C})$ can be added to $\mathcal{C}$ without effecting the laminarity or weak compatibility of the $\mathcal{C}$. There is a special class of cuts known as trivial cuts which can be added to any weakly compatible or laminar collection of cuts without destroying the collections weak compatibility or laminarity.

Definition 1.2.7 *Cut metrics which separate a unique $i \in S$ from every other $j \in S$ are called* **trivial cuts**.

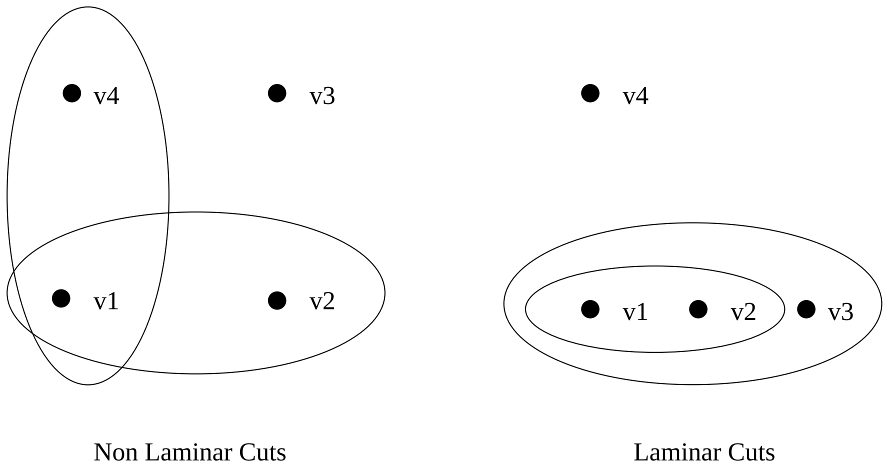


Figure 1.1:

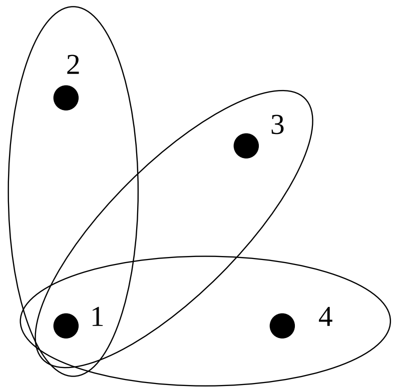


Figure 1.2: A Non-Weakly Compatible Collection of Cuts

1.2.3 Induced Metrics and Cuts

In later chapters we will be interested in what happens when a set of rows and columns is deleted from D. This idea is formalized in the following definition.

Definition 1.2.8 *Given a metric D and a subset $T = \{t_1, ..., t_m\}$ of S, define the $m \times m$ metric D^T according to the rule $D^T[i,j] = D[t_i, t_j]$ for all $1 \leq i, j \leq m$. D^T is called the* **sub-metric of D induced by T**.

In order to study the relationship between the decomposition of D and the decomposition of D^T, we define induced cuts.

Definition 1.2.9 *Let $(C, \overline{C})$ be a cut of a metric D and let $T \subset S$. If $C^T = C \cap T$ and $\overline{C^T} = \overline{C} \cap T$ are both non empty, then we call $(C^T, \overline{C^T})$* **the cut of D^T induced by** $(C, \overline{C})$.

It is important to notice that distinct cuts $(C_1, \overline{C}_1)$ and $(C_2, \overline{C}_2)$ may induce the same cut $(C^T, \overline{C^T})$ in D^T. For example if $S_D = \{1, 2, ..., 8\}$, and $T = \{1, 4, 5, 8\}$, the two cuts $(\{1, 3, 5, 7\}, \{2, 4, 6, 8\})$, and $(\{1, 2, 5, 6\}, \{3, 4, 7, 8\})$ both induce $(\{1, 5\}, \{4, 8\})$ in D^T.

Chapter 2

Totally Decomposable Metrics

The purpose of this chapter is to define and motivate the classes of metrics we will be using throughout this thesis. Section one motivates the definition of totally decomposable metrics (TDMs) by considering them as an extension of the well known class of tree metrics. In addition, we present definitions of other well known subclasses of TDMs, such as two-tree metrics and taxi metrics. Section 2 discusses circular decomposable metrics (CDMs), a subclass of TDMs defined by placing circularity conditions on the the cuts in the decomposition of the metric. As we continue, it will become clear that CDMs share more of the useful properties with tree metrics than do TDMs. These common properties will allow us design algorithms that run on CDMs by extending existing algorithms for tree metrics.

This chapter is only intended to provide a motivation for the definitions of the classes of metrics. A detailed discussion of CDMs is the topic of chapter 3. Recognition and decomposition algorithms for the classes are discussed in chapter 4.

2.1 Totally Decomposable Metrics

In order to motivate the definition of totally decomposable metrics, we first discuss a class of metrics known as tree metrics. Tree metrics are most easily defined in terms of weighted trees.

Definition 2.1.1 *A* **weighted tree** *is a tree T along with a weight function $c : E(T) \rightarrow R^+$*

$$\begin{bmatrix} 0 & 3 & 5 & 6 & 1 & 3 \\ 3 & 0 & 4 & 5 & 1 & 2 \\ 5 & 4 & 0 & 5 & 3 & 2 \\ 6 & 5 & 5 & 0 & 4 & 3 \\ 1 & 1 & 3 & 4 & 0 & 1 \\ 3 & 2 & 2 & 3 & 1 & 0 \end{bmatrix}$$

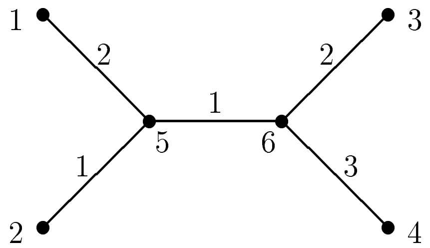


Figure 2.1: A tree metric

Definition 2.1.2 *A metric D is called a* **tree metric** *if there exists a weighted tree, T and a function mapping each $i \in S$ to a unique vertex $v_i \in T$ such that $D[i,j]$ is the sum of the costs of the edges in the unique v_i, v_j path in T.*

Notice that the definition of tree metrics does not require each vertex of T to be labeled. The distance matrix and tree representation of a tree metric are given in Figure 2.1.

Tree metrics occur naturally in a variety of classification problems. Phylogenetic trees have been used to model evolution, see Felsenstein [24], or Robinson and Foulds [36]. Historians have used trees to model the migrations of early cultures, see Buneman [16]. Tree metrics have also been used to define easily solvable special cases of several NP-complete problems such as the traveling salesman problem, see [18], and the k-minimum spanning tree problem, see Blum et al. [13]. Finally, Bartal [10] has used tree metrics to design randomized algorithms for several on line optimization problems.

Given the range of applications of tree metrics, it is natural that people have tried to generalize them. Bandelt and Dress [8] do this by defining totally decomposable metrics. To see how they did this, consider how a tree metric can be expressed as the sum of weighted cut metrics. Removing an edge of T breaks T into two trees with vertex sets C_e and $\overline{C_e}$. It is easy to check that

$$D[i,j] = \sum_{e \in T} w(e) D_{C_e}[i,j]$$

Bandelt and Dress [8] propose an interesting approach for finding this decomposition of D without knowing the tree. To illustrate their ideas, consider the tree metric D and its tree representation in Figure 2.1. The edge $e_{1,5}$, has weight 2 in T and its removal disconnects the tree into two trees with vertex sets $\{1\}$ and $T - \{1\}$. Using the notation of [8], choose any two (not necessarily distinct) vertices b and b' in $T - \{1\}$ and consider the expression

$$\frac{1}{2}(D[1, b] + D[1, b'] - D[b, b']) = \beta_{\{1\},\{bb'\}} \tag{2.1}$$

It is easy to check that no matter how b and b' are chosen $\beta_{\{1\},\{bb'\}} \geq D[1, 5] = 4$.

Now consider an arbitrary edge $e_{i,j}$ in T. Removal of $e_{i,j}$ breaks T into two trees T_1 and T_2 with vertex sets A and B respectively. Using the notation of Bandelt and Dress, define $\beta_{a,a',b,b'}$ and $\alpha_{A,B}$ as

$$\beta_{a,a',b,b'} = \max\{ab + a'b', ab' + ba', aa' + bb'\} \tag{2.2}$$

$$\min\{\frac{1}{2}(\beta_{a,a',b,b'} - aa' - bb')\} = \alpha_{A,B} \tag{2.3}$$

where ab is used as shorthand for $D[a, b]$ and the minimum is taken over all (not necessarily distinct) $a, a' \in A$ and $b, b' \in B$. $\alpha_{A,B}$ is defined for all pairs of disjoint subsets of S. When (A, B) is a cut of S, $\alpha_{A,B}$ is called the **isolation index** of (A, B). If the isolation index of $(C, \overline{C})$ is positive then $(C, \overline{C})$ is called a **d-split**.

Bandelt and Dress used these ideas to develop the following approach for finding the decomposition of a tree metric D. For each cut $(C, \overline{C})$ of D, compute its isolation index. Bandelt and Dress have shown that D is a tree metric exactly when the collection $\mathcal{C}$ of d-splits is laminar and that the following equation holds for all $1 \leq i, j \leq n$.

$$D[i, j] = \sum_{C \in \mathcal{C}} \alpha_C D_C[i, j] \tag{2.4}$$

If we drop the requirement that the cuts be laminar, this approach can be used to recognize, not only tree metrics, but also a much larger class of metrics known as totally decomposable metrics.

Definition 2.1.3 *(Bandelt and Dress [8]) A metric D is called a* **totally decomposable metric** *(TDM) if the following equality holds for all $i, j \in S$.*

$$D[i, j] = \sum \alpha_C D_C[i, j] \tag{2.5}$$

$$D_1 = \begin{bmatrix} 0 & 2 & 2 & 3 & 3 \\ 2 & 0 & 2 & 1 & 2 \\ 2 & 2 & 0 & 1 & 1 \\ 3 & 1 & 1 & 0 & 2 \\ 3 & 2 & 1 & 2 & 0 \end{bmatrix} \qquad D_2 = \begin{bmatrix} 0 & 2 & 1 & 1 & 1 \\ 2 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 2 & 2 \\ 1 & 1 & 2 & 0 & 2 \\ 1 & 1 & 2 & 2 & 0 \end{bmatrix}$$

$$\alpha_{\{1\}} = \alpha_{\{5\}} = \alpha_{\{1,2\}} = \alpha_{\{2,4\}} = 1$$

A TDM A non-TDM

Figure 2.2:

Given a totally decomposable metric D, we define a **total decomposition** of D to be the collection $\mathcal{C}$ of d-splits along with their isolation indices α_C. Figure 2.2 shows a TDM D_1 and lists the isolation indices of the d_1-splits.

As you might expect, not every metric can be decomposed in this manner. In fact, some metrics, known as split prime metrics, have no d-splits.

Definition 2.1.4 *(Bandelt and Dress [8]) A metric D is called* **split prime** *if $\alpha_C = 0$ for every cut C of D.*

D_2 in Figure 2.2 is an example of a split prime metric. Avis and Deza [4] mention that D_2 can not be represented as a sum of weighted cut metrics, even if no restrictions are placed on the cuts.

In order to help develop some intuition about TDMs, we state the main results of Bandelt and Dress [8]. More technical results are presented as needed in subsequent chapters.

Theorem 2.1.5 *Every finite metric space D can be uniquely decomposed into a TDM D_0 and a split prime metric R such that*

$$D[i,j] = D_0[i,j] + R[i,j] \qquad \text{for all } i,j \in S$$

The metric R is called the **split prime residue** *of D.*

Although there are 2^{n-1} cuts in D, Bandelt and Dress have shown that there at most $\binom{n}{2}$ d-splits. Furthermore, they prove the following theorems on the structure of d-splits.

Theorem 2.1.6 *(Bandelt and Dress [8]) Given any metric D, the collection of d-splits is weakly compatible.*

Theorem 2.1.7 *(Bandelt and Dress [8]) A metric D is a TDM if and only if there exists a weakly compatible collection $\mathcal{C}$ of weighted cut metric such that*

$$D[i,j] = \sum_{(C,\overline{C}) \in \mathcal{C}} \alpha_C D_C[i,j]$$

in addition $\mathcal{C}$ is the collection of d-splits and α_C is the isolation index of $(C,\overline{C})$.

These theorems allow them to design an $O(n^5)$ algorithm for computing the d-splits and their isolation indices. Their algorithm first computes the isolation indices of all cuts in D_4, the sub-matrix of D induced by $\{1,2,3,4\}$. After computing the isolation indices of all cuts in D_i their algorithm computes the isolation indices of the cuts in D_{i+1} using linear algebra and the following theorem.

Theorem 2.1.8 *(Bandelt and Dress [8]) Let $\mathcal{C}$ be the collection of cuts in the decomposition of a TDM D let $T \subset S$, and let $\mathcal{C}^T$ be collection of cuts in the decomposition of D_T. The cut $(C^T, \overline{C^T})$ is an element of $\mathcal{C}^T$ if and only if it is induced by a cut $(C,\overline{C})$ of $\mathcal{C}$. In addition, the isolation index α_{C^T} of $(C^T, \overline{C^T})$ in the decomposition of D^T is the sum of the isolation indices of the cuts of $\mathcal{C}$ which induce $(C^T, \overline{C^T})$.*

Since any laminar collection of cuts is weakly compatible, tree metrics are totally decomposable. In fact, the union of two laminar collections of cuts is always weakly compatible. Therefore, any metric which is the sum of two tree metrics is a TDM.

Definition 2.1.9 *A metric D is called a* **2-tree metric** *if there exist two $n \times n$ tree metrics D_1 and D_2 such that*

$$D[i,j] = D_1[i,j] + D_2[i,j] \qquad \text{for all } i,j \in S$$

The tree representation of D_1 together with the tree representation of D_2 is called a **2-tree representation of** *D.*

2-tree metrics are interesting because some special case algorithms which work on tree metrics can be easily extended to two tree metrics. Unfortunately, this is not always the case.

Here is a well known subclass of 2-tree metrics. Let R_1^2 denote the metric space of points in the real plane with the metric

$$D[(x_1, y_1), (x_2, y_2)] = |x_1 - x_2| + |y_1 - y_2|$$

A metric D is called a **taxi metric** if there exists a map $\phi : S \to R_1^2$ such that $D_2[\phi(i), \phi(j)] = D[i, j]$.

It is easy to construct a two-tree decomposition of a taxi metric D. Denote a vertex v_i in a 2-tree metric by the coordinates (x_i, y_i) of $\phi(v_i)$. Create a weighted path $P = p_1, ..., p_n$ with n vertices. Label p_1 with the vertex (x_i, y_i) which has minimum x coordinate. Label p_2 with the vertex (x_i, y_i) which has second largest x coordinate. Continue this process, breaking ties arbitrarily, until all vertices in P have been labeled. Assign a weight equal to the absolute value of the difference of the x coordinates of the labels p_j and p_{j+1} to the edge $e_{p_i, p_{i+1}}$ in P. Repeat this process using the y coordinates of the vertices to construct a weighted path Q. P and Q from a 2-tree decomposition of D. Bandelt and Chepoi [7] prove several interesting results on the structure of the d-splits of a taxi metric. In chapter 4 we use the algorithmic proof of their main theorem as part of a recognition algorithm for 2-tree metrics.

2.2 Circular Decomposable Metrics

The final subclass of totally decomposable metrics we are interested in is the class of circular decomposable metrics (CDMs). Although CDMs are an extension of tree metrics, their definition is best motivated by considering convex Euclidean metrics.

Consider a finite subset $V = \{v_1, ..., v_n\}$ of points on the boundary of a convex polygon in the Euclidean plane. Define the metric D_2 where $D_2[i, j]$ is the Euclidean distance between v_i and v_j. D_2 is called a **convex Euclidean metric (CEM)**.

Assume that the points are labeled so that if you start at the point v_1 and travel around the boundary in a counter-clockwise direction the points are encountered in the order $v_1, v_2, ..., v_n$. This natural ordering of the points together with the structure of the Euclidean metric gives CEMs the following properties for all $i < j < k < l$

$$\begin{aligned} D[i,j] + D[k,l] &\leq D[i,k] + D[j,l] && (2.6)\\ D[i,l] + D[j,k] &\leq D[i,k] + D[j,l] && (2.7)\end{aligned}$$

These properties have been exploited to solve optimization problems on CEMs more quickly than it is possible to do in general. CDMs are defined to extend CEMs while preserving these properties. Before formally defining CDMs, we present two technical definitions that allow us to define a circular ordering for S_D when D is a CDM just as we did when D was a CEM.

Definition 2.2.1 *The* **node-cut incidence matrix** *of a TDM D is the matrix N which has one row for each $i \in S$ and one column for each d-split $(C_j, \overline{C}_j)$ with $N[i,j] = 1$ if $i \in C_j$ and $N[i,j] = 0$ if $i \in \overline{C}_j$.*

Definition 2.2.2 *We say that the node-cut incidence matrix has the* **circular ones property** *if there is a permutation ρ of the rows of N (indices of D) such that in each column of N, either all the ones or all the zeros occur in consecutive rows.*

CDMs are defined as a subclass of TDMs by imposing the circular ones property on the node-cut incidence matrix.

Definition 2.2.3 *A let D be a TDM whose cut-node incidence matrix has the circular ones property, and let ρ be a permutation of the indices of D as described in the previous definition. We call D a* **permuted circular decomposable metric**. *If ρ can be taken to be the identity permutation, D is called a* **circular decomposable metric** *(CDM). In addition ρ is called a* **Kalmanson permutation** *of D.*

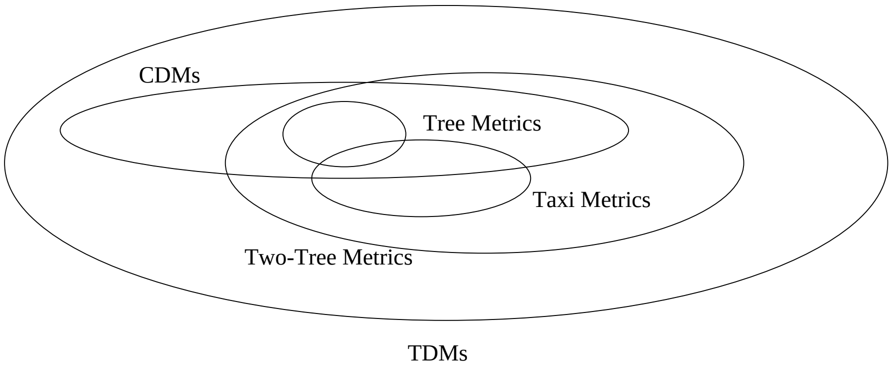
In order to simplify our discussion, the results in the following chapters are stated in terms of CDMs. The conceptually simple but notationally horrific reformulation of the results in terms of permuted CDMs is left to the reader.

A d-split $(C, \overline{C})$ of a CDM D has either $C = \{i, i+1, ..., j\}$ or $\overline{C} = \{i, i+1, ..., j\}$. Any cut of this form is called **compatible** with D. We assume the cuts have been complemented so that $C = \{i, i+1, ..., j\}$ with $i > 1$. Such a cut is denoted $C_{i,j}$. Figure 2.3 shows the distance matrix of a CDM metric and its non-zero isolation indices.

$$\begin{bmatrix} 0 & 5 & 7 & 5 & 2 & 1 \\ 5 & 0 & 2 & 4 & 7 & 6 \\ 7 & 2 & 0 & 2 & 5 & 6 \\ 5 & 4 & 2 & 0 & 3 & 4 \\ 2 & 7 & 5 & 3 & 0 & 1 \\ 1 & 6 & 6 & 4 & 1 & 0 \end{bmatrix}$$
$$\alpha_{\{1,2\}} = 1 \quad \alpha_{\{2,3\}} = 2$$
$$\alpha_{\{3,4,5\}} = 1 \quad \alpha_{\{2,3,4\}} = 3$$

Figure 2.3: A CDM

We close this section with a picture summarizing the relationship of the classes of metrics defined in this section.



The Totally Decomposable Heirarchy

Chapter 3

Properties of CDMs

3.1 Introduction

This chapter contains an in depth discussion of several important properties of CDMs. There are two main results in this section. The first gives a set of inequalities that characterizes CDMs. The second explains how PQ-trees can be used to store the set of Kalmanson permutations for a specific CDM.

Section 3.2 introduces Kalmanson matrices, and the master tour property. Theorem 3.2.3, due to Deĭneko, Rudolf, and Woeginger [21] relates these two ideas.

In section 3.3, we use theorem 3.2.3 to prove the equivalence of the classes of CDMs and Kalmanson metrics. This equivalence is used to characterize Kalmanson matrices in terms of cuts. These results are exploited in chapters 5 and 6 to design algorithms for the maximum weight b-matching problem, the prize collecting traveling salesman problem, and the k-MST problem on CDMs.

In section 3.4 we prove that there is a PQ-tree representing the set of Kalmanson permutations of a CDM. The simple algorithmic proof of the result combines some of Booth and Lueker's original ideas on PQ-trees with theorem 3.3.12 to give an $O(n^3)$ algorithm for computing the PQ-tree. The rest of the section explains how a careful analysis of optimal traveling salesman tours can be used to build this tree in $O(n^2)$ time. An example of this algorithm is given in section 3.5.

3.2 Kalmanson Matrices

In 1975 Kalmanson [29] defined the class of Kalmanson matrices as a generalization of convex Euclidean metrics.

Definition 3.2.1 *A $n \times n$ matrix D is called a* **Kalmanson matrix** *if the following properties hold for all $i < j < k < l$.*

$$D[i,j] + D[k,l] \leq D[i,k] + D[j,l] \tag{3.1}$$
$$D[i,l] + D[j,k] \leq D[i,k] + D[j,l] \tag{3.2}$$

If D is also a metric, it is called a **Kalmanson metric**.

We call (3.1) and (3.2) the Kalmanson conditions. These conditions do not place any restrictions on the diagonal entries of D. In order to simplify our discussion, we assume that every Kalmanson matrix is a Kalmanson metric and that $D[i,i] = 0 \ \forall i \in S_D$ for every Kalmanson metric. We show how to remove these assumptions at the end of section 3.3.

Kalmanson has shown that the TSP over Kalmanson matrices is solved by the identity permutation. Deĭneko, Rudolf, and Woeginger [21] have shown an even stronger result.

Definition 3.2.2 *An instance of the traveling salesman problem is said the have the* **master tour property** *if there is an optimal tour π such that the optimal tour for any subset of points is simply π with cities not in the subset omitted. π is called a* **master tour**.

Theorem 3.2.3 *(Deĭneko, Rudolf, and Woeginger [21])*
The tour $\langle 1, 2, ..., n\rangle$ is a master tour for the instance of the TSP with objective function coefficients given by the matrix D if and only if D is a Kalmanson matrix.

The main result of this chapter is that a metric D is a Kalmanson metric if and only if it is a CDM. This result is extended to characterize Kalmanson matrices in terms of cuts. In section 3.4, this equivalence is used to design an algorithm which finds all Kalmanson permutations of a Kalmanson matrix in $O(n^2)$ time. Since we will show that CDMs and Kalmanson metrics are equivalent, we will call the cuts $C_{i,j}$ compatible with Kalmanson metrics also.

3.3 Equivalence of Kalmanson and CDMs

In this section we prove the equivalence of Kalmanson and Circular Decomposable Metrics. First we apply a result of Dieneko, Rudolf and Woeginger to show that every CDM satisfies the Kalmanson conditions. Second we state a number of theorems on totally decomposable metrics which were originally proven by Bandelt and Dress. Finally we use these results to show that every metric which satisfies the Kalmanson conditions is a CDM.

3.3.1 CDMs are Kalmanson Metrics

We use the idea of a master tour to show that CDMs are Kalmanson metrics. First, we show that all CDMs are Kalmanson metrics, by proving that the tour $\langle 1, 2, \ldots, n \rangle$ is a master tour for a CDM.

Lemma 3.3.1 *An optimal tour for a CDM is the tour* $T = \langle 1, 2, ..., n \rangle$

Proof: We say that the tour $T_0 = \langle v_1, v_2, ..., v_n \rangle$ crosses the cut $(C, \overline{C})$ k times if there exist exactly k indices such that $(C, \overline{C})$ separates v_i and v_{i+1} (mod n). Consider an arbitrary tour T_1 of a CDM. The cut $(C, \overline{C})$ contributes α_C to the cost of the tour each time T_1 crosses $(C, \overline{C})$. Every tour crosses each trivial d-split exactly twice and each non-trivial d-split at least twice. T crosses each d-split exactly twice. Therefore, T is an optimal tour. ∎

Lemma 3.3.2 *If D is a CDM, then D^T is a CDM for any $T \subset S$.*

Combining these lemmas yields the following theorem.

