An Algorithmic Approach to Continuous Location

Y. B. Chiang
Department of Mathematics and Applied Mathematics
University of Cape Town

A thesis submitted in partial fulfillment
of the requirements for
the degree of Master of Science in Mathematics

2 November 1995
The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.
Abstract

We survey the $p$-median problem and the $p$-centre problem. Then we investigate two new techniques for continuous optimal partitioning of a tree $T$ with $n - 1$ edges, where a nonnegative rational valued weight is associated with each edge. The continuous Max-Min tree partition problem (the continuous Min-Max tree partition problem) is to cut the edges in $p - 1$ places, so as to maximize (respectively minimize) the weight of the lightest (respectively heaviest) resulting subtree. Thus the tree is partitioned into approximately equal components. For each optimization problem, an inefficient implementation of the algorithm is given, which runs in pseudo-polynomial time, using a previously developed algorithm and a construction. We then derive from it a much faster algorithm using a top-down greedy technique, which runs in polynomial time. The algorithms have a variety of applications among others to highway and pipeline maintenance.
Contents

Acknowledgements

Introduction

1 Preliminaries
   1.1 Notation, Concepts and Terminology
   1.2 Problems, Algorithms and Complexity

2 The p-Median Problem
   2.1 The p-Median
   2.2 Comparability and the Vertex Optimal Theorem
   2.3 The Complexity of the p-Median Problem
   2.4 The p-Median Problem on Tree Networks
      2.4.1 Algorithm for Finding a 1-Median

3 The p-Centre Problem
   3.1 The p-Centre
   3.2 Algorithmic Approach to the p-Centre Problem
      3.2.1 Algorithm on weighted V/V/1/G

1
6.1.3 Conclusions .................................................. 117

Appendix ......................................................... 119

A A Detailed Proof of The One-to-One Correspondence of $\sigma$ .................................. 119

B Some Results For Chapter 5 ........................................ 121
   B.1 The Formal Description of the Algorithm .................................................. 121
   B.2 Complexity of the Continuous Down-Shifting Algorithm .................................. 124

Bibliography ...................................................... 126
Acknowledgements

To Prof R. I. Becker, my supervisor, I am truly grateful for his time, support and invaluable advice. His outstanding explanation and patience have provided invaluable guidance both for this thesis and in my development as a mathematician. Again, I express my deep gratitude.

I would like to thank Prof C. Brink, the Head of the Department of Mathematics and Applied Mathematics, for incorporating me in the Laboratory for Formal Aspects and Complexities in Computer Science (FACCS-lab). The FACCS-lab research group provided a stimulating environment for learning and research. It introduced many aspects of Mathematics which were new to me.

I would like to thank Prof B. Simeone for the numerous discussions, during his visit to University of Cape Town.

Finally, I would like to thank my parents, who stood by my side during all my years of study. I am grateful for their support and encouragement.
Introduction

Locational analysis, in short, is analysis of the notion of optimal choice within a spatial context. Here are some examples of such choices:

- In the design of urban service systems or analysis of transportation systems, there are problems of where new facilities are to be located on a network, in order to minimize with respect to cost or to satisfy time constraints.

- In communication systems and computer network planning, to serve a set of customers or communities, there are restrictions on the locations where the hardware is to be placed or restrictions on the distance between the connections, to reduce the noise interference.

- In circuit design, a designer may want to partition the circuits into subcomponents, which are constrained by the physical restrictions of the board, the size of each subcomponent or the number of connectors on the board.

Locational analysis is employed in a wide range of fields, such as in industrial engineering, operations research, management science and computer science. Hence there is a huge and rapidly growing amount of literature on this topic. However, in most of the literature, the emphasis is put on the discrete formulation of optimal locational decisions, discrete location theory. This thesis focuses on the continuous formulation of optimal locational decisions. The difference between the continuous formulation and the discrete formulation is that the continuous formulation presumes the objects to be located can be placed anywhere, that is, within a context of continuous space, whereas in the discrete formulation, the problems are analyzed in a discrete space, where the objects to be located can only be placed at a finite number of potential sites, selected by prior analysis. Here is a list of some possibilities of potential sites, if the network is represented as a graph:

- At some vertex.
- At the head of some edge.
• At the tail of some edge.
• At a point whose distances from the endpoints are in a fixed ratio.

Now, the question that needs to be answered is why one should have such an option and when a continuous formulation is more appropriate than a discrete one. To answer the above question, one needs to ask what is the goal of locational analysis. The ultimate goal of locational analysis is to provide decision makers with quantitative tools for finding good solutions to realistic locational problems. To justify the choice of formulation an analyst makes, many issues are often considered, such as:

(a) What is the nature of the network?
(b) Is the restriction to a finite number of potential sites a reasonable model for the particular network? Is this set of finite potential sites identifiable on the particular network?
(c) Is the error caused by the restriction of finite potential sites justifiable?
(d) Are the optimal solutions obtained by a continuous formulation "comparable" to the optimal solutions obtained by some discrete formulation?
(e) Are there considerable computational simplifications obtainable via either a discrete or a continuous formulation?