Theorem 3.3.3 *Every CDM is a Kalmanson metric.*

Proof: Lemmas 3.3.1 and 3.3.2 imply that $\langle 1, 2, ..., n \rangle$ is a master tour of any CDM D. Since $\langle 1, 2, ..., n \rangle$ is a master tour for D, it must be a Kalmanson matrix by theorem 3.2.3. ∎

3.3.2 Kalmanson Metrics are CDMs

In order to show that every Kalmanson metric is a CDM we must show two things. First we must show that if D is a Kalmanson metric then any d-split is compatible with D. Second, we need to show that D has no split prime residue. The first step is relatively simple, but we need the following theorem of Bandelt and Dress [8] to accomplish our second goal.

Theorem 3.3.4 *(Bandelt and Dress [8]) If a metric, D, has $\binom{n}{2}$ d-splits, then D is a CDM.*

We would like to apply theorem 3.3.4 to show that a Kalmanson metric leaves no split prime residue. Unfortunately, not every Kalmanson metric has enough d-splits, so we construct a new metric D' such that

- Every d-split is a d'-split.
- There are $\binom{n}{2}$ d'-splits.
- D' has the same residue as D.

D' is constructed by adding cut metrics to D. The cut metrics added to D can be thought of intuitively as "d-splits" with weight 0 in the decomposition of D. Before a cut is added it must pass two tests: its isolation index must be 0, and it must be weakly compatible with the d-splits and any other cuts already added to D. This intuition is formalized in the next two definitions.

Definition 3.3.5 *(Bandelt and Dress [8]) Given a metric D and a cut C, C is said to be a* **virtual d-split** *if for all $\{a, a'\} \subset C$ and $\{b, b'\} \subset \overline{C}$ (a, a', b, b' not necessarily unique)*

$$D[a, a'] + D[b, b'] \leq max\{D[a, b] + D[a', b'], D[a, b'] + D[a', b]\} \tag{3.3}$$

Definition 3.3.6 *(Bandelt and Dress [8]) Two virtual d-splits $\{C, \overline{C}\}$ and $\{B, \overline{B}\}$ are called* **d-compatible** *if for all $x \in C \cap B$, $y \in \overline{B} \cap \overline{C}$, $v \in B \cap \overline{C}$, and $u \in C \cap \overline{B}$, the following holds*

$$D[x, y] + D[u, v] \geq max\{D[x, u] + D[y, v], D[x, v] + D[y, u]\} \tag{3.4}$$

The following theorem of Bandelt and Dress [8] can be applied to determine which cut metrics can be added to D without altering the structure of the d-splits.

Theorem 3.3.7 *(Bandelt and Dress [8]) Let R be a split prime metric and let D be a totally decomposable metric. R is the split prime residue of $R + D$ exactly when the collection $\mathcal{C}$ of d-splits consists of pairwise r-compatible virtual r-splits.*

Let $D = D^0 + R$ as in theorem 2.1.5. The basic idea of the proof is to describe a collection $\mathcal{C}^R$ of pairwise r-compatible virtual r-splits, use $\mathcal{C}^R$ to build the metric D' described above, and apply theorem 3.3.4 to show that R is the zero metric. The proof begins with three lemmas describing $\mathcal{C}^R$.

Lemma 3.3.8 *Given a Kalmanson metric D, any d-split is compatible with D.*

Proof: Consider any cut C which is not compatible with D. Since C is not compatible with D there must be $\{a, a^{'}\} \subset C$ and $\{b, b'\} \subset \overline{C}$ such that either $a < b < a' < b'$ or $b < a < b' < a'$. Since D is a Kalmanson metric,

$$D[a,a'] + D[b,b'] \geq max\{D[a,b] + D[a',b'], D[a,b'] + D[b,a']\} \tag{3.5}$$

Combining (3.5) with the definition of α_C yields that $\alpha_C = 0$. ∎

Lemma 3.3.9 *Given a Kalmanson metric D any cut of the form $C_{i,j}$ is a virtual d-split.*

Proof: Consider $\{a, a'\} \subset C$ and $\{b, b'\} \subset \overline{C}$. We need to show

$$D[a,a'] + D[b,b'] \leq \max D[a,b] + D[a',b'], D[a',b] + D[b',a]$$

There are three cases we need to consider:

$$\begin{array}{ll} Case\ 1: & a < a' < b < b' \\ Case\ 2: & b < b' < a < a' \\ Case\ 3: & b < a < a' < b' \end{array}$$

If $a < a' < b < b'$, then, since D is a Kalmanson metric,

$$\begin{aligned} D[a,a'] + D[b,b'] &\leq D[a,b] + D[a',b'] \\ &\leq \max\{D[a,b] + D[a',b'], D[a,b'] + D[a',b]\} \end{aligned}$$

The other two cases are handled similarly. Therefore C is a virtual d-split. ∎

Lemma 3.3.10 *Let R denote the split prime residue of a Kalmanson metric D. The collection $\mathcal{C}$ of cuts of the form $C_{i,j}$ $1 < i \leq j \leq n$ is a collection of pairwise r-compatible virtual r-splits.*

Proof: First we show that each $C_{i,j}$ is a virtual r-split. Bandelt and Dress have shown that if D is a metric, so is R. Therefore every trivial cut is a virtual r-split. Suppose that there is a $C_{i,j}$ which is not a virtual r-split. This means that there exist sets of vertices $\{a, a'\} \in \{i, ..., j\}$ and $\{b, b'\} \in \{1, ..., i-1, j+1, ..., n\}$ such that

$$R[a, a'] + R[b, b'] \quad > \max\{R[a, b] + R[a', b'], R[a, b'] + R[a', b]\}$$

Again, we have the 3 cases of theorem 3.3.9 and only explicitly consider the first. By assumption, $R[a, a'] + R[b, b'] > R[b', a] + R[b, a']$. Let $\mathcal{C} = \{C_1, ..., C_k\}$ be the set of d-splits. Theorem 3.3.7 states that every $C \in \mathcal{C}$ must be a virtual r-split. Thus, there can be no d-split separating the sets $\{a, a'\}$ and $\{b, b'\}$. This implies that $D[a, a'] + D[b, b'] > D[b, a'] + D[b', a]$, contradicting the fact that D is a Kalmanson matrix. Therefore each $C_{i,j}$ is a virtual r-split.

Now we show that the cuts in $\mathcal{C}$ are pairwise r-compatible. Consider $C_{i,j} = C$ and $C_{k,l} = B$. If the cuts are laminar, then there is nothing to show. If C and B are not laminar, then they break $\{1, ..., n\}$ into 4 subsets. Up to exchanging the roles of C and B, these subsets are of the form $\overline{B} \cap \overline{C} = \{1, ..., i_1\} \cup \{i_4 + 1, ..., n\}$, $C \cap \overline{B} = \{i_1 + 1, ..., i_2\}$, $C \cap B = \{i_2 + 1, ..., i_3\}$, and $\overline{C} \cap B = \{i_3 + 1, ..., i_4\}$. ($i_4$ may be n.) Therefore we can find $y \in \overline{B} \cap \overline{C}$, $u \in C \cap \overline{B}$, $x \in C \cap B$ and $v \in \overline{C} \cap B$ such that $y < u < x < v$ or $u < x < v < y$.

In order for C and B to be r-compatible, we need to show that

$$R[x, y] + R[u, v] \geq max\{R[x, u] + R[y, v], R[x, v] + R[y, u]\}$$

Suppose not, then either $R[x, u] + R[y, v] > R[x, y] + R[u, v]$ or $R[x, v] + R[y, u] > R[x, y] + R[u, v]$. The first of these cases contradicts the fact that C is a virtual r-split, and the second contradicts the fact that B is a virtual r-split, Therefore B and C are r-compatible. ∎

Now we combine these three lemmas and apply theorems 3.3.4 and 3.3.7 to get the main result of this section.

Theorem 3.3.11 *If D is a Kalmanson metric, then D is a CDM.*

Proof: Let $\mathcal{C}_0$ denote the collection of cuts of the form $C_{i,j}$ such that $\alpha_{C_{i,j}} = 0$. Let $\mathcal{C}_1$ denote the collection of cuts of d-splits. Let $\mathcal{C} = \mathcal{C}_0 \cup \mathcal{C}_1$. Let R denote the split prime residue of D.

Create a new metric D' with

$$\begin{aligned} D'[i,j] &= D[i,j] + \textstyle\sum_{C \in \mathcal{C}_0} D_C \\ &= R[i,j] + \textstyle\sum_{C \in \mathcal{C}_0} D_C + \sum_{C \in \mathcal{C}_1} \alpha_C D_C \end{aligned}$$

Since $\mathcal{C}$ is a collection of pairwise r-compatible virtual r-splits, D and D' have the same residue, R. Since there are $\binom{n}{2}$ d'-splits, R is the zero metric. Finally, note that the node-cut incidence matrix N of D has the circular ones property. Therefore, D is a CDM. ∎

This completes the proof that every Kalmanson metric is a CDM, however, the Kalmanson condition does not require that D be a metric. In the case that D is a symmetric function that does not necessarily satisfy the triangle inequality, it is straightforward to show that D is CDM with possibly negative weight trivial cuts. This can be done by adding M to each entry of D to get D_M where M is any number larger that the maximum entry of D. Computing the decomposition of D_M and then subtracting $\frac{M}{2}$ from the weight of each trivial cut gives us a "decomposition" of D with (possibly) negative weight trivial cuts. Therefore we can combine the last two theorems to get the main result of this section.

Theorem 3.3.12 *A matrix K is a Kalmanson matrix if and only if*

$$K[i,j] = D[i,j] + c_i + c_j \qquad \text{for all } i \neq j$$

where D is a CDM and c_i is a non positive constant, and the diagonal entries of K are arbitrary constants.

This result allows us to use the Kalmanson conditions in proofs involving CDMs.

3.4 Computing all Kalmanson Permutations

3.4.1 Introduction

Deǐneko, Rudiger, and Woeginger [21] gave an algorithm which can find a Kalmanson permutation of a permuted Kalmanson metric D and posed the problem of finding all such permutations. To solve this problem we use the PQ tree data structure developed by Booth and Lueker [15] as a compact way to represent sets of permutations. Briefly, each PQ tree, T, describes a set $\Pi(T)$ of permutations. We say that the permutation π is **compatible with the PQ tree** T if $\pi \in \Pi(T)$. We combine this data structure and definition 2.2.3 to prove the following theorem:

Theorem 3.4.1 *Given a CDM D there exists a PQ tree T describing the set of Kalmanson permutations of D.*

To prove this theorem, we need to explicitly define what it means for a PQ tree to represent the set of all Kalmanson permutations of a CDM. We use the notation $\langle s_1, s_2, ..., s_n \rangle$ to denote the permutation of the set S which places s_1 in the first position s_2 in the second position, ..., and s_n in the final position. Let $\pi = \langle s_1, s_2, ..., s_n \rangle$ and $\rho = \langle s_k, s_{k+1}, ..., s_n, s_1, ..., s_{k-1} \rangle$ (for some $1 \leq k \leq n$). The permutations π and ρ are called **equivalent up to rotation**. It follows immediately from definition 2.2.3 that if π and ρ are equivalent up to rotation then π is a Kalmanson permutation if and only if ρ is a Kalmanson permutation. This justifies the following definition.

Definition 3.4.2 *We say that the PQ tree T represents the set of Kalmanson permutations of a CDM D if the following hold.*

- *Every permutation compatible with T is a Kalmanson permutation.*
- *Every Kalmanson permutation with* 1 *in the first position is compatible with T.*

This section covers three topics. First, a brief description of PQ trees is given. Second, we prove theorem 3.4.1 by exhibiting an $O(n^3)$ algorithm which constructs the PQ tree described in the statement of the theorem. Finally, we discuss an algorithm which constructs this PQ tree in $O(n^2)$ time.

3.4.2 PQ trees and the proof of Theorem 3.4.1

A **PQ tree** on the ground set S is a rooted tree T with n leaves whose nodes are labeled according to the following rules:

- Each leaf of T is labeled with an element of S.

- Each element of S labels a unique leaf of T.
- Each non-leaf node of T is labeled either P or Q.

In addition, consider the children of any node to be ordered from left to right. A minimal description of PQ trees follows. For further information about PQ trees, see Booth and Lueker [15].

Our approach in describing how PQ trees represent permutations follows the explanation given in Burkard, Deĭneko, and Woeginger [17]. We denote the set of all permutations compatible with the PQ tree T by $\Pi(T)$ and denote the set of leaves on the PQ tree T by $L(T)$.

$\Pi(T)$ for a PQ tree T with root r can be defined recursively according to the following rules. Let $v_1, ..., v_j$ denote the (ordered) set of children of r, and let T_i denote the maximal subtree of T rooted at v_i. If r is a P node,

$$\Pi(T) = \bigcup_{\rho \in \mathcal{S}_j} \Pi(T_{\rho(1)}) * ... * \Pi(T_{\rho(j)})$$

where $\mathcal{S}_j$ is the set of all permutation of the set $\{1, ..., j\}$ and $*$ represents concatenation. If r is a Q node,

$$\Pi(T) = \Pi(T_1) * ... * \Pi(T_j) \cup \Pi(T_j) * ... * \Pi(T_1)$$

There may be more than one PQ tree representing a set of permutations. In order to avoid any confusion this may cause, we will place restrictions on the PQ tree our algorithm builds that make it unique. In addition to the standard restrictions placed on PQ trees to guarantee a unique topology of the tree representing a set of permutations, we also restrict how the tree will be drawn once a topology is chosen. This is done with the following definition.

Definition 3.4.3 *Given a planar drawing of a PQ tree. The* **frontier** *of the drawing is the permutation of the elements of S obtained by reading the labels of the leaves from left to right.*

Booth and Lueker [15] mention that a permutation of the elements of S is the frontier of a drawing of a PQ tree if and only if it is in the set of permutations described by that tree. In order to force the topology of our PQ trees and the way they are drawn to be unique we make the following assumptions.

- Every P node has at least two children.
- Every Q node has at least three children.
- The frontier of the PQ tree is the identity permutation.

The third assumption is reasonable, since the identity permutation is always a Kalmanson permutation of a Kalmanson matrix. The PQ tree data structure was originally developed by Booth and Lueker [15] to help answer the following question:

Question: *Given a collection of subsets $\mathcal{S}$ of the set $S = \{1, 2, ..., n\}$ does there exist a permutation π of S such that for each $S_i \in \mathcal{S}$ the elements of S_i occur consecutively in π.*

Their algorithm for answering this question proceeds as follows. Initially the algorithm builds a PQ tree T_0 describing all permutations of $\mathcal{S}$. In iteration k, the algorithm modifies the tree T_{k-1}, which describes all permutations π such that the elements of S_i occur consecutively in π for all $i < k$ to get a new tree T_k describing all permutations such that the elements of S_i occur consecutively in π for all $i \leq k$. (If no such permutations exist, T_k chosen to be an empty tree, i.e. $V(T_k) = \emptyset$.) Booth and Lueker give an algorithm for building T_0 in $O(n + |\mathcal{S}| + \sum_{S \in \mathcal{S}} |S|)$ time.

Booth and Lueker's algorithm for building PQ trees can be combined with Theorem 3.3.12 and Farach's [23] algorithm for decomposing CDMs to construct a PQ tree describing all the Kalmanson permutations of a CDM D. First use Farach's algorithm to decompose D. Second, let the collection $\mathcal{S}$ of subsets of S contain the set $\{i, ..., j\}$ if and only if $C_{i,j}$ is a d-split. Third, construct a PQ tree T from $\mathcal{S}$ using Booth and Lueker's algorithm.

Every Kalmanson permutation with 1 in the first position must visit the elements of each set in $\mathcal{S}$ consecutively. Therefore, it follows from the definition of a PQ tree that each of these permutations is described by T. Similarly, a permutation π is a Kalmanson permutation if it visits the elements of S consecutively for each $S \in \mathcal{S}$. Therefore every permutation described by T is a Kalmanson permutation, and we have proven theorem 3.4.1.

Unfortunately, this approach to building a PQ tree is slow. Since Booth and Lueker's algorithm requires $O(n + |\mathcal{S}| + \sum_{S \in \mathcal{S}} |S|)$ time to build T, and there may be $O(n^2)$ members of $\mathcal{S}$ each of which may contain $O(n)$ elements of S, this algorithm runs in $O(n^3)$ time.

3.4.3 An $O(n^2)$ Algorithm

Here we consider a different approach to building a PQ tree describing all Kalmanson permutations of a CDM. The new algorithm never figures out all the d-splits; instead it infers which $C_{i,j}$ are not d-splits by examining $O(n^2)$ traveling salesman tours.

The inference works as follows. Let $\langle v_1, v_2, ..., v_n \rangle$ denote the tour which starts at v_1 then visits v_2 through v_n, and returns to v_1. Let $T_0 = \langle 1, 2, ..., n \rangle$, and $T[i, j]$ denote the tour $\langle 1, ...i-1, j, j-1, ..., i, j+1, ..., n \rangle$. By lemma 3.3.1, we know that T_0 is an optimal tour, and that its cost is twice the sum of the isolation indices of D. If $T[i, j]$ is also optimal, no $C_{k,l}$ with $k < i < l < j$ or $i < k < j < l$ can be a d-split, since $T[i, j]$ crosses these cuts four times. This inference allows our algorithm to build the PQ tree directly from the metric without computing the decomposition.

The algorithm proceeds in two phases. Phase 1 starts with the leaves of the tree and uses the costs of specific tours to deduce the tree's structure from the bottom up. Phase 2 builds the tree from the top down using information collected in phase 1.

Given a node n in a PQ tree, we use the phrase **descendants of the node** n to indicate the set of labels on the leaves of the sub-tree rooted at n. Notice that, since D is a Kalmanson metric, there must exist a PQ tree H, with frontier $\{1, 2, ..., n\}$, which describes all permutations compatible with D. This fact implies that the set of descendants of any node n of H is of the form $\{i, i+1, ..., j-1, j\}$. Therefore the set of descendants of a node can be represented using only 2 elements of S.

A doubly linked list $\mathcal{L}$ of **buckets** is the main data structure the algorithm uses to store the structure of the PQ tree. Each bucket B holds six fields of information: its successor $S(B)$, its predecessor $P(B)$, the set of buckets representing its children, its smallest descendant $m(B)$, its largest descendant $M(B)$, and its label $L(B)$. $L(B)$ is set to P, Q, or L depending on whether the algorithm thinks that B corresponds to a P node, a Q node, or a leaf in H. $P^k(B)$ denotes the k^{th} predecessor of B and $S^k(B)$ denotes the k^{th} successor of B. If $P^k(B)$ is null then we will define $P^l(B)$ to be null whenever $l > k$. A similar convention is adopted for handling successors of null nodes. There will always be a unique bucket B of $\mathcal{L}$ with $P(B) =$ null this bucket is called the **first bucket** of $\mathcal{L}$.

As the algorithm progresses, some buckets are removed from $\mathcal{L}$ and replaced by newly formed buckets. Buckets in $\mathcal{L}$ are called **active** buckets.

Buckets not in $\mathcal{L}$ are called **inactive**. A set $\mathcal{B}$ of active buckets is called **consecutive** if $\mathcal{B} = \{B, S(B), ..., S^k(B)\}$ for some bucket B and some positive integer k.

The predecessor and successor fields define a linear ordering on the active buckets. Given two active buckets B_1 and B_2 we say that B_1 occurs before B_2 if $S^k(B_1) = B_2$ for some positive integer k. If B_1 occurs before B_2 we will write $B_1 < B_2$.

Here is an intuitive description of the two main procedures used in the algorithm: collapsing and merging. The goal of a collapse is to create a bucket which corresponds to a node in the PQ tree. Unfortunately, collapses are not perfect. Occasionally a bucket is created which does not correspond to a node of the PQ tree. Merging fixes any mistakes that have been made by previous collapses. There are two types of collapses. A **Q collapse** occurs when the algorithm recognizes a consecutive subset of the active buckets which corresponds to the complete set of children of a Q node. A **P collapse** occurs when the algorithm recognizes a maximal consecutive subset of active buckets which correspond to children of a single P node. Merging is used to combine buckets corresponding to consecutive sets of children of the same P node. After each collapse a clean up procedure is carried out. How the cleanup is carried out depends on the type of collapse that preceeded it. An example of how the algorithm works is given in section 3.5.

Now we present a formal description of the procedures used in the algorithm. Since P collapses are simpler, we discuss them first. Let $\{B_i, S(B_i), ..., S^l(B_i) = B_j\}$ denotes a maximal set of buckets such that $T[m(B), M(S(B))]$ is optimal for every B such that $B_i \leq B < B_j$. In a P collapse, buckets $B_i, ..., B_j$ are stored in memory, removed from $\mathcal{L}$ and replaced by a single bucket B. The fields of B are set as follows.

- $P(B) = P(B_i)$
- $S(B) = S(B_j)$
- The list of children of B is $B_i, ..., B_j$
- $m(B) = m(B_i)$
- $M(B) = M(B_j)$
- $L(B) = P$

P collapses also make the following changes to buckets of $\mathcal{L}$.

- $P(S(B_j)) = B$ if $S(B_j)$ is not null.
- $S(P(B_j)) = B$ if $P(B_j)$ is not null.

Since P collapses are designed to recognize a maximal consecutive subset of active buckets that are children of a single P-node, not all children of a P node, occasionally a P collapse creates a bucket containing a subset of the children of a P node. In order to correct this problem, the algorithm executes a merge step following each P collapse. Let $\{\overline{B_1}, ..., \overline{B_c}\}$ denote the set of children of B with label P. Let B' denote the child of B such that $M(B') = M(B)$. Let B_i' denote the child of $\overline{B_i}$ such that $M(B_i') = M(B_i)$. For each $\overline{B_i} \neq B'$ check the tour $T[m(B_i'), M(B')]$. If this tour is optimal, each child of $\overline{B_i}$ is added to the list of children of B and $\overline{B_i}$ is deleted from memory. If $\overline{B_i} = B'$ re-choose B' to be the child of B such that $m(B') = m(B)$, and let B_i' be the child of $\overline{B_i}$ with $m(B_i') = m(\overline{B_i})$. Check the tour $T[m(B'), M(\overline{B_i})]$ for optimality. In either case, if the tour checked is optimal, add each child of $\overline{B_i}$ to the list of children of B and $\overline{B_i}$ is deleted from memory. The only time buckets are ever deleted is in this merge step.

Now we turn our attention to Q collapses. Let k denote the minimal value of y such that for some $B_x \in \mathcal{L}$, $T[m(B_x), M(S^y(B_x))]$ is optimal. A Q collapse occurs when $k \geq 2$ and B_x is chosen to be the minimal bucket for which $T[m(B_x), M(S^y(B_x))]$ is optimal. When this occurs, the algorithm executes a Q collapse by removing the buckets $B_x, S(B_x), ..., S^y(B_x) = B_z$ from the list and replacing them with a bucket B where the fields of B are set as follows

- $P(B) = P(B_x)$
- $S(B) = S(B_z)$
- The list of children of B is $B_x, ..., B_z$
- $m(B) = m(B_x)$
- $M(B) = M(B_z)$
- $L(B) = Q$

Q collapses also make the following changes to buckets of $\mathcal{L}$.

- $P(S(B_z)) = B$ if $S(B_z)$ is not null.
- $S(P(B_x)) = B$ if $P(B_x)$ is not null.

As we will see in the description of the full algorithm, the following distance property will hold before any collapse: $T[m(B_1), M(P^w(B_2))]$ is suboptimal if $w < k$, or if $w = k$ and $B_1 < B_x$.

However, the distance property does not usually hold after a Q collapse. The algorithm needs to restore the distance property before continuing. To do this, the algorithm checks tours $T[m(B_1), M(B_2)]$ where the number of links between B_1 and B_2 has decreased below k. This is done in the simplest possible way. Just invoke a procedure which receives k as a parameter and checks the following tours in order, skipping any tours for which $P^l(B)$ or $S^l(B)$ is null. These tours are checked to make sure that the distance property holds before the algorithm continues.

$$\begin{gathered} T[m(P(B)), M(B)] \\ T[m(B), M(S(B))] \\ T[m(P^2(B)), M(B)] \\ T[m(P(B)), M(S(B))] \\ T[m(B), M(S^2(B))] \\ \vdots \\ T[m(B), M(S^k(B))] \end{gathered}$$

Collapses occurring in the cleanup procedure are treated exactly as in the rest of the algorithm, including a recursive call to the cleanup procedure. If $T[m(P(B)), M(B)]$ or $T[m(B), M(S(B))]$ or both are optimal a P collapse occurs, otherwise any optimal tour found leads to a Q collapse. Collapses during the recursive cleanup process may change the list of tours which need to be checked when control is returned from the recursion, but the end goal is always to restore the distance property. This procedure looks expensive, but as we see later it is not.

We are now able to describe the algorithm. Initially the active buckets $B_1, ..., B_n$ are defined according to the following rules:

- $P(B_i) = B_{i-1}$ (except $P(B_1) =$ null)
- $S(B_i) = B_{i+1}$ (except $S(B_n) =$ null)
- B_i has no children

- $m(B_i) = i$
- $M(B_i) = i$
- $L(B_i) = L$ (for Leaf)

The main computation carried out in phase one is checking weather $T[m(B_x), M(B_z)]$ $(B_x < B_z)$ is an optimal tour for the TSP. In order to check these tours efficiently the algorithm keeps track of a pointer B_x and counter y, and chooses $B_z = S^y(B_x)$. The algorithm initially points B_x at the first bucket of $\mathcal{L}$, sets $y = 1$ and begins checking tours. If $T[m(B_x), M(S^y(B_x))]$ is not optimal, B_x is pointed to $S(B_x)$. Whenever $S^y(B_x)$ is null, y is incremented and B_x is reset to the first bucket on the list. When $y = 1$ and the algorithm finds a B such that $T[m(B), M(S(B))]$ is optimal, it records this fact, and continues checking tours until it finds a bucket B' such that either $S(B')$ is null or $T[M(B'), M(S(B'))]$ is not optimal, then the buckets $B, S(B), ..., B'$ are collapsed using a P collapse. The clean up procedure follows the collapse. When $y > 1$ and an optimal tour $T[m(B_x), M(S^y(B_x)]$ is found a Q collapse and clean up occurs as described above. At some point there will be fewer than k buckets in L when B_x points to the first bucket. If there are more than two buckets remaining, the algorithm collapses the remaining buckets into a Q node. If there are exactly two buckets remaining the algorithm collapses them into a P node. Phase one then halts.

When phase one finishes, a PQ tree, H, is built from the buckets. Each bucket B corresponds to a node in H. If $L(B) = P$ then B is a P node. If $L(B) = Q$ then B is a Q node. If $L(B) = L$ then B is a leaf labeled with the index $m(B)$. The node B' is a child of B in H if B' is listed in the child field of B. H is built from the top down in a breadth first fashion following the data in child field.

This concludes of the description of the algorithm. A pseudocode version of and an example are given in section 3.5. In the pseudo code, B_1 denotes the first bucket in L and B denotes the most recently added bucket.

Now we must prove that Algorithm 3.1 finishes quickly and correctly.

Theorem 3.4.4 *The algorithm described in the preceeding paragraphs runs in $O(n^2)$ time.*

Proof: First we consider the amount of time spent in phase one, then the time spent in phase two. During phase one, each tour $T[i, j]$ is examined at most once outside of the merging procedure and at most once within all merging procedures. To check the optimality of $T[i, j]$ the algorithm compares its cost to the cost of T_0 which is known to be optimal. This comparison takes $O(1)$ time, since $T[i, j]$ can be built from T_0 by swapping two edges. This leads to a total of $O(n^2)$ time required to do all the tour checking. $O(n)$ buckets are created, and each takes $O(n)$ time to create, for a total of $O(n^2)$ time spent creating buckets. Therefore phase one runs in $O(n^2)$ time.

The breadth first search which builds H in phase two takes only $O(n)$ time since there are only $O(n)$ nodes H. Therefore the algorithm will halt in $O(n^2)$ time. ∎

Before embarking on the proof that our algorithm computes PQ tree describing all Kalmanson permutations of a CDM, we need to develop a little machinery.

Definition 3.4.5 *A Q node q in a PQ tree H is called a* **leaf q node** *if all of its children are leaves of H.*

Here is an easy theorem about leaf q nodes.

Theorem 3.4.6 *Let D be an $n \times n$ CDM, let $\mathcal{C}$ denote the collection of d-splits, and let H be the PQ tree describing the Kalmanson permutations of D. Assume $1 \leq i < j \leq n$. The following are equivalent.*

1. *The set $S_0 = \{i, i+1, ..., j\}$ is the complete set of children of a leaf q node in H.*

2. *The following conditions hold*

   (a) *$C_{i,j}$ is laminar with every cut of $\mathcal{C}$*

   (b) *For every $i \leq a < b < j$ and every $i < a < b \leq j$ the cut $C_{a,b}$ is not laminar with some cut $C \in \mathcal{C}$.*

3. *The tours T_0 and $T[i, j]$ are both optimal tours, but none of the tours $T[a, b]$ where $i \leq a < b < j$ or $i < a < b \leq j$ are optimal.*

Proof: We show $(1) \Longrightarrow (3) \Longrightarrow (2) \Longrightarrow (1)$.

(1) implies (3) by the definition of a leaf q node and the correspondence between Kalmanson permutations and optimal tours. The optimality of $T[i, j]$ implies that (2a) holds, the non-optimality of $T[a, b]$ when

$i < a < b \leq j$ or $i \leq a < b < j$ implies (2b). Therefore (3) implies (2). Since there exists a cut $C \in \mathcal{C}$ which is not laminar with $C_{a,a+1}$ for all $i \leq a < j$ there can be no optimal tour which visits a vertex between a and $a+1$. Therefore the vertices $\{i, ..., j\}$ must be visited in either decreasing or increasing order. Since every $C \in \mathcal{C}$ is laminar with $C_{i,j}$ they can be visited in either order. Thus (2) implies (1). ∎

Definition 3.4.7 *The set* $\{i, ..., j\}$ *is a* **maximal contiguous subset** *of the children of a P node p in a PQ tree if each a* ($i \leq a \leq j$) *is a child of p but neither* $i - 1$ *nor* $j + 1$ *is a child of p.*

When $i - 1$ or $j + 1$ is not between 1 and n the definition ignores any requirements placed on those nodes. Here is the P node version of theorem 3.4.6.

Theorem 3.4.8 *Let D, $\mathcal{C}$, and H be as described in the previous theorem. Let $1 < i < j < n$. The following are equivalent.*

1. *The set $S_0 = \{i, i + 1, ..., j\}$ is a maximal contiguous subset of the children of a P node p of H.*

2. (a) *There exist cuts $C_1, C_2 \in \mathcal{C}$ (not necessarily distinct) such that $C_{i-1,j}$ is not laminar with C_1 and $C_{i,j+1}$ is not laminar with C_2.*

   (b) *For every $i \leq a < b < j$ and every $i < a < b \leq j$ the cut $C_{a,b}$ is laminar with every cut $C \in \mathcal{C}$.*

3. *$T[a, a+1]$ is optimal for $i \leq a < j$, but, neither $T[i-1, i]$ nor $T[j, j+1]$ is optimal.*

Proof: We show (1) $\Longrightarrow$ (3) $\Longrightarrow$ (2) $\Longrightarrow$ (1).

(1) implies (3) by the definition of a p node and the maximality of S_0. Since $T[a, a+1]$ is optimal for each $i \leq a < j$ (3) implies (2b). Since $T[i-1, i]$ and $T[j, j+1]$ are not optimal, there must be cuts C_1 and C_2 as described in (2a). Therefore (3) implies (2). Since every $C \in \mathcal{C}$ is laminar with every cut $(C, \overline{C})$ where $C \subset S_0$, the vertices of S_0 may be permuted among themselves in any tour. Therefore $\{i, ..., j\}$ is a contiguous subset of the children of a P node, p. Since there is a $C \in \mathcal{C}$ which separates $i - 1$ and i and a cut $C' \in \mathcal{C}$ which separates j and $j + 1$ neither $i - 1$ nor $j + 1$ can be children of p. Therefore (2) implies (1). ∎

A similar version of theorem 3.4.8 can be proved when $i = 1$, $j = n$ or both, but we will use but not explicitly state the theorem. (The restated theorem simply ignores any conditions placed on indices 0 and $n+1$.) We are now ready to prove that the algorithm gives us the correct PQ tree.

Theorem 3.4.9 *When run on a Kalmanson metric D, the algorithm described in this section outputs a PQ tree describing all Kalmanson permutations of D.*

Proof: Let D be a counterexample to the theorem chosen such that n is minimal, and (given the minimality of n) such that $|\mathcal{C}|$ is maximal, where $\mathcal{C}$ is the collection of d-splits. Let H be the PQ tree the algorithm builds and let H' be the PQ tree describing all Kalmanson permutations of D.

First, we show that our assumption of the maximality of $|\mathcal{C}|$ is justified. When every $C_{i,j}$ is a d-split it is clear that the only optimal tours will be T_0 and $T[n,1]$. Therefore, both H and H' will be the PQ tree where the root is a Q node and $\{1,...,n\}$ are leaves which are children of that node.

Consider the consequences of the first collapse of buckets. Let the set of buckets collapsed be $B_i, B_{i+1}, ..., B_j$ and let the bucket formed by the collapse be denoted by B. The case where this collapse is a Q collapse and the case where this collapse is a P collapse are considered separately.

Q collapses are simpler to handle so we discuss them first. The buckets collapsed correspond to leaves in the PQ tree H. The bucket created corresponds to a leaf Q node in H. In order for the algorithm to collapse B, condition 3 of theorem 3.4.6 must hold. Applying theorem 3.4.6 shows that indeed the set $\{i,...,j\}$ is the complete set of children in a leaf Q node in H'.

Consider the metric D_0 formed from D by deleting rows and columns i through j and inserting a new row v in their place. With the entries $D_0(v,k)$ defined as follows

- $D_0(v,k) = D(i,k)$ when $k < i$
- $D_0(v,k) = D(j,k)$ when $k > j$

Notice that D_0 has less than n rows and columns, since $j - i \geq 3$. It is easy to verify that D_0 is a CDM with decomposition $\mathcal{C}_0$ formed from $\mathcal{C}$ as described below. (Recall that we have complemented the cuts in $\mathcal{C}$ so that each is of the form $C_{s,t}$ $s > 1$.)

- Whenever C is added to $\mathcal{C}_0$, $\alpha_0 = \alpha$, where α is the isolation index of C in $\mathcal{C}$.
- For each $C_{a,b} \in \mathcal{C}$ with $i < a < b < j$, add no cut to $\mathcal{C}_0$.
- For each $C_{a,b} \in \mathcal{C}$ with $a < b < i$ or $j < a < b$, add the cut $C_{a,b}$ to $\mathcal{C}_0$.
- For each $C_{a,b} \in \mathcal{C}$ with $a < i \leq b < j$, add the cut $C_{a,v}$ to $\mathcal{C}_0$.
- For each $C_{a,b} \in \mathcal{C}$ with $i < a \leq j < b$, add the cut $C_{v,b}$ to $\mathcal{C}_0$.
- If $C_{i,j} \in \mathcal{C}$, add $C_{v,v}$ to $\mathcal{C}_0$.

Since D is a minimal counterexample, the algorithm will get a correct PQ tree H_0 when run on D_0. It is easy to see that the correct tree H' for D is just H_0 with the leaf v replaced by a Q node with children $i, ..., j$. This is also the tree H built by the algorithm, contradicting the fact that D is a counterexample.

The case where the first collapse is a P collapse is a little harder to handle, because there are two possible outcomes for the bucket B created by the collapse. B could become a P node in the final tree, or B could be merged into another bucket. In order to avoid a lot of special case analysis when $i = 1$ or $j = n$ we will assume that neither of these cases occurs. The proof is easily modified to handle these cases.

If B is not merged and becomes a P node p, the analysis follows the analysis of the Q collapse above. The children of p are identified, the metric D_0 is defined as above, and the same contradiction is reached. The only difference is that where we used theorem 3.4.6, we now use theorem 3.4.8.

Suppose that the bucket B created by the P collapse is merged later in the algorithm. Since B was merged, there is at least one optimal tour which crosses the cut $C_{i,j}$ at least four times. Therefore, $C_{i,j}$ is not an element of $\mathcal{C}$. Since $C_{i,j}$ is a virtual d-split, we can create a new CDM D_1 where

$$D_1[k,l] = D[k,l] + D_{C_{i,j}}[k,l] \qquad \text{for all } k, l \in S$$

When the algorithm runs on D_1 it will run exactly as it did on D except it will not merge B. It will also find the correct PQ tree H_1 describing all Kalmanson permutations of D_1. The only time a bucket B_2 is ever merged is when its label and its parent's label are both P. This implies that H_1 contains a P node p_1 whose children are the leaves i through j and whose parent is a P node p_2 with child set $\mathcal{S}$. Let H_2 denote the PQ tree which is formed from H_1 by removing p_1 and its children and adding the leaves i through j to $\mathcal{S}$. Clearly H_1 is the tree obtained from H_2 by adding the cut $C_{i,j}$ to $\mathcal{C}$. Therefore, since proper PQ trees are unique up to their frontier and our algorithm builds a proper PQ tree with frontier $1, ..., n$, H_2 is the same PQ tree as H'.

Notice that the only difference between the run of the algorithm on D_1 and on D is the merger of B. Therefore, the only difference between the tree H and the tree H_1 is the same replacement of children that occurs in the previous paragraph. Thus, $H = H'$ and the algorithm found the correct tree, contradicting the fact that D was a minimal counterexample. Therefore there is no minimal counterexample and the theorem is true. ∎

3.5 Pseudocode and an Example

The example in this section has been carefully chosen to illustrate all possible situations that may occur in the construction of the PQ tree. Consider the CDM and its collection of d-splits on page 42. The PQ tree representing the set of Kalmanson permutations of the matrix is given on the bottom of page 43. The ovals represent P nodes and the rectangles represent Q nodes. The structure of the tree is represented by inclusion.

A pictorial description of the run of the algorithm on D is given on page 45. The k^{th} row of numbers represents the collection of buckets before the k_{th} collapse. Ovals represent buckets which are labeled P and rectangles represent buckets labeled Q. Buckets corresponding to leaves contain a single index and are not shown. In the discussion of the algorithm's execution on D we use $B_{i,j}$ to represent the bucket with $m(B) = i$ and $M(B) = j$.

When the algorithm starts, it sets $y = 1$ and $B_x = B_1$ and checks to see if any consecutive buckets may be exchanged. First, the algorithm discovers $T[1, 2]$ and $T[2, 3]$ are optimal, but $T[3, 4]$ is not. With this information, the algorithm knows to collapse buckets B_1, B_2, and B_3 into a single bucket $B_{1,3}$ which is labeled P. (row 2)

Next, the algorithm discovers that $T[4, 5]$ is optimal, but $T[5, 6]$ is not and collapses B_4 and B_5 into $B_{4,5}$ which is labeled P. (row 3)

Next, the algorithm must check if $B_{1,3}$ and $B_{4,5}$ can be merged. Since $T[1, 5]$ is optimal and $T[1, 6]$ is not, the algorithm creates the bucket $B_{1,5}$ which is labeled P. This causes the algorithm to run its cleanup step. The

Build B_i for each $i \in S$, set $B_x = B_1$ and $k = 1$.

While $S(B_x)$ is not null.

  If $T[m(B_x), M(S(B_x))]$ is optimal.

    Set $B_y = B_x$ and set $B_x = S(B_x)$.

    While $T[m(B_x), M(S(B_x))]$ is optimal.

      Set $B_x = S(B_x)$.

    Collapse B_y through B_x with a P collapse.

    Run the P collapse clean up proceedure.

  Set $B_x = B$.

While $S^k(B_1)$ is not null and $|\mathcal{L}| > 1$.

  Set $B_x = B_1$ and increment k.

  While $S^k(B_x)$ is not null

    If $T[m(B_x), M(S^k(B_x))]$ is optimal.

      Collapse B_x through $S^k(B_x)$ with a Q collapse.

      Run the Q collapse clean up proceedure.

      Set $B_x = B$.

    Else, set $B_x = S(B_x)$

If $|\mathcal{L}| > 2$, collapse the remaining buckets with a Q collapse.

If $|\mathcal{L}| = 2$, collapse the remaining buckets with a P collapse.

Algorithm 3.1: Building the P-Q tree of Kalmanson permutations.

description of the algorithm says that we have to check the optimality of the tour $T[3,5]$. Since this tour is optimal, we have to delete $B_{1,3}$ and make B_1, B_2 and B_3 children of $B_{1,5}$. (row 4)

The algorithm continues, checking $T[1,6]$, $T[6,7]$, $T[7,8]$, $T[8,9]$, $T[9,10]$, $T[10,11]$, $T[11,12]$ and finding them all to be sub-optimal. The algorithm then notes that $S(B_{12})$ is null, increments y, and sets $B_x = B_{1,5}$. The algorithm begins checking the tours $T[B_x, S^2(B_x)]$ until it finds one that is optimal. $T[1,7]$, and $T[6,8]$ are not optimal, but $T[7,9]$ is. Therefore, the algorithm collapses B_7, B_8, and B_9 into the bucket $B_{7,9}$ which is labeled Q. (row 5)

After the Q collapse, the algorithm must be recursively called to restore the distance property. The recursive call occurs on the subset of buckets $\{B_6, B_{7,9}, B_{10}\}$. Since $T[6,9]$ is optimal and $T[7,10]$ is not, B_6 and $B_{7,9}$ are collapsed into $B_{6,9}$ which is labeled P. (row 6)

The recursive call must now check to see if $T[1,9]$ is optimal. $T[1,9]$ is optimal, so $B_{1,5}$ and $B_{6,9}$ are merged into a P node. (row 7)

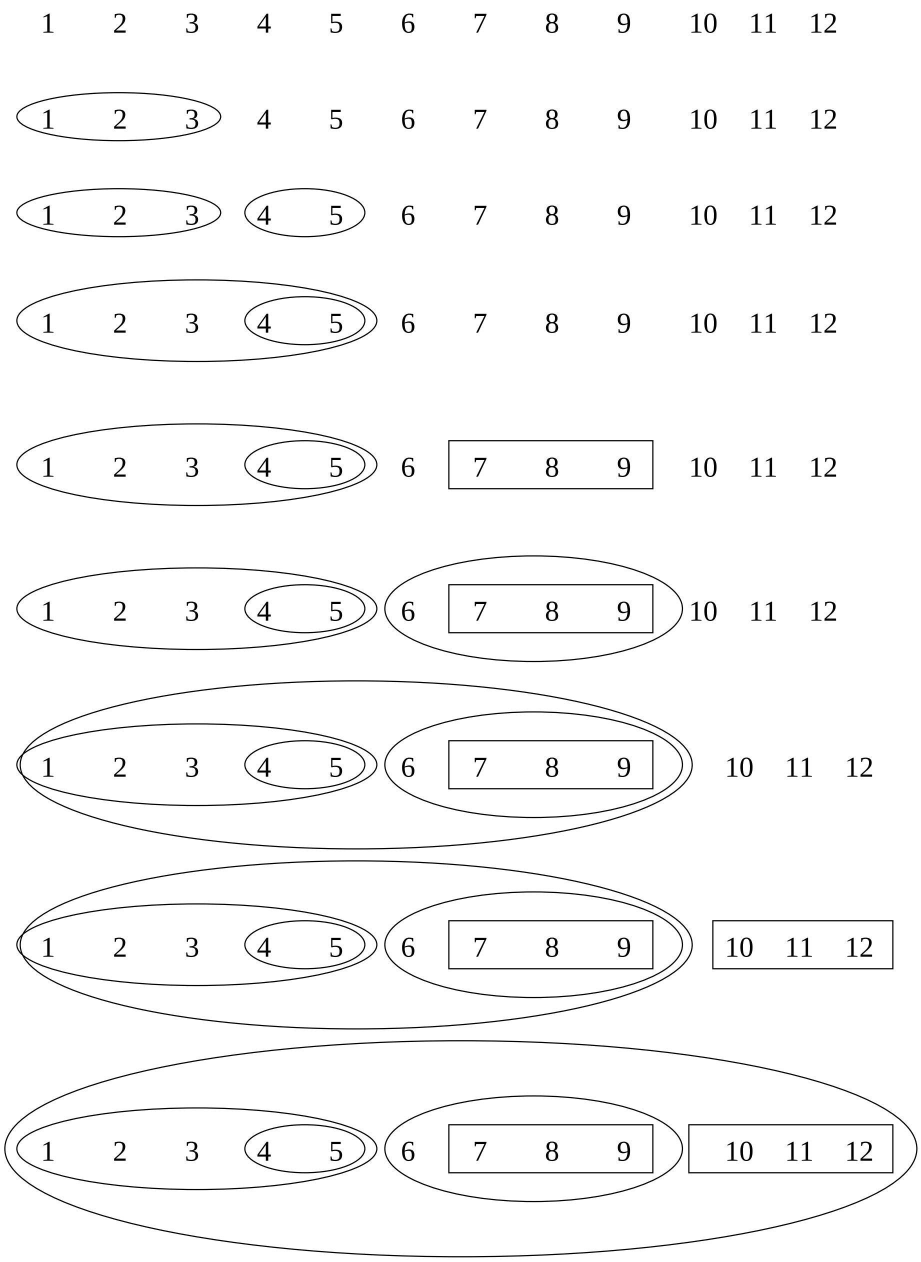
The cleanup step realizes that $T[4,9]$ is not optimal, so no cleanup is necessary. The recursive call now returns control to the main algorithm. When control has been returned to the main algorithm, is sets $B_x = B_{6,9}$ and continues checking tours. It notices that $T[6,11]$ is sub-optimal. $T[10,12]$ is optimal, so buckets B_{10}, B_{11} and B_{12} are collapsed into $B_{10,12}$ and another recursive call occurs. (row 8)

This recursive call realizes that $T[1,10]$ is optimal so the remaining two buckets are collapsed into $B_{1,12}$ which is labeled P. The cleanup phase following the P collapse realizes that $T[6,12]$ is optimal and removes $B_{1,9}$ adding its children ($B_{1,5}$ and $B_{6,9}$) to the list of children of $B_{1,12}$. (row 9)

Phase one of the algorithm halts at this step and phase two builds a P-Q tree from the buckets.

$$
\begin{bmatrix}
0 & 2 & 1 & 3 & 4 & 4 & 10 & 9 & 6 & 3 & 7 & 4 \\
2 & 0 & 3 & 5 & 6 & 6 & 12 & 11 & 8 & 5 & 9 & 6 \\
1 & 3 & 0 & 4 & 5 & 5 & 11 & 10 & 7 & 4 & 8 & 5 \\
3 & 5 & 4 & 0 & 5 & 7 & 13 & 12 & 9 & 6 & 10 & 7 \\
4 & 6 & 5 & 5 & 0 & 8 & 14 & 13 & 10 & 7 & 11 & 8 \\
4 & 6 & 5 & 7 & 8 & 0 & 8 & 7 & 4 & 3 & 7 & 4 \\
10 & 12 & 11 & 13 & 14 & 8 & 0 & 7 & 10 & 9 & 13 & 10 \\
9 & 11 & 10 & 12 & 13 & 7 & 7 & 0 & 5 & 8 & 12 & 9 \\
6 & 8 & 7 & 9 & 10 & 4 & 10 & 5 & 0 & 5 & 9 & 6 \\
3 & 5 & 4 & 6 & 7 & 3 & 9 & 8 & 5 & 0 & 4 & 3 \\
7 & 9 & 8 & 10 & 11 & 7 & 13 & 12 & 9 & 4 & 0 & 3 \\
4 & 6 & 5 & 7 & 8 & 4 & 10 & 9 & 6 & 3 & 3 & 0
\end{bmatrix}
$$

$$
\begin{array}{rclrcl}
\alpha_2 & = & 2 & \alpha_{11} & = & 2 \\
\alpha_3 & = & 1 & \alpha_{1,2,3,4,5} & = & 2 \\
\alpha_4 & = & 2 & \alpha_{\{3,4\}} & = & 1 \\
\alpha_5 & = & 3 & \alpha_{6,7,8,9} & = & 1 \\
\alpha_6 & = & 1 & \alpha_{\{7,8\}} & = & 3 \\
\alpha_7 & = & 4 & \alpha_{\{8,9\}} & = & 2 \\
\alpha_8 & = & 1 & \alpha_{\{10,11\}} & = & 1 \\
\alpha_9 & = & 1 & \alpha_{\{11,12\}} & = & 2
\end{array}
$$


1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 4 5 6 7 8 9 10 11 12

Chapter 4

Recognition Algorithms

Before we can build practical algorithms which take advantage of the structure of the classes of metrics defined in chapter 2, we need to develop efficient algorithms for recognizing these classes. Efficient algorithms are already known for recognizing tree metrics and CDMs. Unfortunately, the $O(n^5)$ algorithm of Bandelt and Dress [8] is the best known algorithm for recognizing TDMs. The important characteristics of these algorithms is presented in section 1.