The notion of "comparability" is introduced in (d): We say a continuous formulation is "comparable" to a discrete formulation, when the optimal solutions to a discrete formulation are readily transferable to a continuous formulation without changing the magnitude of the optimal solutions. In general, the discrete and the continuous formulations do not yield "comparable" optimal solutions, but in the later chapters of this thesis, we will present one "comparable" formulation. Note, that a discrete formulation usually has simpler algorithms associated with it.

Due to the vast number of types of location problems available, this thesis will present a selection of location problems. The following is a summary of the subsequent chapters:

Chapter 1 fixes the notation used in this thesis, recalls a few concepts from graph theory and the terminology used in location theory.

Chapter 2 introduces the concept of $p$-median and shows that in this locational analysis, the optimal solutions provided by continuous and discrete formulations are
Chapter 3 introduces the concept of p-centre and shows why the continuous formulations are not "comparable" to the discrete one. Some algorithms and the concept of dual problems are investigated.

Chapter 4 is based on an approach originating in the recent research paper [7]. In that paper, the continuous Max-Min tree partition problem was treated. Here we treat the continuous Min-Max tree partition problem, where we find a polynomial algorithm for solving the continuous Min-Max p-partition on a tree with rational valued edge-lengths. The solution is obtained by solving a related discrete problem.

Chapter 5 presents the material of [7] on the continuous Max-Min tree partitions problem, where we present a polynomial algorithm for solving the continuous Max-Min p-partition on a rational valued edge-lengths tree. Again, the solution is obtained by solving a related discrete problem.

Chapter 6 contains conclusion and some computational results obtained from the continuous Max-Min p-partition algorithm.

The sources of the topics in this thesis are as follows:

Chapter 1-3, contains material studied in the literature.

Chapter 5 is based on the joint work with Becker and Simeone as reported in [7].

Chapter 4 is an original modification of the approach used in Chapter 5 applying to the Min-Max case.

Chapter 6 contains original computation experiments of the continuous Max-Min p-partition algorithm.
Chapter 1

Preliminaries

This chapter is an overview of terminology, notation and concepts, which will be used in the subsequent chapters of this thesis.

The thesis is written with the assumption that the reader has basic knowledge of graph theory and combinatorial theory. My primary reference of basic graph and combinatorial theory has been [4] and the reader may also consult [38] for general definitions and concepts in graph theory.

The basic reference for location theory is [50]. Furthermore, [58] provides an in-depth review on the topics of $p$-medians and $p$-centres. Before the completion of this thesis, I was made aware of a recent text by Daskin [17]. This reference seems to deal with the practical application of location theory.


1.1 Notation, Concepts and Terminology

A convenient way of modeling locational problems, is to model the problems using connected graphs with weighted edges and weighted vertices. The networks can usually be visualized as graphs with vertices representing junctions (such as intersections or cities on a road network, components on a circuit board) and edges representing the links (such as the road joining two cities, connections of two components).

A network $\mathcal{N}$ can be represented by a bar-and-joint graph, where a bar-and-joint graph is defined as follows: Let $\mathcal{G} = (\bar{V}, \bar{E})$, where $\bar{V}$ is a finite set of circumferences of
circles in the plane (joints) with the same radius \( p \), where \( p \) is a small positive number, and \( \bar{E} \) is a finite set of straight line segments of the plane (bars or edges), such that:

(i) any two different joints have an empty intersection;

(ii) any two different edges have an empty intersection;

(iii) the two endpoints of each edge lie on two different joints.

(iv) for any two different joints, there is at most one edge whose endpoints lie on the two joints.

A connected graph will be denoted by \( G = (V, E) \), where \( V \) is a finite set of vertices and \( E \) is a finite set of edges. In particular, a tree, that is an acyclic, connected graph, will be denoted by \( T \). We will always use the letter \( T \), possibly with subscripts or superscripts to denote a tree. Now, the assumptions made on \( G \) are that at most one edge joins any two distinct vertices and \( E \) contains no loops. We let \( |V| = n \) and \( |E| = \tau \), where the conventional notation of \( |S| \) is used to denote the cardinality of any finite set \( S \).

One can associate with any bar-and-joint graph \( \tilde{G} \) an abstract undirected graph \( G \) whose vertex-set is \( V \) and where two vertices \( a \) and \( b \) are adjacent iff there is an edge whose endpoints lie on \( a \) and \( b \), respectively. Whenever it is needed, we shall identify a graph with its associated abstract graph. All our abstract graphs will be assumed to have associated bar-and-joint graphs, in view of property (ii) above.

For each \( v \in V \) (\( e \in E \)), a non-negative, real valued weight (edge-length) is assigned which is denoted by \( w(v) \) (\( l(e) \)).