With the hope of finding a faster TDM recognition algorithm, a general algorithm is presented in section 2. Unfortunately, this algorithm is too powerful to recognize TDMs. Fortunately, it can decompose a TDM in $O(n^4)$ time. Additionally, slight modifications of it give a recognition algorithm for 2-tree metrics and taxi metrics. These modifications are discussed in sections 3 and 4. Finally, section 5 summarizes current recognition algorithms and considers how to speed them up.

4.1 Existing Algorithms

There are many $O(n^2)$ algorithms for recognizing tree metrics. Boesch [14] uses the interesting idea of finding the set L of indices S which correspond to leaves in the tree representation of D, deleting those rows and columns from D and recursively finding the tree representation of the the sub metric of D induced by $S - L$. Other $O(n^2)$ algorithms for recognizing tree metrics have been developed by Bagatelj, Pisanski, and Simões-Pereira [5] and Culberson and Rudnick [20]. Developing fast heuristics for constructing a tree representation of a tree metric D is an active area of research.

Martin Farach has developed an $O(n^2)$ algorithm for recognizing CDMs. His approach is to fix an index $x \in S$ build a new matrix A from D where

$$A[i,j] = D[i,x] + D[j,x] - D[i,j]$$

He shows that there will be a minimum spanning tree T of S (using the matrix A as the cost function) which is a path if D is a CDM. In addition, T has x as a leaf and every d-split corresponds to a sub-path of T. This is enough structure to enable the algorithm to calculate the decomposition of D in $O(n^2)$ time.

In contrast to the fast recognition algorithm for CDMs, the fastest known recognition algorithm for TDMs is slow. Bandelt and Dress prove the following theorem on the structure of TDMs and apply it to get an $O(n^5)$ algorithm for TDM recognition.

Theorem 4.1.1 *The following conditions are equivalent for an $n \times n$ metric D:*

1. *D is a TDM.*
2. $\alpha_{\{t,u\},\{v,w\}} = \alpha_{\{t,u,x\},\{v,w\}} + \alpha_{\{t,u\},\{v,w,x\}}$ *for all $t, u, v, w, x \in S$*

Their recognition algorithm simply verifies that the second condition holds. They also use this approach to get an algorithm which computes the decomposition of a TDM. The decomposition algorithm works as follows. For each set of indices $\{a_1, a_2, a_3, a_4\} \subset S$ with $\alpha_{\{a_1,a_2\},\{a_3,a_4\}} \geq 0$, check that the identity

$$\alpha_{\{a_1,a_2\},\{a_3,a_4\}} = \alpha_{\{a_1,a_2,x\},\{a_3,a_4\}} + \alpha_{\{a_1,a_2\},\{a_3,a_4,x\}}$$

holds for all $x \in S - \{a_1, a_2, a_3, a_4\}$. If it does, calculate the two sets

$$A := \{x \in S | \alpha_{\{a_1,a_2,x\},\{a_3,a_4\}} = 0\}$$

$$B := \{x \in S | \alpha_{\{a_1,a_2\},\{a_3,a_4,x\}} = 0\}$$

If $A \cup B = S$ then $\alpha_{(A,B)}$ should be set to $\alpha_{\{a_1,a_2,x\},\{a_3,a_4\}}$.

4.2 A General Decomposition Algorithm

In this section we present the algorithm Decomp for decomposing a metric into a sum of weighted cut metrics. Decomp receives a symmetric distance matrix, D, as input and either rejects D or returns a collection $\mathcal{C}$ of weighted cuts. Three claims are made about Decomp:

1. If D is not rejected, then $\mathcal{C}$ is a decomposition of D.
2. If D is a TDM then $\mathcal{C}$ is the total decomposition of D.
3. Decomp will halt in $O(max\{n^3, n^2|\mathcal{C}|\})$ time.

Recall Bandelt and Dress's definition of the isolation index of a cut.

$$\beta_{a,a',b,b'} = \max\{ab + a'b', ab' + ba', aa' + bb'\} \tag{4.1}$$

$$\min\{\frac{1}{2}(\beta_{a,a',b,b'} - aa' - bb')\} = \alpha_{A,B} \tag{4.2}$$

where the minimum is taken over all (not necessarily distinct) $a, a' \in A$ and $b, b' \in B$. $\alpha_{A,B}$ is defined for all pairs of disjoint subsets of S. When (A, B) is a cut of S, $\alpha_{A,B}$ is called the **isolation index** (A, B).

Decomp begins by calculating the isolation indices of the trivial cuts using formulas (4.1) and (4.2) as a preprocessing step. Decomp sets D equal to

$$D - \sum \alpha_{\{x\}} D_{\{x\}}$$

After handling the trivial cuts, Decomp arbitrarily fixes an index $x \in S$ and iteratively tries to find a minimal cut around x.

Definition 4.2.1 *Given a collection of cuts $\mathcal{C}$ on a set $S = \{1, ..., n\}$ and an element $x \in S$, the cut $(C, \overline{C}) \in \mathcal{C}$ is called a* **minimal cut around** *x if the shore of $(C, \overline{C})$ containing x does not properly contain any shore of any other cut in $\mathcal{C}$.*

Definition 4.2.2 *Let $\mathcal{C}$ and S be as described in definition 4.2.1. If there exists an $x \in S$ such that $(C, \overline{C})$ is a minimal cut around x, then $(C, \overline{C})$ is called a* **minimal cut** *of $\mathcal{C}$.*

The minimal cut around x is found by finding a maximal subset T of S such that the cut $(\{x\}, T - \{x\})$ of D^T has positive isolation index $\alpha^T_{\{x\}}$. Once T has been found, Decomp assigns the weight $\alpha_{T'} = \alpha^T_{\{x\}}$ to the cut $(S - T', T')$ where $T' = T - \{x\}$. Although these cuts are not necessarily d-splits when D is not a TDM, we denote the weight Decomp assigns to the cut $(C, \overline{C})$ by α_C. $\alpha_{T'} D_{T'}$ is subtracted from D, and this step is repeated. Decomp halts when any of the following become true.

- D is the zero metric.
- $D[i, j] < 0$ for some $i, j \in S$.
- Decomp can not find a non-trivial cut around x.

In the first case, Decomp has found a decomposition of D and returns a list of the cuts and their weights. Otherwise, Decomp rejects D. This halting condition implies the first claim that we made about Decomp: it either returns a collection of weighted cut metrics whose sum is D or it rejects D. Since Decomp has already calculated the isolation indices of all trivial cuts, the third halting condition is added to prevent Decomp from incorrectly adding a trivial cut to its collection of cuts.

To complete the description of Decomp, we describe how Decomp calculates T and $\alpha_{T'}$. Theorem 2.1.8 provides the motivation for this step. Decomp finds a minimal cut around x by looking for a maximal set T such that $\{T - \{x\}, \{x\})$ has positive isolation index in D^T. This implies that if D is a TDM, then Decomp can compute $\alpha_{T'}$ simply by calculating the isolation index of the trivial cut around x in the metric D^T.

The first step in finding a minimal cut around x is initializing $T = S$ and constructing the matrix A with

$$\begin{array}{rcll} A[i,j] & = & D[x,i] + D[x,j] - D[i,j] & i, j, x \text{ distinct} \\ A[i,j] & = & \infty & \text{otherwise} \end{array}$$

The definition of α^T_x implies that $\alpha^T_x > 0$ if and only if $A^T[i, j] > 0$ for all distinct i and j in T. Therefore, Decomp finds a maximal T such that $A^T[i, j] \neq 0$ for all distinct $i, j \in T$.

Decomp finds T in two phases. In phase one, indices are removed from T until $A^T[i, j] > 0$ for all distinct i and j in T. This can be done efficiently by searching lexicographically through A for a zero entry, skipping the entry

$A[i,j]$ if either i or j has already been removed from T. Once a zero entry $A[i,j]$ has been found, removal of indices is carried out according to the following rules. If $A[i,k]=0$ for all $k \in T$ with $k \neq x$, remove i from T and continue searching. If there is a $k \in T$ distinct from i, j, and x such that $A[i,k] \neq 0$ remove j from T and continue searching. If $A[i,l]$ was found to be 0 while searching for k, also remove l from T.

Once phase one is complete, Decomp enters phase two and attempts to return vertices to T. Decomp selects any $k \in S-T$ and if $A[k,t] > 0$ for all $t \in T$, sets $T = T \cup \{k\}$. This process is repeated for each $k \in S-T$. When phase two is completed, T satisfies the conditions:

- $A^T[i,j] \neq 0$ for distinct i,j.
- For any $y \in S-T$, there exists an $i \in T \cup \{y\}$ such that $A^{T\cup\{y\}}[i,y]=0$.

Decomp sets

$$\alpha_{T'} = \frac{1}{2} \min_{i,j \in T'} A[i,j]$$

and stores the cut $(T', S-T')$ along with $\alpha_{T'}$. The iteration is completed by setting

$$D = D - \alpha_{T'} D_{T'}$$

This concludes the statement of the algorithm. Pseudo-code for Decomp is given on the next page. Now we turn to the proofs of the three claims made about Decomp. The first claim follows immediately from the description of the algorithm.

Theorem 4.2.3 *When Decomp is run on the input matrix D, Decomp either rejects D or returns a decomposition of D.*

Proof: This follows immediately from the halting conditions given in the description of Decomp. If the first condition causes termination of the algorithm, Decomp will return a decomposition of D, otherwise Decomp rejects D. ∎

The second claim follows easily from theorem 2.1.8.

Compute α_C for all trivial cuts in D.

Set $D = D - \sum_{y \in S} \alpha_{\{y\}} D_{\{y\}}$

Fix an index x in S_D.

While D is not the zero metric

  Compute A.

  Find a minimal cut $(C, \overline{C})$ around x.

  If $(C, \overline{C})$ is not found or trivial

    Reject D.

  Otherwise

    Store $(C, \overline{C})$ and α_C.

    Set $D = D - \alpha_C D_C$.

  If $D[i,j] < 0$ for any $i, j \in S_D$

    Reject D.

Algorithm 1: Decomp

Theorem 4.2.4 *When run on a TDM, D, Decomp returns the total decomposition of D.*

Proof: The proof works by induction on the number of non-trivial d-splits. If there are no non-trivial d-splits, D is the sum of weighted trivial cut metrics. The isolation indices of these cuts are found in the pre-processing step, and D is set to $D - \alpha_{T'} D_{T'}$ which is the zero metric. Therefore, Decomp halts and returns the total decomposition of D.

Assume that the algorithm returns the total decomposition of any TDM with k or fewer non-trivial d-splits. Let D be a TDM with $k+1$ non-trivial d-splits. Decomp chooses T so that D^T is a maximal sub-metric of D

with $\alpha^T_{\{x\}} > 0$. This along with theorem 2.1.8 implies that that $\alpha^T_x = \alpha_{T'}$. Thus, the algorithm correctly computes the isolation index of $(T', S - T')$ and $D = D - \alpha_{T'} D_{T'}$ has k non-trivial d-splits. The result now follows by induction. ∎

Now that we know that Decomp works, we consider its run time.

Theorem 4.2.5 *Decomp runs in $O(\max\{n^3, n^2|\mathcal{C}|\})$ time.*

Proof: The preprocessing step calculates the isolation indices of the trivial cut in $O(n^2)$ time per cut, for a total of $O(n^3)$ time. In each iteration of the main subroutine, Decomp computes A, finds T, calculates $\alpha_{T'}$, preforms a matrix subtraction, and checks the termination conditions. Since x is fixed, A can be calculated in $O(n^2)$ time. Entries of A are examined at most twice while calculating T; therefore finding T takes $O(n^2)$ time. Clearly the other steps also take $O(n^2)$ time. Therefore each iteration takes $O(n^2)$ time and Decomp halts in $O(\max\{n^3, n^2|\mathcal{C}|\})$ time. ∎

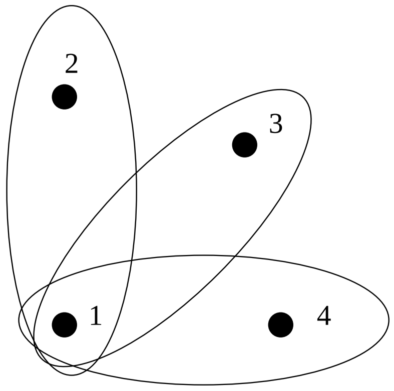
In general Decomp is slow, since $|\mathcal{C}|$ could be large. However, it becomes practical when $|\mathcal{C}|$ is bounded. Bandelt and Dress [8] have shown that every TDM has $O(n^2)$ d-splits. Combining this with Theorem 4.2.5 and Theorem 4.2.4 yields the following corollary.

Corollary 4.2.6 *Decomp finds a decomposition of a TDM in $O(n^4)$ time.*

Decomp beats Bandelt and Dress' [8] $O(n^5)$ algorithm for decomposing TDMs. Unfortunately, it does not recognize TDMs. For example, the metric D_1 in Figure 4.1 can be decomposed into the sum of the three cut metrics represented by ovals. Although D_1 is not a TDM, Decomp finds the decomposition of D_1 as long as it chooses $x \neq 1$.

There are many decomposable metrics that Decomp can not recognize. D_2 in Figure 4.1 is such a metric. Each edge in the graph represents one shore of a cut, thus $D[i, j]$ is the number of edges incident to exactly one of i and j. When run on D_2, Decomp will incorrectly conclude that each trivial cut has isolation index 1. After this mistake, Decomp will halt during the first iteration of its main loop and reject D because it can not find a minimal cut around x.

$$D_1 = \begin{bmatrix} 0 & 2 & 2 & 2 & 3 \\ 2 & 0 & 2 & 2 & 1 \\ 2 & 2 & 0 & 2 & 1 \\ 2 & 2 & 2 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{bmatrix}$$



$$D_2 = \begin{bmatrix} 0 & 4 & 6 & 4 & 6 & 4 \\ 4 & 0 & 4 & 6 & 4 & 6 \\ 6 & 4 & 0 & 4 & 6 & 4 \\ 4 & 6 & 4 & 0 & 4 & 6 \\ 6 & 4 & 6 & 4 & 0 & 4 \\ 4 & 6 & 4 & 6 & 4 & 0 \end{bmatrix}$$

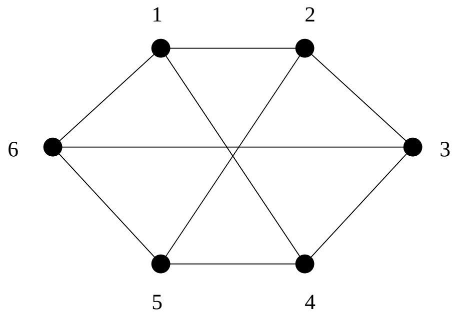


Figure 4.1

We close this section with an example of how Decomp decomposes a matrix. Let the matrix D given below be the input for Decomp. The non-zero isolation indices decomp computes for the trivial d-splits are $\alpha_{\{3\}} = 2$, $\alpha_{\{5\}} = 1$, and $\alpha_{\{7\}} = 3$. Next, Decomp subtracts the corresponding cut metrics from D to get D_1.

$$D = \begin{bmatrix} 0 & 2 & 6 & 4 & 5 & 2 & 4 \\ 2 & 0 & 4 & 4 & 5 & 2 & 6 \\ 6 & 4 & 0 & 4 & 7 & 6 & 8 \\ 4 & 4 & 4 & 0 & 3 & 4 & 6 \\ 5 & 5 & 7 & 3 & 0 & 3 & 7 \\ 2 & 2 & 6 & 4 & 3 & 0 & 6 \\ 4 & 6 & 8 & 6 & 7 & 6 & 0 \end{bmatrix} \quad D_1 = \begin{bmatrix} 0 & 2 & 4 & 4 & 4 & 2 & 1 \\ 2 & 0 & 2 & 4 & 4 & 2 & 3 \\ 4 & 2 & 0 & 2 & 4 & 4 & 3 \\ 4 & 4 & 2 & 0 & 2 & 4 & 3 \\ 4 & 4 & 4 & 2 & 0 & 2 & 3 \\ 2 & 2 & 4 & 4 & 2 & 0 & 3 \\ 1 & 3 & 3 & 3 & 3 & 3 & 0 \end{bmatrix}$$

Decomp next enters its main loop by fixing $x = 1$. First it calculates A_1. It then looks at row 2 and removes 7 from T. Decomp finds that A^T has no zero entries and sets $T = \{1, 2, 3, 4, 5, 6\}$ and $\alpha_{\{1,7\}} = 1$ in the decomposition of D. D_{T^c} is subtracted from D_1 to get D_2.

$$A_1 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 2 & 1 & 1 & 1 & 0 \\ \infty & 2 & \infty & 3 & 2 & 1 & 1 \\ \infty & 1 & 3 & \infty & 3 & 1 & 1 \\ \infty & 1 & 2 & 3 & \infty & 2 & 1 \\ \infty & 1 & 1 & 1 & 2 & \infty & 0 \\ \infty & 0 & 1 & 1 & 1 & 0 & \infty \end{bmatrix} \quad D_2 = \begin{bmatrix} 0 & 1 & 3 & 3 & 3 & 1 & 1 \\ 1 & 0 & 2 & 4 & 4 & 2 & 2 \\ 3 & 2 & 0 & 2 & 4 & 4 & 2 \\ 3 & 4 & 2 & 0 & 2 & 4 & 2 \\ 3 & 4 & 4 & 2 & 0 & 2 & 2 \\ 1 & 2 & 4 & 4 & 2 & 0 & 2 \\ 1 & 2 & 2 & 2 & 2 & 2 & 0 \end{bmatrix}$$

Decomp repeats this process, finding A_2 below. This time it removes 4, 5, 6, and 7 from T when it looks at row 2. Then A^T contains no zeros, so $T = \{1, 2, 3\}$ and $\alpha_{\{2,3\}}$ is set to 1 and the corresponding cut metric is subtracted to get D_3.

$$A_2 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 2 & 0 & 0 & 0 & 0 \\ \infty & 2 & \infty & 4 & 2 & 0 & 2 \\ \infty & 0 & 4 & \infty & 4 & 0 & 2 \\ \infty & 0 & 2 & 4 & \infty & 2 & 2 \\ \infty & 0 & 0 & 0 & 2 & \infty & 0 \\ \infty & 0 & 2 & 2 & 2 & 0 & \infty \end{bmatrix} \quad D_2 = \begin{bmatrix} 0 & 0 & 2 & 3 & 3 & 1 & 1 \\ 0 & 0 & 2 & 3 & 3 & 1 & 1 \\ 2 & 2 & 0 & 1 & 3 & 3 & 1 \\ 3 & 3 & 1 & 0 & 2 & 4 & 2 \\ 3 & 3 & 3 & 2 & 0 & 2 & 2 \\ 1 & 1 & 3 & 4 & 2 & 0 & 2 \\ 1 & 1 & 1 & 2 & 2 & 2 & 0 \end{bmatrix}$$

A_3 is calculated and shown below. This time, Decomp looks at row 2 of A and removes 2 from T. When row three is considered, 6 is removed from T and the matrix A^T contains no zeros. Therefore, $\alpha_{\{1,2,6\}}$ is set to 1 and subtracted from D_2 to get D_3

$$A_3 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 0 & 0 & 0 & 0 \\ \infty & 0 & \infty & 4 & 2 & 0 & 2 \\ \infty & 0 & 4 & \infty & 4 & 0 & 2 \\ \infty & 0 & 2 & 4 & \infty & 2 & 2 \\ \infty & 0 & 0 & 0 & 2 & \infty & 0 \\ \infty & 0 & 2 & 2 & 2 & 0 & \infty \end{bmatrix} \quad D_2 = \begin{bmatrix} 0 & 0 & 1 & 2 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 & 2 & 1 & 0 \\ 1 & 1 & 0 & 1 & 3 & 2 & 1 \\ 2 & 2 & 1 & 0 & 2 & 3 & 2 \\ 2 & 2 & 3 & 2 & 0 & 1 & 2 \\ 1 & 1 & 2 & 3 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 & 2 & 1 & 0 \end{bmatrix}$$

This process can be repeated until D_8 is the zero metric. This gives the isolation indices of all the d-splits.

$$
\begin{aligned}
\alpha_{\{3\}} &= 2 & \alpha_{\{2,3\}} &= 1 \\
\alpha_{\{5\}} &= 1 & \alpha_{\{3,4\}} &= 1 \\
\alpha_{\{7\}} &= 3 & \alpha_{\{4,5\}} &= 1 \\
\alpha_{\{2,3\}} &= 1 & \alpha_{\{5,6\}} &= 1 \\
\alpha_{\{1,7\}} &= 1 & \alpha_{\{1,2,6\}} &= 1
\end{aligned}
$$

4.3 Recognizing 2-Tree Metrics

In this section Decomp is modified to give the algorithm 2-Tree Rec for recognizing 2-tree metrics. Lemma 4.3.2 below characterizes what happens when Decomp is run on a 2-tree metric. As we will see in its proof, this theorem follows from this corollary of the work of Bandelt and Dress [8].

Corollary 4.3.1 *(Bandelt and Dress [8]) A metric D is a 2-tree metric if and only if it is a TDM and the collection of d-splits $\mathcal{C}$, can be partitioned into two laminar collections $\mathcal{C}_1$ and $\mathcal{C}_2$.*

Lemma 4.3.2 *A metric D is a 2-tree metric if and only if Decomp returns a decomposition of D which consists of a collection $\mathcal{C}$ of cuts which can be partitioned into two laminar collections $\mathcal{C}_1$ and $\mathcal{C}_2$.*

Proof: This theorem of Bandelt and Dress describes the structure of 2-tree metrics.

Combining corollary 4.3.1 and theorem 4.2.4 shows that Decomp behaves as the theorem claims when it is run on a 2-tree metric.

Notice that if $\mathcal{C}$ is the union of two laminar collections of cuts, then $\mathcal{C}$ is a weakly compatible collection of cuts. This combined with theorem 2.1.7 and corollary 4.3.1 implies that when Decomp runs as described in the statement of the theorem, D is a 2-tree metric. ∎

Lemma 4.3.2 allows us to modify Decomp to recognize 2-tree metrics. 2-Tree Rec first runs Decomp with the restriction that Decomp will automatically reject D if the main loop runs for more than $2n-2$ iterations. 2-Tree Rec rejects D if the modified version of Decomp rejects D. Otherwise, 2-Tree Rec tries to partition $\mathcal{C}$ into 2 laminar collections of cuts. If this can be done, 2-Tree Rec accepts D. Otherwise D is rejected.

To check if $\mathcal{C}$ can be partitioned into two laminar collections, 2-Tree Rec constructs a graph G from $\mathcal{C}$ as follows. Each vertex v_C in $V(G)$ corresponds to a cut in $\mathcal{C}$. The edge $e(V_{C^1}, V_{C^2})$ is added to E(G) if and only if the cuts $(C_1, \overline{C}_1)$ and $(C_2, \overline{C}_2)$ are not laminar. G is bipartite exactly when laminar collections $\mathcal{C}^1$ and $\mathcal{C}^2$ can be found. 2-Tree Rec applies the standard breadth first search algorithm for checking weather a graph is bipartite to G. Pseudocode for 2-Tree Rec is given on the following page.

Now we prove that Decomp behaves as claimed. This is divided into 2 theorems.

Theorem 4.3.3 *2-Tree Rec recognizes 2-tree metrics.*

Proof: It is well known that a laminar collection of cuts can contain at most $n-1$ non-trivial cuts. Therefore Decomp will decompose and accept any 2-tree metric with $2n-2$ iterations of its main loop. The theorem follows from this observation and corollary 4.3.1. ∎

Theorem 4.3.4 *2-Tree Rec recognizes 2-tree metrics in $O(n^3)$ time.*

Proof: Since Decomp will halt after finding $2n-2$ non-trivial cuts, the decomposition phase of the algorithm will take $O(n^3)$ time. Deciding if two cuts are laminar takes $O(n)$ time. Building G requires $O(n^2)$ of these comparisons, for a total of $O(n^3)$ time. Finally, we can check if G is bipartite in $O(n)$ time. Therefore total run time is $O(n^3)$. ∎

This section ends with an example of 2-Tree Rec. Consider the matrix D below. We will assume that Decomp has already calculated the isolation indices which are also given below.

2-tree rec next builds the graph G. (see Figure 4.2) We have omitted the trivial cuts from G since they are laminar with any cut and correspond to isolated vertices in G. The vertices $v1, v2, v3, v4$ in the upper diagram represent the cuts $\{1,2,4\}, \{4,6\}, \{6,7\}, \{1,2,7\}$ respectively. It is easy to see that G is bipartite with bipartition $\{v1, v3\}, \{v2, v4\}$.

$$D = \begin{bmatrix} 0 & 2 & 5 & 5 & 5 & 7 & 6 \\ 2 & 0 & 3 & 3 & 3 & 5 & 4 \\ 5 & 3 & 0 & 4 & 2 & 4 & 5 \\ 5 & 3 & 4 & 0 & 4 & 4 & 7 \\ 5 & 5 & 2 & 4 & 0 & 4 & 7 \\ 7 & 5 & 4 & 4 & 4 & 0 & 5 \\ 6 & 4 & 5 & 7 & 7 & 5 & 0 \end{bmatrix} \qquad \begin{array}{rcl rcl} \alpha_{\{1\}} & = & 2 & \alpha_{\{7\}} & = & 2 \\ \alpha_{\{3\}} & = & 1 & \alpha_{\{1,2,4\}} & = & 1 \\ \alpha_{\{4\}} & = & 1 & \alpha_{\{4,6\}} & = & 1 \\ \alpha_{\{5\}} & = & 1 & \alpha_{\{6,7\}} & = & 1 \\ \alpha_{\{6\}} & = & 1 & \alpha_{\{1,2,7\}} & = & 1 \end{array}$$

Run Decomp on D.

If Decomp runs for more than $2n-2$ iterations or rejects D.

Reject D.

Else.

Build a graph G from $\mathcal{C}$ as follows:

Add v_C to $V(G)$ for each $(C, \overline{C}) \in \mathcal{C}$.

If $(C_1, \overline{C}_1)$ and $(C_2, \overline{C}_2)$ are non-laminar.

Add $e(V_{C_1}, V_{C_2})$ to $E(G)$.

If G is bipartite

Accept D and return the decomposition.

Otherwise

Reject D.

Algorithm 2: 2-Tree Rec

The two trees at the bottom of Figure 4.3 are the two tree representation of D. Each edge in the representation has weight 1. This representation was chosen to illustrate that the same trivial cuts could occur in both trees of the decomposition.

4.4 Recognizing Taxi Metrics

In this section we show how Decomp can be modified to give an algorithm Taxi which recognizes taxi metrics. Like 2-Tree Rec, Taxi begins by running Decomp. D is rejected if Decomp either rejects D or has not finished decomposing D after a preset number of iterations. Taxi handles Decomp's output differently than 2-Tree Rec. Instead of building G as 2-Tree Rec does, Taxi takes a direct approach and tries to use the cuts returned by Decomp to find and embedding of D in the rectilinear plane. To do this, Taxi builds the

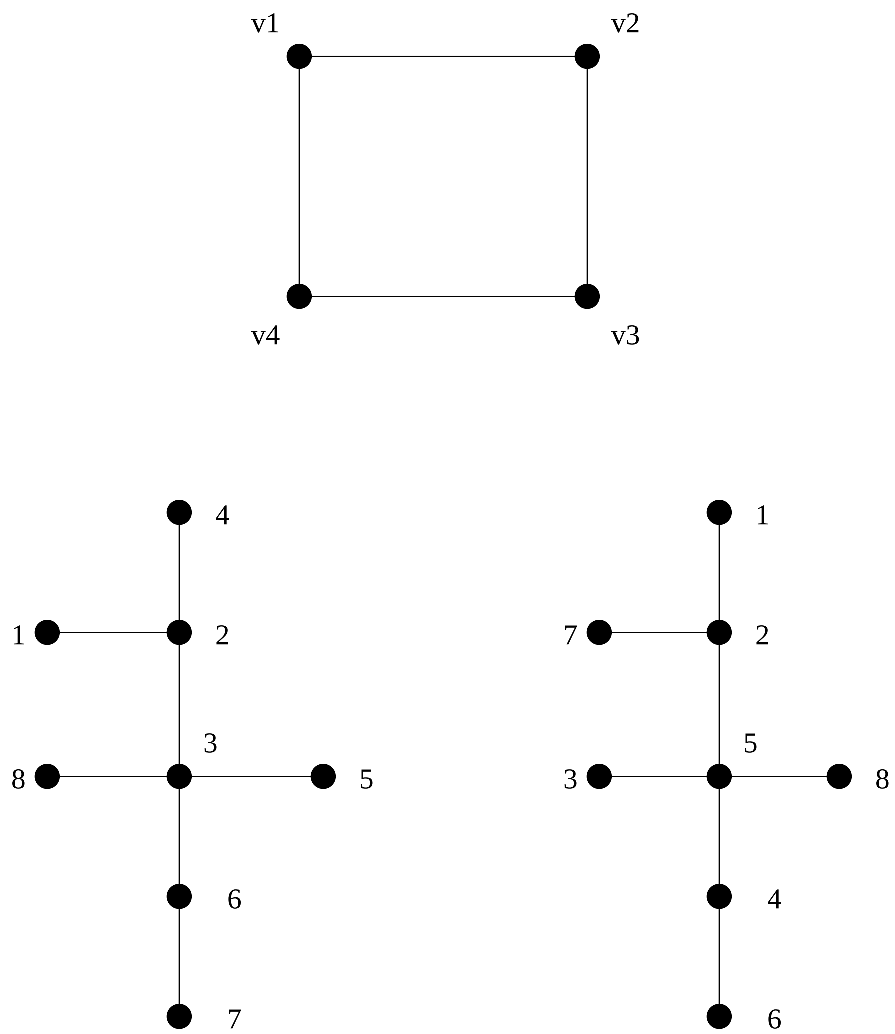


Figure 4.2:

shore lattice of $\mathcal{C}$.

Definition 4.4.1 *The* **shore lattice** *of a collection* $\mathcal{C}$ *of cuts is the lattice describing the shores of the cuts ordered by inclusion.*

Once the shore lattice has been built, Taxi uses Bandelt and Chepoi's [7] algorithmic proof of the following theorem to find an embedding of D in the rectilinear plane, if one exists.

Theorem 4.4.2 *(Bandelt and Chepoi [7]) A TDM D is a taxi metric if and only if for any sub-collection $\mathcal{S} = (C_1, \overline{C}_1), ..., (C_k, \overline{C}_k)$ of the collection of d-splits with $k \leq 5$ the shore lattice of $\mathcal{S}$ has at most four minimal elements.*

To help the reader's intuition, we state a lemma relating the d-splits of a taxi metric D to an embedding of D.

Lemma 4.4.3 *Let D be a rectilinear planar metric, and let ϕ be an embedding of D. Up to interchanging the roles of C and $\overline{C}$, every d-split is of one of two forms:*

- $\left\{ \begin{array}{lcl} C & = & \{i | \phi_x(i) \geq c \textit{ for some constant } c\} \\ \overline{C} & = & \{i | \phi_x(i) < c \textit{ for some constant } c\ \} \end{array} \right.$

- $\left\{ \begin{array}{lcl} C & = & \{i | \phi_y(i) \geq c \textit{ for some constant } c\} \\ \overline{C} & = & \{i | \phi_y(i) < c \textit{ for some constant } c\} \end{array} \right.$

Now we present the highlights of the inductive step in Bandelt and Chepoi's proof of the sufficiency of the conditions in theorem 4.4.2. (For more details see [7].) Let D be a metric with m d-splits satisfying the conditions of theorem 4.4.2. Choose any minimal d-split $(C_0, \overline{C}_0)$ and define the metric

$$D' = D - \alpha_{C_0} D_{C_0}$$

By induction, D' is a taxi metric. Let $\phi'(i) : S \to R^2$ be an embedding of D' into the rectilinear plane. It follows immediately from lemma 4.4.3 that the minimal shores of $\mathcal{C}'$ are of the form:

$$\begin{array}{lcl} C_1 & = & \{i | \phi_x(i) \geq \phi_x(j) \text{for some } j \in S\} \\ C_2 & = & \{i | \phi_x(i) \leq \phi_x(j) \text{for some } j \in S\} \\ C_3 & = & \{i | \phi_y(i) \geq \phi_y(j) \text{for some } j \in S\} \\ C_4 & = & \{i | \phi_y(i) \leq \phi_y(j) \text{for some } j \in S\} \end{array}$$

Bandelt and Chepoi [7] show that since $\mathcal{C}$ has at most four minimal shores, at least one of the following must hold.

$$\begin{array}{ll} C_0 \subset C_i & \text{for some } 1 \leq i \leq 4. \\ \overline{C_0} \subset C_i & \text{for some } 1 \leq i \leq 4. \end{array}$$

Assume that $C_0 \subset C_1$. (The other cases are similar.) To get an embedding of D, simply increase $\phi_x(i)$ by α_{C_0} for all $i \in C_0$.

Now we combine Decomp with Bandelt and Chepoi's proof to get an $O(n^3)$ algorithm for recognizing and embedding Taxi metrics. A pseudo-code description of the algorithm is given on the following page. Taxi starts by running Decomp on D, rejecting D if Decomp halts and rejects D, more that 4 trivial cuts are returned or Decomp returns more than $2n-6$ non-trivial cuts.

Next, Taxi builds the shore lattice $\mathcal{L}$ and orders the cuts in $\mathcal{C}$ so that C_i is a minimal d-split of the metric

$$D_i = \sum_{j=i}^{m} \alpha_{C_j} D_{C_j}$$

This ordering can be found by repeatedly removing a minimal shore C_i and its complement $\overline{C}$ from $\mathcal{L}$, halting when $\mathcal{L}$ is empty.

After the cuts are ordered, Bandelt and Chepoi's [7] algorithmic proof is used to find an embedding of D in the rectilinear plane or show that none exist. Start with all the indices of S mapped to the point (x, y). Consider the cut C_m. The cut metric $(C_m, \overline{C}_m)$ can be embedded in the rectilinear plane by re-mapping all of the points in C_m to the point $(x, y + \alpha_{C_m})$. Once the metric D_{i+1} has been embedded, the inductive step in the proof of theorem 4.4.2 is used to embed D_i as described above. D is rejected if at any step, C_i is not contained in one the sets

$$\begin{array}{lll} S_1 & = & \{i | \phi_x(i) \geq \phi_x(j) \text{for all } j \in S\} \\ S_2 & = & \{i | \phi_x(i) \leq \phi_x(j) \text{for all } j \in S\} \\ S_3 & = & \{i | \phi_y(i) \geq \phi_y(j) \text{for all } j \in S\} \\ S_4 & = & \{i | \phi_y(i) \leq \phi_y(j) \text{for all } j \in S\} \end{array}$$

The following theorem follows easily from the proof of theorem 4.4.2 and theorems 2.1.7 and 4.2.4.

Theorem 4.4.4 *When run on a metric D, the algorithm will either embed D in the rectilinear plane or show that this can not be done.*

Proof: Since taxi metrics are 2-tree metrics, theorems 4.2.4 and 2.1.7 guarantee that Decomp will return the correct decomposition of a taxi metric. The inductive proof of theorem 4.4.2 finds an embedding of D if D is a taxi metric. Thus, taxi finds an embedding whenever possible.

If D is not a taxi metric, either Decomp can't find a decomposition of D (hence D is rejected) or the decomposition finds a collection of d-split $\mathcal{S} = \{(C_1, \overline{C}_1), ..., (C_5, \overline{C}_5)\}$ such that the shore lattice of $\mathcal{S}$ has 5 minimal shores. In either case, D can not be a taxi matric. ∎

Theorem 4.4.5 *The algorithm runs in $O(n^3)$ time.*

Proof: Computing the cost of the trivial cuts takes $O(n^3)$ time. Since there are $O(n)$ non trivial d-splits, Decomp will halt in $O(n^3)$ time. Checking if $A \subset B$ can be done in $O(|A \cup B|)$ time, so a shore can be added to the shore lattice (which contains $O(n)$ shores) in $O(n^2)$ time. Thus it takes $O(n^3)$ time to build the shore lattice. A minimal

shore and its complement can be removed from the lattice in $O(n)$ time. Adding a cut to the embedding takes $O(n)$ time and the embedding is built from the list of minimal shores in $O(n)$ time. A total of $O(n^3)$ time. ∎

This section concludes with an example of Taxi embedding a taxi metric in the rectilinear plane. Taxi runs on the matrix D given below.

$$D = \begin{bmatrix} 0 & 2 & 5 & 6 & 4 & 2 & 2 & 5 \\ 2 & 0 & 7 & 8 & 6 & 4 & 4 & 3 \\ 5 & 7 & 0 & 3 & 3 & 3 & 3 & 4 \\ 6 & 8 & 3 & 0 & 6 & 4 & 4 & 5 \\ 4 & 6 & 3 & 6 & 0 & 4 & 4 & 7 \\ 4 & 6 & 1 & 2 & 4 & 0 & 2 & 3 \\ 2 & 4 & 3 & 4 & 4 & 2 & 0 & 3 \\ 5 & 3 & 5 & 5 & 7 & 3 & 3 & 0 \end{bmatrix}$$

Run Decomp on D.

If it runs for more than $2n-2$ iterations or rejects D.

    Reject D.

Else.

    Build the shore lattice $\mathcal{L}$.

    Set i=1

    While the shore lattice is non-empty

        Remove a minimal shore and its complement from $\mathcal{L}$.

    Embed D_m in the rectilinear plane.

    While $j \geq 1$ and D_j can be embedded in the plane.

        Extend the embedding of D_j to an embedding of D_{j-1}.

        If the embedding can be extended

            Decrement j

        Else

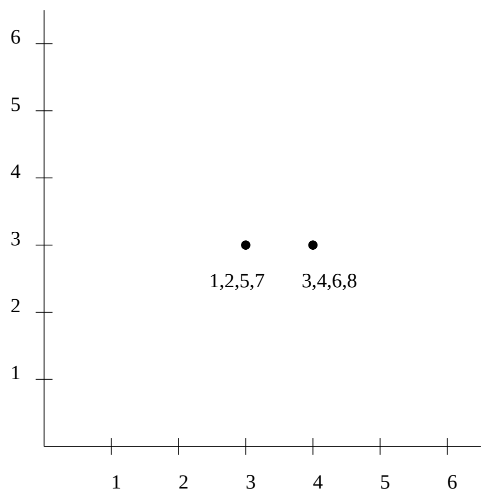
            Reject D

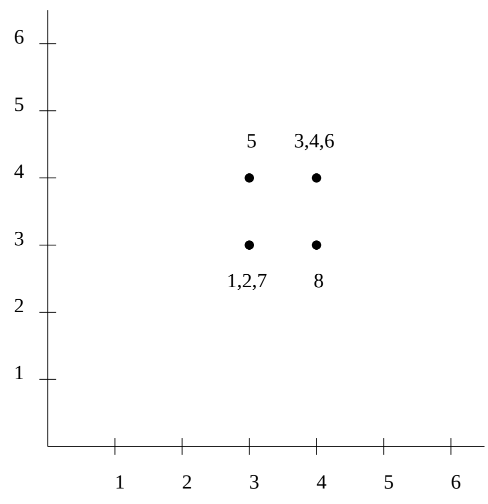
Algorithm 3: Taxi

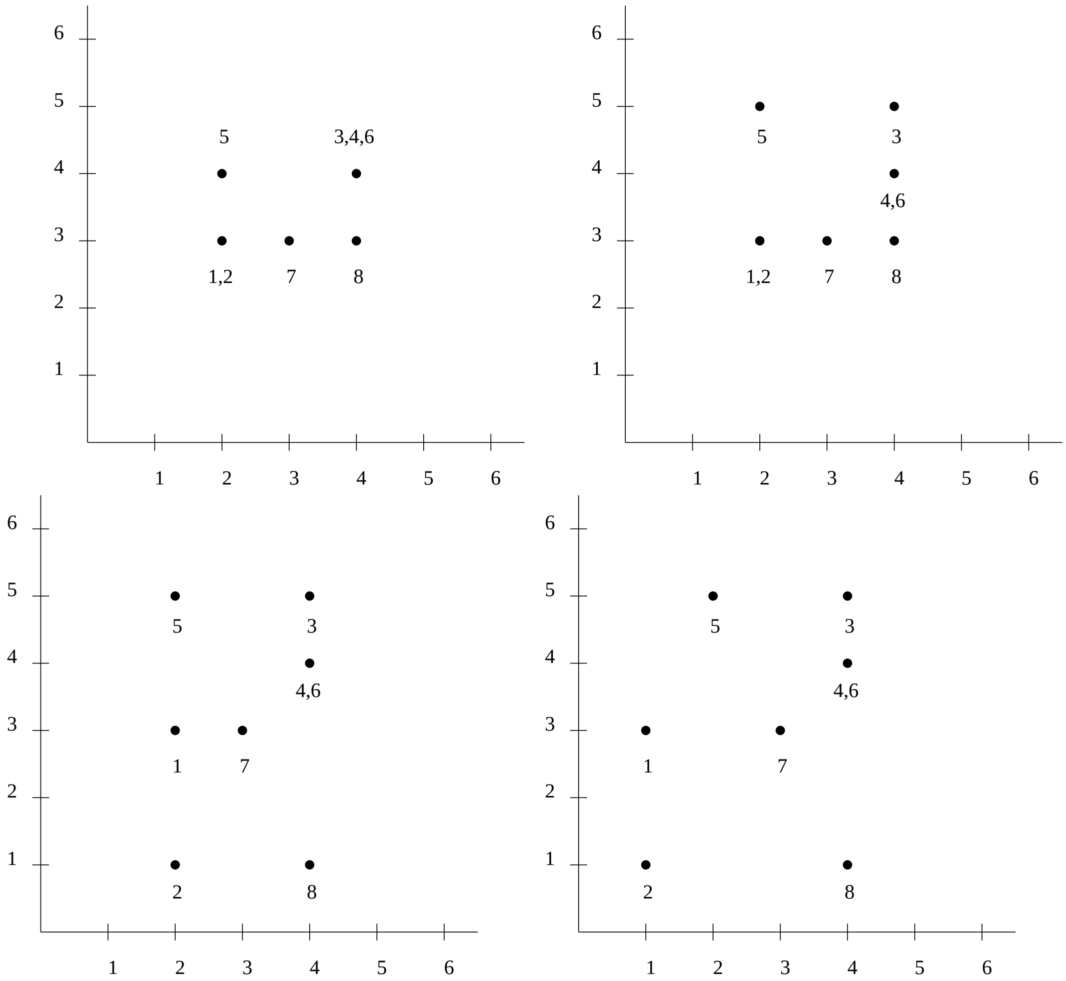
Taxi runs Decomp to calculate the isolation indices and d-splits shown below.

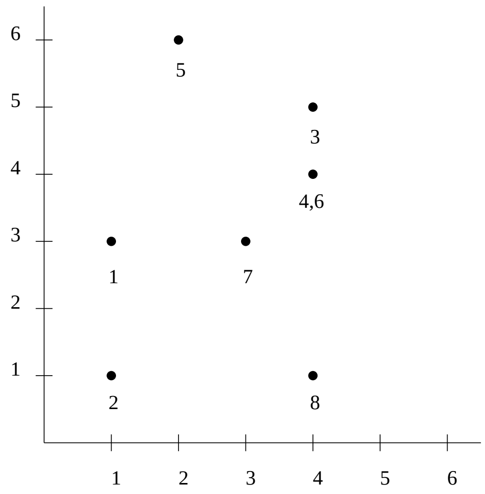
$$\begin{array}{lcllcllcllcl}
\alpha_{\{1,2\}} & = & 1 & \alpha_{\{1,2,5\}} & = & 1 & \alpha_{\{1,2,5,7\}} & = & 1 & \alpha_{\{4\}} & = & 2 \\
\alpha_{\{5\}} & = & 1 & \alpha_{\{3,5\}} & = & 1 & \alpha_{\{3,4,5,6\}} & = & 1 & \alpha_{\{2,8\}} & = & 2
\end{array}$$

Once the d-splits have been calculated, the shore lattice is built and minimal shores of the collection of d-splits are removed from the lattice in the following order: $S_1 = \{4\}$, $S_2 = \{5\}$, $S_3 = \{1,2\}$, $S_4 = \{2,8\}$, $S_5 = \{3,5\}$, $S_6 = \{1,2,5\}$, $S_7 = \{1,2,7,8\}$, $S_8 = \{1,2,5,7\}$. The embedding of the metric is illustrated below and on the following page.

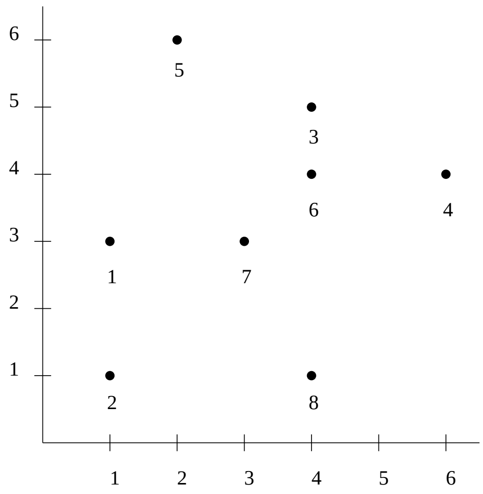





5
3,4,6
1,2
7
8
5
3
4,6
1,2
7
8
5
3
4,6
1
7
2
8
5
3
4,6
1
7
2
8



6
5
4
3
2
1
1
2
3
4
5
6
5
3
4,6
1
7
2
8



6
5
4
3
2
1
1
2
3
4
5
6
5
3
6
4
1
7
2
8

4.5 Conclusions

Here is a chart that sums up the fastest known recognition algorithms for classes of metrics in the totally decomposable hierarchy.

Metric Type	Recognition Time	Reference
TDM	$O(n^5)$	[8]
CDM	$O(n^2)$	[23]
2-Tree	$O(n^3)$	theorem 4.3.4
Taxi	$O(n^3)$	theorem 4.4.4
Tree	$O(n^2)$	[5]

There is little room for improvement in the case of CDMs and tree metrics. Perhaps 2-tree metrics can be recognized faster than by using the algorithm given above. Taxi metrics may be recognizable by an algorithm similar to 2-Tree Rec, which first finds the decomposition of a metric and then checks the structure of the decomposition. Certainly the recognition time for TDMs can be improved. One way to get an immediate improvement to $O(n^4)$ is to devise an algorithm which can recognize a weakly compatible collection $\mathcal{C}$ of cuts in $O(n^4)$time.

Chapter 5

Applications to b-matching

5.1 Introduction

An instance (D, b) of the maximum weight b-matching problem consists of an $n \times n$ metric D and non-negative integer weight function, b, on S. The goal is to solve the following integer program.

$$
\begin{array}{llll}
 & \text{Maximize} & \sum\limits_{u,v \in S} D[u,v]x(u,v) & \\
\text{(IP)} & \text{Subject to:} & \sum\limits_{u \neq v} x(u,v) \leq b(v) & \text{For all } v \in S \\
 & & x(u,v) & \text{Positive integer}
\end{array}
$$

A feasible point, x, of (IP) is called an integral b-matching. Given a feasible x, define

$$D(x) = \sum D[u,v]x(u,v)$$

Unless otherwise noted, assume that $\sum_{x \in S} b(x)$ is even.

Tamir [38] has shown how to solve the maximum weight integral b-matching problem in $O(n)$ time when D is either a tree metric or a taxi metric. In this chapter, Tamir's ideas are extended to give an $O(n)$ algorithm if D is the sum of two tree metrics. This is an improvement over Anstee's [2] $O(\min\{n^3 \log \hat{b} + n^4, n^4 \log n\})$ algorithm for the general b-matching problem. Where $\hat{b} = max\{b(i) | i \in S\}$.

The key observation in this chapter is that given an instance (D, b) of the maximum weight b-matching problem where D is a TDM there is a b-matching with $x(i, j) > 0$ only if i and j are separated by every d-split. This observation is exploited in different ways depending on the interaction of the cuts. First we discuss its use when D is a 2-tree metric, then when D is a CDM, and finally when D is a TDM. The final section of this chapter discusses a more theoretical reason why these observations are true.

5.2 Max b-matching in 2-tree metrics

Consider the primal and dual linear programming relaxations of (IP).

$$
\text{(LP)} \qquad \begin{array}{lll}
\text{Maximize} & \sum_{u,v \in S} D[u,v]x(u,v) & \\
\text{Subject to:} & \sum_{u \neq v} x(u,v) \leq b(v) & \text{For all } v \in S \\
 & x(u,v) \geq 0 &
\end{array}
$$

$$
\text{(D)} \qquad \begin{array}{lll}
\text{Minimize} & \sum_{1 \leq u \leq n} b(u)y(v) & \\
\text{Subject to:} & y(u) + y(v) \geq D[u,v] & \text{For all } u \neq v \\
 & y(v) \geq 0 &
\end{array}
$$

We use the fact that D is a 2-tree metric, to construct an integral optimal solution to (LP). Our algorithm is designed to iteratively build an optimal b-matching. In each iteration,the algorithm finds an edge, $e_{i,j}$, used in the optimal b-matching and assigns as large an integer weight to $x^*(i, j)$ as possible. We show that x^* is an optimal solution by constructing a feasible solution y^* to (D) with

$$\sum_{u,v\in S} D[u,v]x^*(u,v) = \sum_{v\in S} b(v)y^*(v)$$

Before examining the maximum weight b-matching problem on a two-tree metric, we state and prove a theorem about a simpler problem. Although this problem may seem contrived, the main subroutine of our algorithm is derived from its solution.

Let $S = \{1, 2, ..., n\}$, and let b(v) be an integer weight function on the elements of S. Suppose we are given a collection $\mathcal{S} = \{A_1, A_2, A_3, B_1, B_2, B_3\}$ of subsets of S satisfying the following properties:

$$b(A_i) \leq \frac{b(S)}{2} \tag{5.1}$$

$$b(B_i) \leq \frac{b(S)}{2} \tag{5.2}$$

$$A_i \cap A_j = \emptyset \qquad i \neq j \tag{5.3}$$

$$B_i \cap B_j = \emptyset \qquad i \neq j \tag{5.4}$$

$$\cup_i A_i = S \tag{5.5}$$

$$\cup_i B_i = S \tag{5.6}$$

Where $A(v)$ is the unique A_i such that $v \in A_i$, $B(v)$ is the unique B_j such that $v \in B_j$, and $b(T) = \sum_{t\in T} b(t)$ for all $T \subset S$.

Theorem 5.2.1 *Let $b(v)$, and $\mathcal{S}$ be as described above. If $b(S)$ is even, we can always find an integral b-matching, x, such that $x(u,v) > 0$ implies both $A(u) \neq A(v)$ and $B(u) \neq B(v)$, and*

$$\sum x(u,v) = \frac{b(S)}{2}$$

Proof: Let $b(S) = 2k$ and proceed by induction on k. The theorem obviously holds when $k = 0$. In the proof of the inductive step, we make the following assumptions without loss of generality:

- $b(A_1) \geq b(A_2) \geq b(A_3)$
- $b(v) > 0$ for all $v \in S$

To begin the proof, choose any $v \in A_1$ and relabel the B_k such that:

- $v \in B_1$
- $b(B_2) \geq b(B_3)$

If $b(B_2) = \frac{b(S)}{2}$, replace B_1 by $B_1 \cup B_3$ and B_3 by $\emptyset$. Since $b(A_1) \leq b(S)/2$, $b(B_1) \leq b(S)/2$, and $b(A_1 \cap B_1) > 0$, we can find a vertex, $u \in \overline{A_1 \cup B_1}$, with $b(u) > 0$. Notice that since $A(u) \neq A(v)$ and $B(u) \neq B(v)$, $x(v, u)$ can be positive in the b-matching we are constructing. In order to apply induction, define a new weight function b' on G as follows.

- $b'(u) = b(u) - 1$
- $b'(v) = b(v) - 1$
- $b'(w) = b(w)$ for all other $w \in S$.

By our choice of u and v, any A_i or B_j with $b(A_i) = \frac{b(S)}{2}$ or $b(B_j) = \frac{b(S)}{2}$ must contain either u or v. Therefore, the hypothesis of the theorem holds when b is replaced by b', and we can apply induction to find an integral b-matching $\overline{x}$ which satisfies the conclusion of the theorem with b replaced by b'. Construct an integral b-matching x on S from $\overline{x}$ as follows.

- $x(u, v) = \overline{x}(u, v) + 1$
- $x(w, z) = \overline{x}(w, z)$ for $(w, z) \neq (u, v)$

Notice that if $x(w, z) > 0$ and $(w, z) \neq (u, v)$ then $A(w) \neq A(z)$ and $B(w) \neq B(z)$ by induction. Recall that $A(u) \neq A(v)$ and $B(u) \neq B(v)$ by our choice of u and v. Therefore x is an integral b-matching as described in the conclusion of the theorem. ∎

The next step in constructing our algorithm is to show how to construct S from the 2-tree representation of D. To do this, we introduce the concept of the median of a tree metric.

Definition 5.2.2 *Let (D, b) be an instance of the integral b-matching problem on a tree metric. The* **median** *of D, is the index, $m \in S$, which minimizes*

$$\sum_{v \in S} b(v) D[v, m]$$

Sabidussi [37] has shown how to find the median of D in linear time. It is also well known that if T is the tree representation of D any connected component, C of $T - \{m\}$ satisfies

$$b(V(C)) <= \frac{b(V(T))}{2}$$

Given an instance of the b-matching problem on a tree metric D, Algorithm 5.1 constructs three sets of vertices, S_1, S_2, and S_3, from T such that:

- Each connected component of $T - m$ is contained in some S_i
- $b(S_i) \leq b(S)/2$
- $S_i \cap S_j = \emptyset \qquad i \neq j$
- $S_1 \cup S_2 \cup S_3 = S$

Algorithm 5.1 and the proof of Theorem 5.2.1 can be combined to give a slow, $O(b(S))$, but effective algorithm for constructing an optimal b-matching on a 2-tree metric. Given an instance (D, b) of the b-matching problem on a two-tree metric, apply Algorithm 5.1 twice to construct A_1, A_2, and A_3 from D_1 and B_1, B_2, and B_3 from D_2, where D_1 and D_2 are the two tree metrics whose sum is D. Then apply the proof of theorem 5.2.1 to construct a b-matching x^* on S.

The following theorem uses linear programming duality to prove that x^* is a maximum weight b-matching on G.

Theorem 5.2.3 *x^* is an optimal solution to (LP).*

Proof: To prove this, consider the dual linear program.

$$\begin{array}{lll} & \text{Minimize} & \sum_{1 \leq v \leq n} b(v)y(v) \\ \text{(D)} & \text{Subject to:} & y(u) + y(v) \geq D[u,v] \quad \text{For all } u \neq v \\ & & y(v) \geq 0 \end{array}$$

Find the median m of D.

Label the connected components of $T - m$ $\{C_1, ..., C_k\}$.

Let $C_{k+1} = \{m\}$.

Set $S_1 = S_2 = S_3 = \emptyset$.

For $i = 1$ to $k + 1$

  If $b(S_1) + b(C_i) \leq \frac{b(S)}{2}$, set $S_1 = S_1 \cup C_i$.

  Else

    If $b(S_2) + b(C_i) \leq \frac{b(S)}{2}$, set $S_2 = S_2 \cup C_i$.

    Else, set $S_1 = S_1 \cup C_i$.

Algorithm 5.1: Constructing S_1, S_2, and S_3 from a tree.

Construct a feasible solution to the dual as follows. Let $D_1[u, v]$ and $D_2[u, v]$ be tree metrics whose sum is D, and let m_k denote the median of D_k. For each vertex in S let

$$y^*(v) = D_1[v, m_1] + D_2[v, m_2]$$

y^* is feasible for (D), and the dual objective function value for y^* is

$$b(y^*) = \sum_{v \in S} b(v)(D_1[v, m_1] + D_2[v, m_2])$$

Recall that $x^*(u, v) = 0$ if u and v are in the same connected of $T_k - m_k$ for $k \in \{1, 2\}$. Therefore, if $x^*(u, v) > 0$ then $D_1[u, v] = D_1[u, m_1] + D_1[m_1, v]$ and $D_2[u, v] = D_2[u, m_2] + D_2[m_2, v]$. So, the total amount $x(i, j)$ contributes to the objective function must be

$$x^*(u, v)(D_1[u, m_1] + D_2[u, m_2] + D_1[v, m_1] + D_2[v, m_2])$$

If we now sum the contribution of all the $x(i, j)$ we get

$$\begin{aligned}
D(x^*) &= \sum_{(u,v)\in G} x^*(u,v)D[u,v] \\
&= \sum_{(u,v)\in G} x^*(u,v)(\sum_{k=1,2} D_k[u,m_k] + D_k[v,m_k]) \\
&= \sum_{w\in S} b(w)(D_1[w,m^1] + D_2[w,m^2]) \\
&= \sum_{w\in S} b(w)y^*(w) \\
&= b(y^*)
\end{aligned}$$

Thus x^* is an optimal b-matching. ∎

Combining Algorithm 4.1 and the proof of theorem 5.2.1 gives an $O(b(S))$ algorithm for finding a maximum weight b-matching. More care is needed to get an $O(n)$ algorithm. The improved algorithm proceeds in two steps.

Step one constructs A_1, A_2, and A_3 from D_1, and B_1, B_2, and B_3 from D_2 using Algorithm 5.1, and initializes $b'(v) = b(v)$ for all $v \in S$.

Step two repeatedly finds an edge, $e_{u,v}$, of the b-matching as in the proof of theorem 5.2.1. A bit of preprocessing is needed to insure that this is done efficiently. Construct an array storing $A(v)$ and $B(v)$ for each $v \in S$. Also construct the sets $V_{i,j} = S - \{A_i \cup B_j\}$ and $W_{i,j} = B_i - A_i$ for all i, j. The algorithm uses these sets to find $u \in A_1$ and $v \in \overline{A_1 \cup B_{B(u)}}$ quickly.

After finding a suitable $x(i,j)$ relabel the A_i and B_i so that $u \in A_1 \cap B_1$ and $v \in A_2 \cap B_2$. Let $x_{e(u,v)} = x_{e(u,v)} + z$ where z is defined by

$$z = \min\{b(v),\ b(u),\ \frac{b(S)}{2} - b(A_3),\ \frac{b(S)}{2} - b(B_3)\}$$

After finding the appropriate value of z, the algorithm updates the following

- $b'(v) = b'(v) - z$
- $b'(u) = b'(u) - z$
- $x(u,v) = x(u,v) + z$
- If $z = \frac{b'(S)}{2} - b'(A_3)$, replace A_1 by $A_1 \cup A_2$, A_2 by A_3, and A_3 by $\emptyset$.
- If $z = \frac{b'(S)}{2} - b'(B_3)$, replace B_1 by $B_1 \cup B_2$, B_2 by B_3, and B_3 by $\emptyset$.
- If $b'(v) = 0$ remove v from S.

Build A_1, A_2, A_3, B_1, B_2, and B_3 using Algorithm 5.1.

Build the $V_{i,j}$ and the $W_{i,j}$.

While $b(S) \neq 0$

Rearrange the A_i so that $b(A_1) \geq b(A_2) \geq b(A_3)$

Choose any $v \in A_1$ and relabel the B_is so that $v \in B_1$.

Choose any $u \in W_{1,1}$ and relabel the B_is so that $b(u) = 2$.

Set $z = \min\{b(v),\ b(u),\ \frac{b(S)}{2} - b(A_3),\ \frac{b(S)}{2} - b(B_3)\}$

Decrease $b(u)$ and $b(v)$ by z.

If $b(u) = 0$ remove u from S

If $b(v) = 0$ remove v from S

Algorithm 5.2: Finding a B-Matching in a 2-Tree Metric.

It is easy to verify that the A_i and B_i still satisfy the assumptions of theorem 5.2.1 with b replaced by b'. Repeat step two until $S = \emptyset$. When the algorithm stops, we have a maximum weight b-matching, x^* on S. Pseudo code for the improved algorithm is given on the next page.

Theorem 5.2.4 *The algorithm described above finds an optimal b-matching in $O(n)$ time.*

Proof: Algorithm 5.1 takes

$O(n)$ time to find m_1 and m_2.

$O(n)$ time to find $\{C_1, ..., C_{k+1}\}$.

$O(k) = O(n)$ time to complete the for loop.

Step one of the algorithm runs Algorithm 5.1 twice taking $O(n)$ time to run. The sets $V_{i,j}$ and $W_{i,j}$ can be constructed in $O(n)$ time. Armed with these data structures we can find u and v in $O(1)$ time.

To see that the algorithm only needs to find $O(n)$ edges, consider what happens when an $x(u, v)$ is increased. If $z < \min\{\frac{b(S)}{2} - b(A_3), \frac{b(S)}{2} - b(B_3)\}$, then one of $b'(u)$ or $b'(v)$ will be set to zero. Since $b'(v)$ never increases, this happens at most n times. If $z = \{\min \frac{b(S)}{2} - b(A_3), \frac{b(S)}{2} - b(B_3)\}$ then either A_3 or B_3 has become empty. This can happen in at most 2 iterations. Therefore step two needs to find and increase $x(u, v)$ at most $n + 2$ times. Since u and v can be found in $O(1)$ time using the $V_{i,j}$'s and $W_{i,j}$'s built in the first step, the total runtime of step 2 is $O(n)$. Therefore the algorithm runs in $O(n)$ time. ∎

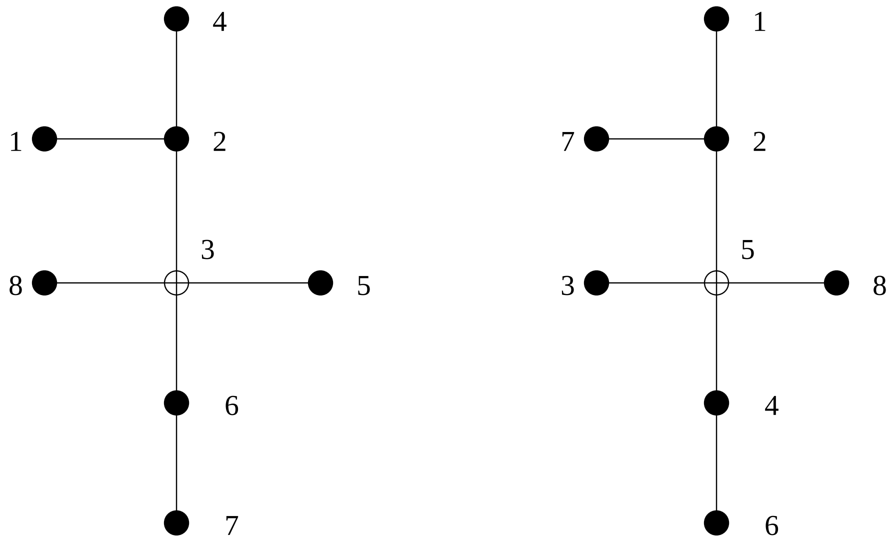
We conclude this section with an example of our algorithm. In order to save space, (and spare the reader tedium) the A_i's and B_j's are not explicitly relabeled in each iteration. Consider the instance of the maximum weight b-matching problem below.

$$D = \begin{bmatrix} 0 & 2 & 5 & 5 & 5 & 7 & 6 \\ 2 & 0 & 3 & 3 & 3 & 5 & 4 \\ 5 & 3 & 0 & 4 & 2 & 4 & 5 \\ 5 & 3 & 4 & 0 & 4 & 4 & 7 \\ 5 & 5 & 2 & 4 & 0 & 4 & 7 \\ 7 & 5 & 4 & 4 & 4 & 0 & 5 \\ 6 & 4 & 5 & 7 & 7 & 5 & 0 \end{bmatrix}$$

$$\begin{array}{llll} b(v_1) = 2 & b(v_2) = 2 & b(v_3) = 1 & b(v_4) = 2 \\ b(v_5) = 1 & b(v_6) = 3 & b(v_7) = 1 & b(v_8) = 4 \end{array}$$

D is the two tree metric that was recognized in chapter 4. The two-tree representation of D that was found then is given on the next page with the medians of the trees are represented by clear circles. One possible set of A_is and B_js is listed below.

$$\begin{array}{lclclcl} A_1 & = & \{v_3, v_5, v_6, v_7\} & \quad & B_1 & = & \{v_1, v_2, v_3, v_7\} \\ A_2 & = & \{v_1, v_2, v_4\} & & B_2 & = & \{v_5, v_8\} \\ A_3 & = & \{v_8\} & & B_3 & = & \{v_4, v_6\} \end{array}$$



Initially, the algorithm chooses $v = v_6$ and $u = v_8$. z is set to the minimum of $b(v_6) = 3$, $b(v_8) = 4$, $\frac{b(S)}{2} - b(A_2) = 2$, and $\frac{b(S)}{2} - b(B_1) = 2$. Since this minimum occurs when $\frac{b(S)}{2} - b(A_2) = 2$, and $\frac{b(S)}{2} - b^{'}(B_1) = 2$, we need to replace out A_is and B_js with the following sets.

$$\begin{array}{rclcrcl} A_1 & = & \{v_3, v_5, v_6, v_7, v_8\} & \quad & B_1 & = & \{v_1, v_2, v_3, v_7\} \\ A_2 & = & \{v_1, v_2, v_4\} & & B_2 & = & \{v_5, v_6, v_8\} \\ A_3 & = & \emptyset & & B_3 & = & \emptyset \end{array}$$

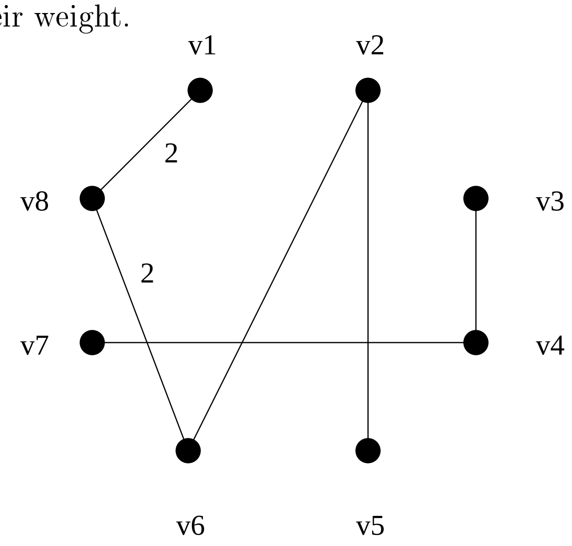
This leaves the weight function:

$$\begin{array}{cccc} b(v_1) = 2 & b(v_2) = 2 & b(v_3) = 1 & b(v_4) = 2 \\ b(v_5) = 1 & b(v_6) = 1 & b(v_7) = 1 & b(v_8) = 2 \end{array}$$

From now on z will only be determined by $b(v)$ and $b(u)$ since the terms $\frac{b(S)}{2} - b(A_i)$, and $\frac{b(S)}{2} - b^{'}(B_j)$ will always be $\frac{b(S)}{2}$. Next the algorithm could choose $v = v_8$ and $u = v_1$. z would be set to $2 = b(v_1) = b(v_8)$ and v_1 would be removed from S. Leaving the weight function:

$$\begin{array}{ccc} b(v_2) = 2 & b(v_3) = 1 & b(v_4) = 2 \\ b(v_5) = 1 & b(v_6) = 1 & b(v_7) = 1 \end{array}$$

The algorithm continues, assigning: $x(2, 6) = x(3, 4) = x(4, 7) = 1$. A graphical representation of the optimal b-matching is given on the following page. Edges which have $x(i,j) \geq 1$ are shown and those with $x(i,j) \geq 2$ are labeled with their weight.



5.3 B-matching and TDMs.

The assumptions theorem 5.2.1 places on $\mathcal{S}$ are very restrictive. In this section, we show how to relax those restrictions and extend theorem 5.2.1 to replace $\mathcal{S}$ any weakly compatible collection of cuts $\mathcal{C}$. The extension of theorem 5.2.1 allows us to extend the b-matching algorithm in section 4.2 to TDMs. Unfortunately, the $O(n)$ runtime does not extend. In this section we assume $b(C) \leq b(\overline{C})$ for all cuts.

Theorem 5.3.1 *Given an instance of the b-matching problem (D, b) with b(S) even, and a weakly compatible collection $\mathcal{C}$ of cuts of D, there exists an integral b-matching, x, on D such that if $x(u,v) > 0$ then $|C_k \cap \{u,v\}| \leq 1$ for all $(C_k, \overline{C}_k) \in \mathcal{C}$ and $|C_k \cap \{u,v\}| = 1$ if $b(C) = b(\overline{C})$*

Proof: Without loss of generality, assume that $b(w) > 0$ for all $w \in S$. As in the proof of theorem 5.2.1 we proceed by induction on k where $b(S) = 2k$. If $k = 0$ the result is trivial. Assume the result is true for graph with $b(S) = 2k$.

Construct a collection $\mathcal{S}$ of subsets of S from $\mathcal{C}$ as follows. If $(C, \overline{C}) \in \mathcal{C}$ add the set C to $\mathcal{S}$. In addition, if $b(C) = \frac{b(S)}{2}$ we also include $\overline{C}$ in $\mathcal{S}$. An edge (u,v) satisfies $|\{u,v\} \cap S| \leq 1$ for all $S \in \mathcal{S}$ exactly when u and v satisfy

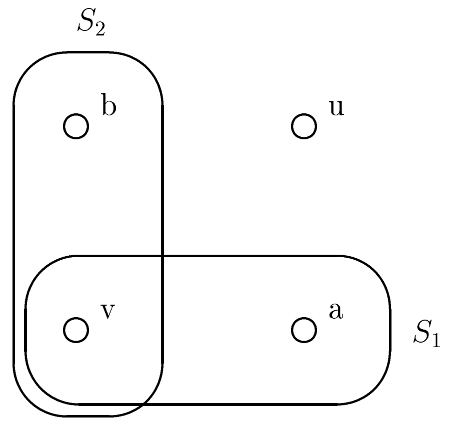


Figure 5.1:

the conditions of the theorem. The first step in the proof is to find u and v such that $|\{u, v\} \cap S| \leq 1$ for all $S \in \mathcal{S}$. Choose $v \in S$ and choose S_1 and S_2 from $\mathcal{S}$ such that $v \in S_1 \cap S_2$ and $b(S_1 \cup S_2)$ is maximal over all possible choices of S_1, and S_2. Since $v \in S_1 \cap S_2$, $b(S_1 \cup S_2) < b(S)$, and we can find a vertex $u \in S - (S_1 \cup S_2)$. (See Figure 5.1)

Now we show that there can not be a $S_0 \in \mathcal{S}$ with $\{u, v\} \subseteq S_0$. Suppose there is such a S_0. There must be some vertex $a \in S_1 - (S_2 \cup S_0)$ otherwise, $b(S_0 \cup S_2) > b(S_1 \cup S_2)$, violating the maximality of $b(S_1 \cup S_2)$. Similarly there must be a vertex $b \in S_2 - (S_0 \cup S_1)$. (see Figure 5.2) This implies

- $(S_0, \overline{S_0})$ separates $\{u, v\}$ and $\{a, b\}$.
- $(S_1, \overline{S_1})$ separates $\{a, v\}$ and $\{b, u\}$.
- $(S_2, \overline{S_2})$ separates $\{b, v\}$ and $\{a, u\}$.

Therefore the cuts $(S_0, \overline{S_0})$, $(S_1, \overline{S_1})$, and $(S_2, \overline{S_2})$ are not weakly compatible. But all of these cuts are in $\mathcal{C}$, contradicting the fact that $\mathcal{C}$ is a weakly compatible collection of cuts. Thus we have found an edge (u, w) which crosses every cut in $\mathcal{C}$.

Armed with this information we complete the proof using the inductive hypothesis to apply the theorem with b replaced by $b^{'}$, where b' is defined below.

- $b'(w) = b(S)$ if $w \notin \{u, v\}$

- $b'(v) = b(v) - 1$

- $b'(u) = b(u) - 1$

∎

An immediate corollary to theorem 5.3.1 is

Corollary 5.3.2 *Given an instance of the b-matching problem on a TDM D the maximum weight b-matching on D has weight $\sum_{i=1}^{k} \alpha_i b(C_i)$ where $\mathcal{C} = \{C_1, ..., C_k\}$ along with $\alpha_1, ..., \alpha_k$ is a total decomposition of the distance matrix of D.* ∎

Theorem 5.3.1 gives an easy way to find the weight of a max weight b-matching on a totally decomposable matrix. Unfortunately, the maximum weight b-matching algorithm which can be designed from the proof of theorem 5.3.1 runs in time comparable to existing general b-matching algorithms.

5.4 B-matching and CDMs.

When applied to CDMs, theorem 5.3.1 leads to an extremely fast and easy to follow algorithm. The additional structure of a CDM allows us to find an optimal b-matching knowing only that the identity permutation is a Kalmanson permutation. The algorithm does not even need to know the metric!

Our algorithm is a generalization of Marcotte and Suri's maximum weight matching algorithm for convex Euclidean metrics [31]. When run on a $2n \times 2n$ CEM, their algorithm assigns a values of 1 to $x(i, n+i)$ (mod n) and 0 to all other $x(i, j)$. Our algorithm extends their idea of matching half way around the circle.

Find the minimum a such that $\sum_{i=1}^{a} b(v_i) > \frac{b(S)}{2}$.

Set $u = v_1$, $v = v_a$, $p = b(u)$ and $q = \sum_{i=1}^{a} b(v_i) - \frac{b(S)}{2}$.

While $b(S) > 2\sum_{i,j \in S} x(i,j)$.

    Set $x(p,q) = \min\{p, q, \frac{b(S)}{2} - 2\sum_{i,j \in S} x(i,j)\}$.

    Set $p = p - x(p,q)$, and $q = q - x(p,q)$.

    If $p = 0$.

        Increment u and set $p = b(u)$.

    If $q = 0$.

        Increment v and set $q = b(v)$.

Algorithm 5.3: Finding a B-Matching in a CDM.

The first step in our algorithm finds the minimum index a such that the following holds.

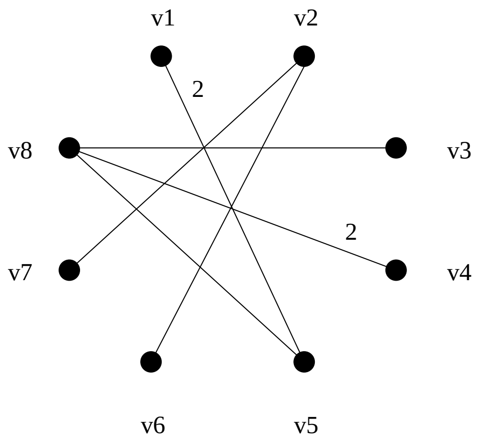
$$\sum_{i=1}^{a} b(v_i) > \frac{b(S)}{2}$$

The algorithm initializes its main loop by setting $q = \sum_{i=1}^{a} b(v_i) - \frac{b(S)}{2}$, $p = b(v_1)$, $u = v_1$, and $v = v_a$. In each iteration, it sets $x(u,v) = \min\{p,q\}$, and subtracts $x(u,v)$ from p and q. If $p = 0$ then increment u and set $p = b(u)$. If $q = 0$ it increments v and sets $q = b(v)$. The algorithm stops, when

$$b(S) = 2\sum_{i,j \in S} x(i,j)$$

Pseudo code for this algorithm is given above.

Theorem 5.4.1 *Algorithm 5.3 will return an optimal solution to the maximum weight b-matching problem on a CDM D.*



Proof: First we check the feasibility of the solution the algorithm generates. Since the algorithm halts when $2\sum_{i,j\in S} x(i,j) = b(S)$, it is clear that the only value that could be assigned to both u and v is v_a. Therefore, $\sum_j x(i,j) \leq b(v_i)$ when i is not a. The algorithm may assign $x(i,a)$ a non-zero value when $u = a$ or when $v = a$. A weight of at most $\sum_{i=1}^{a} b(v_i) - \frac{b(v)}{2}$ may be assigned to (x,a) when $u = a$ and a total of $\frac{b(v)}{2} - \sum_{i=1}^{a-1} b(v_i)$ may be assigned when $v = a$. Therefore $\sum_{j\in S} x(a,j) \leq b(v_a)$.

To see that x is an optimal b-matching, consider any d-split $C_{i,j} = (C, \overline{C})$. Complement this cut so that $b(C) \geq b(\overline{C})$. Clearly, if $\{i,j\} \subseteq \overline{C}$ then $x(i,j) = 0$. Therefore the b-matching x the algorithm constructs each d-split the minimum of $b(C)$ and $b(\overline{C})$ times. This implies that x is optimal. ∎

Theorem 5.4.2 *Algorithm 5.3 runs in $O(n)$ time.*

Proof: All of the steps before the loop can be easily completed in $O(n)$ time. Each iteration of the loop uses a constant number of operations. Each iteration of the loop either increments u, increments v or halts. Clearly u and v can be incremented at most n times and the algorithm halts at most once. This gives a run time of $O(n)$. ∎

Here is a brief example of Algorithm 5.3. Since the weights of the cuts are irrelevant to the algorithm, we only explicitly state $b(v)$.

$$\begin{array}{llll} b(v_1) = 2 & b(v_2) = 2 & b(v_3) = 1 & b(v_4) = 2 \\ b(v_5) = 3 & b(v_6) = 1 & b(v_7) = 1 & b(v_8) = 4 \end{array}$$

The algorithm finds that the minimum value of a with $\sum_{i=1}^{a} b(i) > \frac{b(S)}{2}$ is $a = 5$. Then the algorithm sets $v = v_1$, $u = v_5$, $p = 2$, and $q = 2$. After the first iteration of the loop, $x(1,5)$ is set to 2, v is incremented to v_6, q is set to 1, u is incremented to v_2 and p is set to 2. The second iteration of the loop sets $x(2,6)$ to 1, increments v to v_7, set q to 1, and sets p to 1. The third iteration of the loop sets $x(2,7)$ to 1, increments v to v_8, sets q to 4, increments u to v_3, and sets p to 1. The fourth iteration of the loop sets $x(3,8)$ to 1, sets q to 3, increments u to v_4, and sets p to 2. The fifth iteration of the loop sets $x(3,8)$ to 2, sets q to 1, increments u to v_4, and sets p to 3. The final iteration of the loop sets $x(4,8)$ equal to $\frac{b(S)}{2} - 2\sum_{i,j \in S} x(i,j)$ and halts. The solution is pictured below. Edges with $x(i,j) > 0$ are drawn, and edges with $x(i,j) > 1$ are labeled with their weight.

5.5 An alternative approach

In this section, we give present some theoretical ideas that form a foundation for the results in this chapter. In this section, it is convienient to change our perspective on the b-matching problem slightly. Instead of considering the maximum weight b-matching on a single metric as we have done previously, we consider the relationship between the maximum weight b-matching on distinct instances of the problem. Given three instances of the max weight b-matching problem, (D_1, b), (D_2, b) and (D_0, b) with $D_0 = D_1 + D_2$ we consider when there will exist fractional optimal b-matchings x_1^*, x_2^*, and x_0^* such that

$$D_1(x_1^*) + D_2(x_2^*) = D_0(x_0^*)$$

The following is an easy corollary of theorem 5.3.1

Theorem 5.5.1 *Let (D_0, b), (D_1, b), and (D_2, b) be three instances of the maximum weight b-matching problem, where each D_i is totally decomposable and $\mathcal{C}_0 = \mathcal{C}_1 \cup \mathcal{C}_2$. If x_0^* is an optimal integral b-matching on (D_0, b) then x_0^* is also optimal for the other two instances.*

Proof: Every cut in $\mathcal{C}_0$ is crossed a maximal number of times by x_0^*. Since $\mathcal{C}_1$ and $\mathcal{C}_2$ are sub collections of C_0 their cuts are also crossed a maximal number of times by x_0^*. ∎

Here we will give a partial extension of this theorem to a larger class of metrics. Unfortunately, we need to consider fractional b-matchings to prove the extension. The new result relies heavily on the idea of the injective hull of a metric.

Definition 5.5.2 *The* **injective hull** *of a metric D is defined as the set*

$$T(D) := \{f \in R_n | f(x) = sup_{y \in S}(D[x,y] - f(y)) \text{ for all } x \in S\}$$

Closely related to the injective hull is the set

$$P(D) := \{f \in R_n | f(x) + f(y) \geq D[x,y] \text{ for all } x \in S\}$$

Bandelt and Dress [8] point out that, $T(D)$ can be thought of as the minimal elements of $P(D)$ with respect to the partial ordering where $f \leq g$ if and only if $f(G) \leq g(G)$ for all $x \in X$. They [8] relate these ideas to TDMs with the following theorem.

Theorem 5.5.3 *Let $\mathcal{C}$ be a collection of cuts and let $D = \sum_{C \in \mathcal{C}} \alpha_C D_C$. $\mathcal{C}$ is weakly compatible if and only if*

$$T(D) = \sum_{C \in \mathcal{C}} T(\alpha_C D_C)$$

Motivated by this theorem, Bandelt and Dress [8] define coherent decomposition as an extension of a total decomposition. We use their notation in the definitions that follow.

Definition 5.5.4 *Metrics $D_1, ..., D_K$ are called* **coherent** *if $T(D_1 + ... + D_k) = T(D_1) + ... + T(D_k)$. In addition, we say that the metrics $D_1, ..., D_k$ are a* **coherent decomposition** *of $D_1 +, ..., +D_k$.*

To see how this relates to b-matchings, consider two coherent metrics D_1 and D_2 and let $D_0 = D_1 + D_2$. Let x_0^*, x_1^*, and x_2^* denote optimal solutions to (LP_0), (LP_1), and (LP_2) respectively.

$$\begin{array}{lll} & \text{Maximize} & \sum_{u,v \in S} D_l[u,v]x(u,v) \\ (LP_l) & \text{Subject to:} & \sum_{u \neq v} x(u,v) \leq b(v) \qquad \text{For all } v \in S \\ & & x(u,v) \geq 0 \end{array}$$

$$
(D_l) \qquad \begin{array}{lll} \text{Minimize} & \sum_{1 \le v \le n} b(v)y(v) & \\ \text{Subject to:} & y(u) + y(v) \ge D_l[u,v] & \text{For all } u \ne v \\ & y(v) \ge 0 & \end{array}
$$

Clearly the following holds,

$$
\sum D_1(x_1^*) + \sum D_2(x_2^*) \ge \sum D_3(x_0^*) \tag{5.7}
$$

A careful analysis of the D_l shows that (5.7) holds with equality. Let y_l^* be an optimal solution to (D_l). Notice that $y_0^* \in P(D_0)$. Since $D_1 + D_2$ is a coherent decomposition of D_0, there must be elements $y_1^* \in P(D_1)$ and $y_1^* \in P(D_2)$ such that $y_1^* + y_2^* = y_0^*$. Clearly y_1^* must be an optimal solution of (D_1) and y_2^* must be an optimal solution to (D_2). The existence of y_1^* and y_2^* implies that Equation 5.7 holds with equality. Now a simple inductive proof yields the following theorem

Theorem 5.5.5 *Given a metric D_0, a coherent decomposition of D_0, $D_0 = D_1 + ... + D_k$, and any non-negative weight function b on S, the weight of the maximum weight fractional b-matching on the metric D_0 is the sum of the maximum weight fractional b-matchings on the metric spaces $D_1, D_2, ..., D_k$.*

Chapter 6

Applications to Max Cut

6.1 Introduction

Another NP-hard problem that can be solved using one of our special case metrics is the Maximum cut problem. The weight w(C) of a cut $C, \overline{C}$ is defined as

$$w(C) = \sum_{x \in C, y \in \overline{C}} d(i, j)$$

An instance of the maximum cut problem consists of a complete graph on n vertices with a weight $d(i, j)$ assigned to each edge $e_{i,j}$ in the graph. We will call the matrix D with $d_{i,j} = d(i, j)$ the weight matrix of the instance of max cut. The objective of the max cut problem is to find the cut C which maximizes $w(C)$ over all cuts in G. In this section an algorithm is given which solves the max cut problem in polynomial time when the weight matrix D is a tree metric. A cut C is called an n-**cut** if $|C| = n$. The max n-cut problem is similar to the max cut problem except that we only need to find the cut C which has maximum weight over all cuts with $|C| = n$.

6.2 Tree Metrics and Max Cut

We use the notation of section 3 when referring a tree T and its median m. The connected components of $T - m$ are labeled $M_1, ..., M_k$.

The heart of our algorithm is the following lemma.

Lemma 6.2.1 *Given a tree metric M and a tree representation T of M, if C is a max n-cut, at most one M_i is not contained entirely in either C or $\overline{C}$.*

Proof.

The proof of this lemma relies on the algorithm in section 2.7 which finds a cut decomposition of a metric which is the sum of two tree metrics. Denote the cuts which have non zero weight in the total decomposition of M by $\mathcal{C} = \{\mathcal{C}_\infty, ..., \mathcal{C}_\updownarrow\}$ Consider a cut C. Here is an alternative way to express $w(C)$.

$$w(C) = \sum_{u \in C, v \in \overline{C}} d(u,v) \tag{6.1}$$

$$= \sum_{i=1}^{l} \alpha_{C_i} r_i \tag{6.2}$$

where $r_i = |C \cap C_i||\overline{C} \cap \overline{C_i}| + |\overline{C} \cap C_i||C \cap \overline{C_i}|$. Intuitively, r_i represents the contribution of the cut C_i to $w(C)$. To prove the claim we examine the representation of w(C) given in 6.2. The largest that r_i can ever be is $a(n-a)$ where $a = min\{|C|, |C_i|\}$. Suppose that we are given a supposed max cut $\overline{C}$ and there exist M_i and M_j such that $M_i \cap C \neq \emptyset$, $M_i \cap \overline{C} \neq \emptyset$, $M_j \cap C \neq \emptyset$, and $M_j \cap \overline{C} \neq \emptyset$.

Armed with lemma 6.2.1 we are ready to present an algorithm to solve the max cut when the weight matrix is a tree metric.

Theorem 6.2.2 *The above algorithm will find a maximum cut in a complete graph G whose weight matrix is a tree metric*

Proof.

Theorem 6.2.3 *The algorithm runs in $O(n^{100})$ time*

Proof.

Chapter 7

Applications to the TSP

7.1 Introduction

In this section we discuss the relationship between CDMs and three problems related to the traveling salesman problem (TSP): the prize collecting TSP (PCTSP), the k-minimum spanning tree problem (k-MST), and the minimum latency TSP (MLTSP). These three NP-hard problems can be solved in polynomial time when the objective function is a CDM.

In section two, we consider the prize collecting TSP. In this version of the TSP the salesman must sell enough units of a product to meet a sales quota, q. The salesman knows a demand function $p : V \to Z^+$ that tells him how many units he will be able to sell in each city. The salesman would like to meet his quota and return home as quickly as possible. The PCTSP was first considered by Balas [6] and has applications to steel mill production schedules. To solve this problem, we modify a well known dynamic programming algorithm for the single source shortest path.

Section 3 of this chapter considers the minimum latency TSP. This variation of the TSP interprets the distances as times and asks the salesman to minimize the sum of the amounts of time the customers must wait. Blum et al. [13] have related this problem to the problem of finding the minimum cost tree which contains exactly k of the vertices, known as the k-minimum spanning tree (k-MST) problem.

To solve the k-MST problem, we use the Kalmanson conditions to show that there exists an optimal solution x^* which has $x^*(i,j) = 0$ or $x^*(j,k) = 0$ whenever $i < j < k < l$. Such a tree is called **non-crossing**. Once the existence of a non-crossing optimal tree is established, a simple dynamic programming algorithm is used to find the optimal non-crossing tree. Due to the straightforward nature of the dynamic programming, no pseudo code or examples are given in this chapter. We also apply a theorem of Blum et al. [13] to use this construction to get an 8-approximation to the minimum latency tour.

In this chapter, we reformulate our idea of a metric in terms of complete graphs. Given an $n \times n$ CDM, D we build a complete graph G_D on n vertices and assign a cost of $D[i, j]$ to the edge $e(i, j)$. This allows us to simplify our proofs by define crossing in terms of edges.

Definition 7.1.1 *Two edges* (u, v) *and* (x, y) *of the complete graph* K_n *are called* **crossing** *if either of the following conditions hold:*

$$\begin{array}{ccccccc} u & < & x & < & v & < & y \\ x & < & v & < & y & < & u \end{array}$$

In the graphs on the left of the arrows in Figure 6.1 on page 94 the solid edges are crossing.

7.2 The Prize Collecting TSP

An instance (D, p, q) of the prize collection traveling salesman problem (PCTSP) consists of an $n \times n$ metric, D, a weight function, $p(v)$, on S, and a quota, q. The goal is to find a tour T such that

1. $v_1 \in T$
2. $\sum_{v \in T} p(v) \geq q$
3. $\sum_{e_{i,j} \in T} d[i, j]$ is minimal over all T satisfying 1 and 2.

Consider the uniform PCTSP with $p(v) = 1$ for all $v \in S$. In this case the salesman needs to visit q cities and return home as cheaply as possible. To solve this problem, construct a complete directed graph G with $V(G) = \{1, ..., n+1\}$ and assign weights to the edges of G as follows:

$$d'[i,j] = \begin{cases} D[i,j] & 1 \le i < j \le n \\ D[1,i] & 1 \le i \le n \quad\quad j = n+1 \\ \infty & \text{otherwise} \end{cases}$$

The well known Bellman [12] Ford [26] (or see [19]) algorithm for finding single source shortest paths can be used to find the cheapest path from 1 to $n+1$. There is a one to one correspondence between cycles of length q containing 1 in G and paths of length q from v_1 to v_{n+1} in G'. The path $P = \{v_{i_1}, v_{i_2}, ..., v_{i_q} = v_{n+1}\}$ corresponds to the cycle $C = \langle i_1, i_2, .., i_{q-1}, 1\rangle$. Notice that the way that we have constructed G' insures that the cost of the cycle C in G is the same as the cost of the corresponding path in G'. Therefore, we can find the optimal tour containing q vertices by finding the minimum cost path from v_1 to v_{n+1} which contains exactly q edges of G'.

This approach can be easily extended to the uniform PCTSP when $p(v) = p$ for all $v \in V(G)$. In this case, the salesman must visit $\lceil \frac{q}{p} \rceil = r$ cities. Therefore, this problem can be solved exactly as before by placing a unit prize on each city and requiring the salesman to visit r cities.

To extend the algorithm to the non-uniform case, we need to assume that the sum of the prizes is polynomial in the number of cities. More formally, given a polynomial $p(n)$, define the class $\mathcal{I}(p(n))$ as the set of all PCTSP instances with the sum of the prizes less than $p(n)$. Given an instance $(D, p, q) \in \mathcal{I}(p(n))$, construct an instance (D', p, q) of the uniform PCTSP as follows. Create a new metric D' from D which contains $p(i)$ copies of row and column i. This construction be completed in a polynomial amount of time. It is easy to verify that D is a CDM.

Now the uniform PCTSP algorithm can be used to get an optimal solution T' on (D', p, q). Construct an optimal solution T to the original problem as follows. If any of the copies of row i occur in T', then include i in T. The cities in T are visited in increasing order.

This idea is not likely to extend to taxi metrics, since any algorithm which solves the PCTSP over taxi metrics could be used to solve the TSP over taxi metrics. Solving the TSP over the class of taxi metrics is NP-hard [28].

7.3 Minimum Latency TSP

We begin this section by formally stating two related optimization problems: the minimum latency traveling salesman problem (MLTSP) and the k-minimum spanning tree problem (k-MST).

An instance of the **Minimum Latency Traveling Salesman Problem** consists of a set of n points $\{1, ..., n\}$ and an $n \times n$ distance matrix D. The objective is to find the tour which minimizes the quantity

$$\sum_{i=1}^{n} l(i) = \sum_{i=1}^{n} (n-i) d(i, i+1)$$

where $l(i)$ is the distance traveled before reaching the point v_i.

An instance of the **K-minimum Spanning Tree Problem** consists of a complete graph G on n vertices and an $n \times n$ metric D. The goal of the k-MST is to find the minimum cost tree containing exactly k of the vertices of G. Such a tree is called a **k-MST**. Any tree containing k vertices is called a **k-tree**. The cost $D(T)$ of the k-tree T is the sum of the costs of the edges in T.

The k-MST problem is was independently introduced and shown to be NP-complete by Ravi et al. [35], Fischetti et al. [25] and Zelikovski and Lozevanu [39]. Mitchell [33] and Arora [3] have independently shown that the k-MST can be approximated to within an arbitrary constant ϵ when metric D can be isometrically embedded in R^d for a fixed d. Their algorithms are polynomial in the n but exponential in $\frac{1}{\epsilon}$.

Blum et al. [13] show how to solve the k-MST problem on a tree metric using dynamic programming. Ravi et al. [35] give an $O(k^2 n^7)$ dynamic programming algorithm for solving the k-MST on a CEM. In this section, we give an algorithm which solves the k-MST on a CDM in $O(k^3 n^4)$ time. Since tree metrics and CEMs are subclasses of CDMs, our algorithm is more general than the previous algorithms.

Once we can solve the k-MST on a CDM, we can apply the algorithmic proof of the following theorem of Blum et al. [13] to get an 8-approximation to the MLTSP on a CDM.

Theorem 7.3.1 *(Blum et al. [13]) Any algorithm that gives a c-approximate solution to the k-MST, can be modified into an 8c-approximation algorithm to the MLTSP.*

Our dynamic programming algorithm is based on the following simple lemma which explains how to uncross edges in a k-tree T to get another k-tree T' with $D(T) \geq D(T')$.

Lemma 7.3.2 *Given an instance of the k-MST problem where D is a CDM, there exists a non-crossing k-MST.*

Proof: Let D be a minimal (with respect to $|S(D)|$) CDM which has only crossing k-MSTs. Let T be any k-MST of D. The metric D' obtained by deleting the elements of S_D that are not in T has T as a minimum spanning tree (MST). Any non-crossing spanning tree T' of D' must have $D(T') = D'(T') > D'(T) = D(T)$. Otherwise T' would be a k-MST for D, contradicting our choice of D. Therefore, we may assume $k = n$ and we only need to show that D has a non-crossing MST.

Choose an MST T with a minimum number of pairs of crossing edges. We show that if there are any crossing edges in T then we can construct a new tree T^0 which has fewer pairs of crossing edges than T and satisfies $D(T^0) \leq D(T)$.

Consider any pair $(u_1, v_1), (u_2, v_2)$ of crossing edges in T. Due to the circular symmetry of the metric and the symmetry of the edges, we can assume that these edges are of the form (u_1, v_1) (u_2, v_2) where $u_1 < u_2 < v_1 < v_2$. Our goal is to show that we can modify T by replacing the edges (u_1, v_1) and (u_2, v_2) in one of the ways illustrated in figure 6.1 to get a tree T^0 with $D(T) \geq D(T^0)$.

First, we show that one of the two replacements generates a tree T^0. Consider the four possibilities:

1. There is a path in T connecting u_1 and u_2 in $T - \{(u_1, v_1), (u_2, v_2)\}$.

2. There is a path in T connecting u_1 and v_2 in $T - \{(u_1, v_1), (u_2, v_2)\}$.

3. There is a path in T connecting v_1 and u_2 in $T - \{(u_1, v_1), (u_2, v_2)\}$.

4. There is a path in T connecting v_1 and v_2 in $T - \{(u_1, v_1), (u_2, v_2)\}$.

In cases 1 and 4, (u_1, v_1) and (u_2, v_2) should be replaced by (u_1, v_2) and (u_2, v_1) as illustrated at the top of figure 6.1. In cases 2 and 3, (u_1, v_1) and (u_2, v_2) should be replaced by (u_1, u_2) and (v_1, v_2) as illustrated in the bottom of figure 6.1. Thus, a tree is generated by at least one of the replacements illustrated in figure 6.1.

To finish the proof, we need to show that there are fewer crossing edges in T^0 then there are in T. There are several cases. Assume that (u_1, v_1) and (u_2, v_2) were replaced by (u_1, v_2) (u_2, v_1). Other cases can be handled

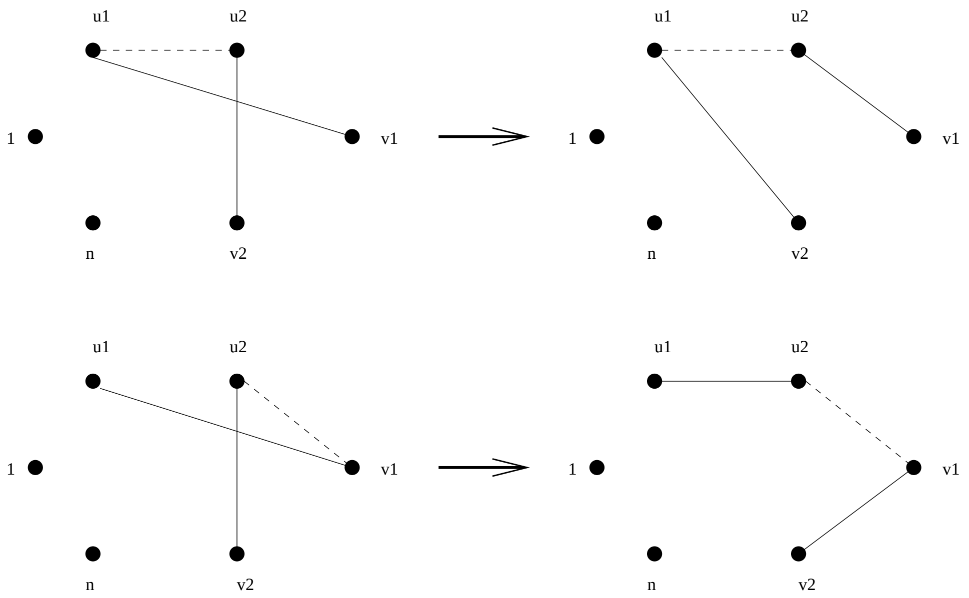


Figure 7.1:

similarly. Suppose (i, j) crosses (u_1, v_2). We may assume, without loss of generality, that $i \in \{u_1 + 1, ..., v_2 - 1\}$ and $j \in \{v_2 + 1, ..., u_1 - 1\}$. If $i \in \{u_1 + 1, ..., v_1 - 1\}$, then (i, j) crosses (u_1, v_1). If $i \in \{u_2 + 1, ..., v_2 - 1\}$, then (i, j) crosses (u_2, v_2). This accounts for all possible values of i since $u_2 < v_2$. A similar argument holds when (i, j) crosses (u_2, v_2). Therefore there are fewer pairs of crossing edges in T^0 then there were in T.

Since CDMs satisfy the Kalmanson conditions, $D(T^0) \leq D(T)$. Therefore, we have constructed a minimum spanning tree T^0 with fewer pairs of crossing edges then T contradicting our choice of T and establishing the theorem. ∎

Using this lemma, we can design a simple dynamic programming algorithm for solving the k-MST problem on a CDM. First we need a piece of notation. Given a CDM D and two elements x and z of S_D. When $x < z \leq n$ we use $S_{x,z}$ to denote the set $\{x + 1, ..., z - 1\}$ and $S_{z,x}$ to denote the set $\{z + 1, ..., n, 1, ..., x - 1\}$.

The dynamic programming algorithm finds the cost of a minimum cost i-tree $T_{x,y,z,i}$ continuing the path (x, y, z) and $i - 3$ vertices of $S_{x,z}$ for all $3 \leq i \leq k$, $x, z \in S(D)$ and $y \in S_{z,x}$, and the cheapest i-tree $T_{x,z}$ using the edge (x, z) and $i - 2$ vertices in $S_{x,z}$ for all $x, z \in S$. It uses this information

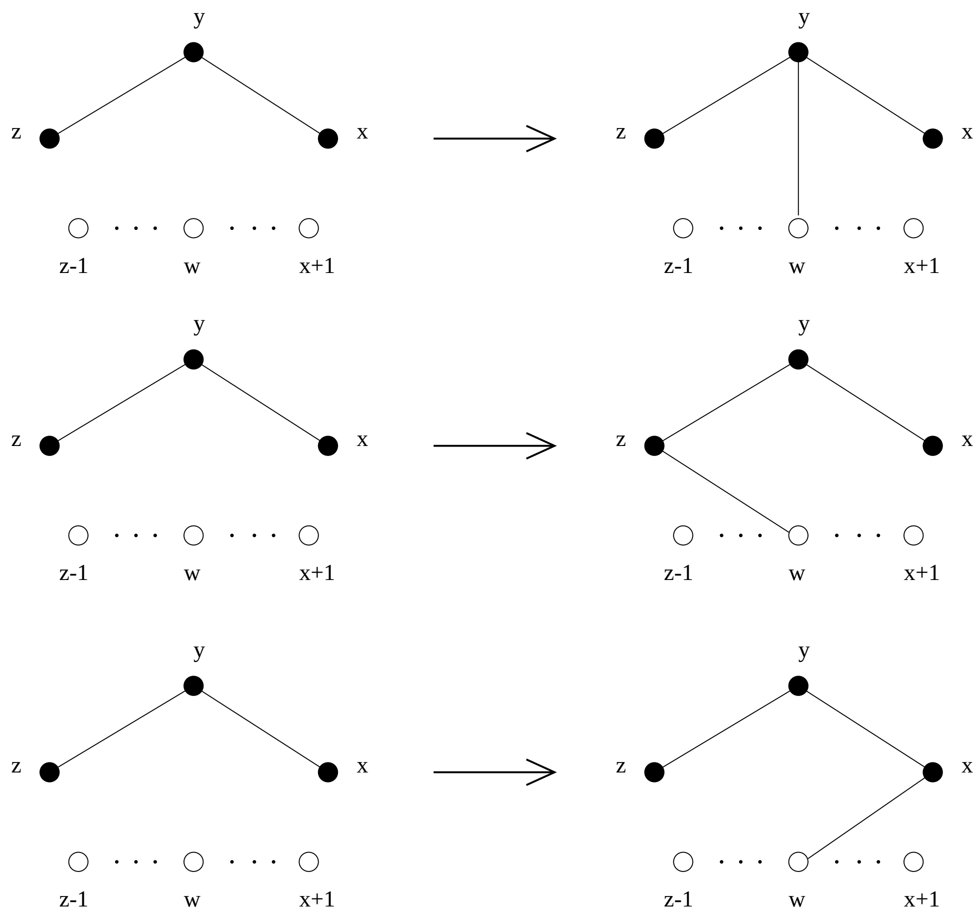


Figure 7.2:

to build a k-MST by gluing together smaller non-crossing trees.

More formally, the algorithm keeps track of an array of $M(x, y, z, i)$ for all distinct $x, z \in S$, $y \in S_{z,x}$ and $3 \leq i, j \leq k$. $M(x, y, z, i)$ stores the minimum cost non-crossing i-tree using the path $P = (x, y, z)$ and $i - 3$ vertices of $S_{x,z}$. A path of length three $P = (x, y, z)$ can be modified into a tree containing four vertices x, y, z, w where $w \in S_{z,x}$ in the three ways illustrated in figure 6.2. These three ways of adding a vertex are called Type I, type II and type III additions. $M(x, y, z, i)$ should be defined as ∞ when $i < 3$, and $D[x, y] + D[y, z]$ when $i = 3$. Otherwise, it is defined recursively as follows:

$$
\begin{aligned}
m_1(x,y,z,w,j) &= M(x,y,w,j) + M(w,y,z,i-j+2) - D[y,w] \\
m_2(x,y,z,w,j) &= M(x,z,w,j) + M(w,z,i-j+2) + a(x,y,z,w,j) \\
m_3(x,y,z,w,j) &= M(w,x,z,j) + M(x,w,i-j+2) + b(x,y,z,w,j) \\
a(x,y,z,w,j) &= D[x,y] + D[y,z] - D[x,z] - D[z,w] \\
b(x,y,z,w,j) &= D[x,y] + D[y,z] - D[x,z] - D[x,w] \\
M(x,y,z,i) &= \min_{\substack{2 \le j < i-1 \\ w \in S_{x,y}}} \{\min_p m_p(x,y,z,w,j)\}
\end{aligned}
$$

The following statements motivate the definitions of the m_i. Assume that all i-trees discussed in this section use only vertices in $S_{x,v} \cup \{x,y,z\}$ and minimality refers to minimality over this class of restricted i-trees. m_1 is the cost the minimum cost non-crossing i-tree containing the path P and using the edge (y,w) for some $w \in S_{x,z}$. m_2 is the cost of the minimum cost non-crossing i-tree containing P and using the edge (x,w) for some $w \in S_{x,z}$, but not using any edge of the form (y,w') where $w' \in S_{x,z}$. m_3 is the cost of the minimum cost non-crossing i-tree containing P, and using the edge (z,w) for some $w \in S_{x,z}$, but not using any edge of the form (y,w') where $w' \in S_{x,z}$. The proof of these claims is embedded in the proof of theorem 7.3.3 below.

The algorithm must also keep track of $M(x,y,i)$ for all distinct $x,y \in S$ and $2 \le i \le k-1$. $M(x,y,i)$ stores the cost of the minimum cost non-crossing i-tree using the edge (x,y) and $i-2$ vertices of $S_{x,y}$. $M(x,y,i)$ is set to ∞ when $i < 2$, and $D[x,y]$ when $i = 2$. Otherwise, it is computed recursively using the following equations

$$
\begin{aligned}
p_1(w,j) &= M(x,w,j) + M(w,x,y,i-j) - D[x,w] && (7.1) \\
p_2(w,j) &= M(y,w,j) + M(x,y,w,i-j) - D[x,w] && (7.2) \\
M(x,y,i) &= \min_{\substack{2 \le j < i-1 \\ w \in S_{x,y}}} \min\{p_1(w,j), p_2(w,j)\} && (7.3)
\end{aligned}
$$

Theorem 7.3.3 *When the dynamic program described above runs on the metric D, the following hold for all values actually calculated by the dynamic program.*

1. *$M(x,y,z,i)$ contains the value of the minimum value non-crossing $i-$ tree containing the path (x,y,z) and $i-3$ other vertices in $S_{x,z}$ for all $i \le k$.*

2. *$M(x, y, i)$ contains the value of the minimum cost non-crossing $i-tree$ containing the edge (x, y) and $i-2$ other vertices of $S_{x,y}$ when $i \leq k$.*

Proof: Our proof proceeds by induction on n. It is straightforward to verify that the claims hold when $n \in \{1, 2, 3, 4\}$. Given a path $P = (x, y, z)$ with $V = S_{z,y} \cup S_{y,x}$ non-empty, choose $v \in V$ and notice that the steps that the algorithm uses to compute $M(x, y, z, k)$ when run on D are identical to the steps it uses when run on D^T where $T = S - \{v\}$. Therefore, by induction the dynamic programming algorithm computes $M(x, y, z, i)$ correctly for all values of $i \leq k$ when (x, y, z) is not of the form $(w, w+1, w+2)$. A similar argument works to show that $M(x, y, i)$ holds the correct value when $y \neq x+1$.

If $i = 3$ then clearly $M(x-1, x, x+1, 3)$ is the claimed value. Assume $i > 3$ and let T be a minimum cost non-crossing i-tree which contains the path $(x-1, x, x+1)$ chosen so that the degree of x, $d(x)$ is maximal over all such choices of T. Consider two cases: $d(x) > 2$ and $d(x) = 2$.

In the first case, there must exist a $w \in S_{x+1,x-1}$ such that the edge (w, x) is in T. Since there are no crossing edges in T, removing the vertices $S_{x,w}$ from T leaves a non-crossing tree T_1 containing the path $(x-1, x, w)$. If T_1 contains j vertices, it must be a minimal cost non-crossing j-tree containing the path $(x-1, x, w)$ and $j-3$ other vertices in $S_{w,x}$. Therefore, by induction, the cost of T_1 is $M(x, x+1, w, j)$. Similarly, the tree built by removing the vertices of $S_{w,x}$ leaves a tree T_2 with cost $M(w, x, x+1, i-j+2)$. Therefore the cost of T is exactly $M(x-1, x, w, j) + M(x, x+1, w, i-j+2) - D[x, w]$. This establishes the fact that, $M(x-1, x, x+1, i)$ contains the value claimed in the statement of the theorem.

In the second case, replacing the path $(x-1, x, x+1)$ in T with the edge $(x-1, x+1)$ to get a minimum cost non-crossing (i-1)-tree T' which contains the edge $(x-1, x+1)$. T' must contain either an edge $(x-1, w)$ or an edge $(x+1, w)$ where $w \in S_{x+1,x-1}$. We assume the first case. (The second is similar.) Now we can use a argument similar to the one in the previous paragraph to break T' into two subtrees. The key observation is that $d(x) = 2$, therefore, there will never be any edges in this tree that cross the edge $(x-1, x+1)$. Therefore, the cost of T can be computed from the cost of the minimum cost $M(x-1, x+1, i-1)$ of the $i-1$ tree T_0 containing the edge $(x-1, x+1)$ by subtracting $D[x-1, x+1]$ and adding $D[x-1, x]$ and $D[x, x+1]$. $M(x-1, x+1, i-1)$ is correctly computed by induction so we can use this to calculate the cost of T'.

The induction hypothesis guarantees that $M(x, y, i)$ is correct when $y \neq x+1$. Clearly, $M(x, x+1, 2)$ is calculated correctly. When $k > 2$ and $y = x+1$ we use essentially the same proof as we did in the proof of the first claim. Let T be any minimum cost non-crossing i-tree containing $(x, x+1)$. Since $k > 2$, there must be another edge in T incident to either x or $x+1$. Assume this edge is of the form (x, w). (The other case is similar.) Now decompose T into two trees, T_1 using the path $(w, x, x+1)$ and $i-2$ vertices in $S_{x,w}$, and T_2 using the edge (w, x) and $k-i-2$ vertices in $S_{w,x}$. T_1 and T_2 must be minimal such trees. Therefore, $M(x, y) = D(T_1) + D(T_2) - D[x, w]$. Using induction and the first claim of the theorem, this is the value claimed in the second claim of the theorem. ∎

To compute the cost of the k-MST when $k \leq 3$ our algorithm enumerates all possible trees and chooses the cheapest. Otherwise, the algorithm runs the dynamic program described above and uses the values to calculate

$$
\begin{aligned}
M_{i,j} &= \min_{x,y,z \in S} M(z, y, i) + M(y, z, j) + M(x, y, z, k-i-j+4) \\
&\quad -D[z, y] - D[y, x] \\
M &= \min_{i,j} M_{i,j}
\end{aligned}
$$

Theorem 7.3.4 *M is the cost of a k-MST.*

Proof: Let T be any non-crossing k-tree. Choose a path of length three $P = (x, y, z)$ in T. T is the union of three trees. T_1 using the edge (z, y) and i vertices in $S_{z,y}$, T_2, using the edge (y, x) and j vertices in $S_{y,x}$ and T_3 using the path P and $k-i-j+4$ vertices in $S_{x,z}$. Each of these trees must be minimal. The edges of P are each used in two of the trees. This establishes the theorem. ∎

The previous theorems can be combined to give:

Theorem 7.3.5 *The dynamic programming algorithm described in this section finds the cost of a k-MST in $O(k^3 n^4)$ time.*

Proof: There are $O(n^3 k)$ $M(x, y, z, i)$ and $M(x, y, i)$ values to calculate. Each one of these is a minimum of $O(nk^2)$ values that can be calculated in $O(1)$ time. This gives a total of $O(n^4, k^3)$ time to compute the M's. The cost of the k-MST is calculated by taking a minimum over $O(n^3 k^2)$ numbers that can be computed in $O(1)$ time. Therefore the algorithm runs in $O(n^4k^3)$ time. ∎

In order to be able to reconstruct the tree, the algorithm should store the values for which the minimum is achieved in the definitions of the $M(x, y, z, i)$ and $M(x, y, i)$. After it finds the minimum $M(x, y, z, k)$ the algorithm can use this information to easily build the tree.

Chapter 8

Future Research

The results in the preceeding chapters are the beginning of a more complete research effort. Future research will focus on three main areas: developing faster recognition algorithms for the classes of metrics developed, finding more algorithmic applications of these classes, and determining how well general metrics can be approximated by these classes. Details of these objectives are given below.

Speeding up algorithms which recognize the various classes of TD metrics is essential. In order to run a special case algorithm on a metric, you need to be able to determine if the metric satisfies all the conditions that the algorithm exploits. Further research in this area will include but mot be limited to further analysis of the general algorithm in chapter 4. Specifically, an algorithm will be developed which recognizes taxi metrics by using the general algorithm.

One of the limitations of the general algorithm is the fact that it must update the matrices D and A each time it finds a minimal cut around x. A considerable speed up can be achieved if more than one cut could be found at a time. This possibility will also be considered.

Once fast recognition algorithms have been developed, additional special case algorithms will be developed. Another area of future research will be the development of algorithms for solving optimization problems over the classes of metrics we have developed. A large group of fast (often linear time) algorithms have been developed to solve problems on convex Euclidean metrics. Since convex Euclidean metrics are a sub class of CDMs, it seems natural to try to extend these algorithms. In fact, the algorithm for solving the maximum weight b-matching problem over a CDM is a generalization of a well known algorithm for finding the maximum weight perfect matching in a convex Euclidean metric.

One way that special case algorithms are used is to find approximate solutions to problems on a more general class of metrics. In order to use this approach, a measure of how closely one metric approximates another must be developed. For instance we could use the idea of an ϵ approximation.

Definition 8.0.6 *We say that the metric* (D_0) *is an* ϵ *approximation of the metric* (D) *if* $||d[i,j] - d_0[i,j]|| \leq \epsilon$ *for all* $i, j \in S$.

Consider a general metric D. Let ϵ_0 denote the minimum ϵ such that there is a tree metric D' which is an ϵ approximation of D. Agarwala et. al. [1] have devised an algorithm which will find a tree metric (G, d_1) which is a $3\epsilon_0$ approximation of D in $O(n^2)$ time. They also show that it is NP-hard to find a tree metric D_2 which is an $\frac{9}{8}\epsilon_0$ approximation of D. Their results lead to another area of further research. Can their approximation results be extended to other classes of totally decomposable matrices?

If time remains after considering these problems, research into the following areas may also be considered. What properties of optimization problems allow them to be solved quickly over totally decomposable metrics? Can the theory developed by Bandelt and Dress be extended to handle a larger class of metrics, for instance the class of metrics recognized by the general algorithm in chapter 4.

Chapter 9

Conclusion

CDMs correct and powerfull extension of Tree Metrics. Both for speeding up algs and for hanling NP=completeness.

TDMs offer more insight into the structure of a problem, but don't help with with the problem of NP-completeness.

Bibliography

[1] R. Agarwala, V. Bafna, M. Farach, B. Narayanan, M. Paterson, and M. Thorup, "On the Approximability of Numerical Taxonimy," *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1996.

[2] R.P. Anstee, "A Polynomial Algorithm for b-matchings: an Alternative Approach," *Information Processing Letters*, 24 (1987) pp. 153-156.

[3] S. Arora, "Polynomial Time Approximation Schemes for Euclidean TSP and other Geometric Problems." preprint.

[4] D. Avis and M. Deza, "The Cut Cone L^1 Embeddability, Complexity, and Multicommodity Flows," *Networks*, 21 (1991) pp. 595-617.

[5] V. Bagatelj, T. Pisanski, and J.M.D. Simões-Pereira, "Tree-Realizability of Distance Metrics," *SIAM Conference on Discrete Mathematics*, San Francisco, 13-16 June 1988.

[6] E. Balas, "The Prize Collecting Traveling Salesman Problem," ORSA/TIMS meeting in Los Angeles, Spring 1986.

[7] H-J. Bandelt, and V. Chepoi, "Embedding Metric Spaces in the Rectilinear Plane," *Disc. and Comp. Geometry*, 15 (1996) pp. 107-117.

[8] H-J. Bandelt, and A.W.M. Dress, "A Canonical Decomposition Theory for Metrics on a Finite Set," *Advances in Mathematics*, 92 (1992), pp. 47-105.

[9] H-J. Bandelt, and A.W.M. Dress, "Reconstructing the Shape of a Tree from Observed Dissimilarity Data," *Adv. in Appl. Math.* 7 (1986), pp. 309-343.

[10] Y. Bartal "Probabilistic Approximation of Metric Spaces and its Algorithmic Applications," *Proceedings FOCS 1996.*

[11] J. Barthelemy and A. Guenoche, **Trees and Proximity Relations**, Wiley, 1991.

[12] R. Bellman, "On a Routing Problem," *Quarterly of Applied Mathematics*, 16 (1958), pp. 87-90.

[13] A. Blum, P. Chalasani, D. Coppersmith, B. Pulleyblank, P. Raghavan, M. Sudan, "The Minimum Latency Problem," *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, (1994), pp. 163-171.

[14] F.T. Boesch, "Properties of the Distance Matrix of a Tree," *Quarterly of Applied Math*, 26 (1968), pp. 607-609.

[15] K. S. Booth, and G. S. Lueker, "Testing for the Consecutive ones Property, Interval Graphs and Graph Planarity Using PQ-tree Algorithms," *Journal of Computer and System Sciences*, 13, (1976), pp. 335-379.

[16] P. Buneman, "The Recovery of Trees from Dissimilarity Data" in **Mathematics in Archeological and Historical Sciences** F.R. Hodson, D.G. Kendal, and P. Tautu eds., Edinburgh University Press, pp. 387-395 (1971).

[17] R.E. Burkard, V.G. Deĭneko, and G.J. Woeginger, "The Traveling Salesman Problem and the PQ-Tree," *Proceedings of Integer Programming and Combinatorial Optimization 1996 (Springer Lecture Notes in Computer Science 1084)* W.H. Cunningham, S.T. McCormick, and M. Queyranne eds., pp. 490-504.

[18] R.E. Burkard, R. van Dal, V.G. Dieneko, J. van der Veen, G. J. Woeginger, "Well solved cases of the TSP: A Survey", Technical Report SFB 52, Institute of Mathematics, University of Technology, Graz, Austria (1995).

[19] T.H. Cormen, C.E. Leiserson, R.L. Rivest **Introduction to Algorithms**, MIT Press, 1990, pp. 532-535.

[20] J.C. Culberson and P. Rudnicki, “A Fast Algorithm for Constructing Trees From Distance Matrices,” *Information Processing Letters* 30 (1989), pp. 215-220.

[21] V. Deĭneko, R. Rudolf, and G. Woeginger, “Sometimes Traveling is Easy: The Master Tour Problem.” Technical Report SFB 13, Institute of Mathematics, University of Technology, Graz, Austria (1994).

[22] E. W. Dijkstra, “A Note on Two Problems in Connection with Graphs.” *Nureirsche Mathematic*, 1 (1959), pp. 269-271.

[23] M. Farach, “Recognizing Circular Decomposable Metrics” submitted for publication.

[24] J. Felsenstein, “Numerical methods for inferring evolutionary trees.” *Quarterly Review of Biology*, 57 (1985), pp. 193-200.

[25] M. Fischetti, H.W. Hamacher, K. Jornsten, and F. Maffioli, “Weighted k-cardinality trees: complexity and polyhedral structure.” *Networks* 24 (1994), pp. 11-21.

[26] L. R. Ford and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, 1962.

[27] P.C. Gilmore, E.L. Lawler, and D.B. Shmoys, “Well–solved special cases,” in [30], pp. 87–144.

[28] M.R. Garey, R.L. Graham, and D.S. Johnson, “Some NP-complete geometric Problems,” *Proceedings 8th Annual ACM Symposium on Theory of Computing* (1976), pp. 10-22.

[29] K. Kalmanson, “Edgeconvex circuits and the traveling salesman problem,” *Canadian Journal of Mathematics*, 27 (1975), pp. 1000-1010.

[30] E.L. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan, and D.B. Shmoys, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley (1985).

[31] O. Marcotte and S. Suri, “Fast Matching Algorithms for points on a polygon,” *SIAM Journal of Computation*, vol 20 no.3, (1991), pp. 405-422.

[32] J.S.B. Mitchell, A. Blum, P. Chalasani, S. Vempala, "A Constant-Factor Approximation Algorithm for the Geometric k-MST Problem in the Plane" preprint.

[33] J.S.B. Mitchell, "Guillotine Subdivisiona Approximate Poligonal Subdivisions: Part II. A Simple PTAS for geometric k-MST and related problems" preprint.

[34] U. Pferschy et. al. "Monge Matrices Make Maximization Manageable," *Operations Research Letters* 16 (1994), pp. 245-254.

[35] R. Ravi, R. Sundaram, M.V. Marathe, D.J. Rosenkrantz, and S.S. Ravi, "Spanning trees short and small." *SIAM Journal of Discrete Math.*, vol. 9, no. 2 (1996), pp. 178-200.

[36] D.F. Robinson, and L.R. Foulds, "Comparison of phylogenetic trees," *Math. Biosc.* 53 (1981), pp. 131-147.

[37] G. Sabidussi, "The centrality index of a graph" *Psychometrika* 31 (1966), pp. 58-603.

[38] A. Tamir, J.S.B Mitchel, "A Maximum b-Matching Problem Arising from Median Location Models with Applications to the Room-mates Problem," Technical Report 1995.

[39] A. Zelikovski, and D. Lozevanu, "Minimal and bounded trees." in *Tezele Cong XVIII Acad. Romano-Americane, Kishinev*, (1993), pp. 25-26.