If a tree \( T \) is rooted, then any edge \( e = \{u, v\} \) of \( T \) can be labeled as \( (u, v) \), where \( u \) is closer to the root of \( T \) than \( v \). Furthermore, \( u \) is called the tail of \( e \), \( \text{tail}(e) \), and \( v \) is called the head of \( e \), \( \text{head}(e) \).

We give a formal definition of partitions and covers [16]: Given a set \( X \) and a family \( S = \{S_1, S_2, \ldots, S_k\} \) of sets \( S_j \subset X \), any subfamily \( S' = \{S_{j_1}, S_{j_2}, \ldots, S_{j_s}\} \) of \( S \) such that \( \cup_{i=1}^{s} S_{j_i} = X \) is called a set-covering of \( X \), and the \( S_{j_i} \) are called the covering sets. In addition, if \( S' \) also satisfies: \( S_{j_h} \cap S_{j_l} = \emptyset \forall \ h, l \in \{1, \ldots, s\}, h \neq l \), then \( S' \) is called a partition of \( X \). In this thesis, we deal with equipartition problems of graphs: let \( \pi = \{S_0, S_1, \ldots, S_{p-1}\} \) be a partition of \( V \) into \( p \) subsets, where the subgraph \( C_i \) induced by \( S_i \subset \pi \) is connected (\( 0 \leq i \leq p-1 \)). For a given \( G = (V, E) \), if \( P = \{C_0, C_1, \ldots, C_{p-1}\} \) then \( P \) is called a \( p \)-partition of \( G \) and \( C_i \) is called the \( i \)-th component of \( P \). The set
of all the $p$-partitions of $G$ is denoted by $\Pi(G,p)$.

Points in $G$ either are vertices or lie on imbedded edges joining vertices. In particular, a cut in a graph is a point which indicates the place where the particular components are split, and a tree $T$ together with cuts is called a dissected tree. A marker is either a cut or a vertex.

For any two points of a given $G$, $x$ and $y$ say, the length of the shortest path between $x$ and $y$ in $G$ is denoted by $d(x,y)$, and this is called the distance between $x$ and $y$ in $G$. For any finite subsets $X$ and $Y$ of $G$, the distance function $d(X,Y)$ is equal to $\min\{d(x,y) \mid x \in X \text{ and } y \in Y\}$. In a rooted $T$, any point $x \in e$ can be uniquely presented as $(\text{edge}(x), \text{dist}(x))$, where $\text{edge}(x) = e$ and $\text{dist}(x) = d(x, \text{head}(e))$. A segment of an edge, is a closed interval $s$ of the edge such that both endpoints of $s$ are markers, and no inner point of $s$ is a cut.

In a tree $T$ with root $r$, the terminology of descendant, ancestor, son and father can be used in relation to who is closer to $r$ on the path (for example, in $e = (u,v)$, $u$ is the father of $v$ and an ancestor of $v$ and $u$). A vertex $v$ is called a fork when $v$ is a tail of at least two edges. The down-tree of an edge $e$, $DT(e)$, is the subtree consisting of all descendants of $e$ ($e$ is included by this definition). The down-tree of a vertex $v$, $DT(v)$, is the union of all the $DT(e)$, where $v = \text{tail}(e)$. A partial down-tree of a vertex $v$ is the down-tree of one of edge $e$, where $v = \text{tail}(e)$, denote by $PDT(v,e)$.

Similar definitions of down-tree hold for the markers and the segments of a dissected rooted tree. Clearly, every subtree (in particular, every component) inherits the property of being rooted. The down-component of a marker $u$ is the set of all segments that belong to the down-tree of $u$ and do not belong to the down-tree of any cut which is a descendant of $u$. If $r$ is the unique marker lying on joint root, the down-component of $r$ is called the top component. The down-component of an edge $e$ (of a segment $s$) is the down-component of tail($e$) (of head($s$)). The cut-tree of a dissected rooted tree, denoted by $C$, is a rooted tree whose vertices are the cuts and $r$; and which has a directed edge $(c,c')$ iff in $T$ there is a directed path from $c$ to $c'$ without inner cuts. Clearly the cut-tree is rooted at $r$.

The level of a vertex $v \in T$ is the number of edges on the path joining the root to $v$. The height of $T$ is the maximum value of the level of a vertex in $T$. 

10
1.2 Problems, Algorithms and Complexity

This thesis aims to investigate the development of algorithms for solving certain loca­
tional decision problems formulated as optimization problems. This section will take a
brief look at the notion of computational complexity, which is a conceptual framework
commonly used in characterizing problems and algorithms.

Decision algorithms correspond to deterministic or non-deterministic Turing ma­
machines which, when they halt, output “yes” or “no”. The problem that such an algo­
rithm solves is called a decision problem. For detailed definitions of deterministic and
non-deterministic Turing machines see [24] page 23 to 27 and page 30 to 31.

We will present an informal description of the theory of NP-completeness:

• The class P is defined to be the class of all decision problems that can be solved by
deterministic algorithms with polynomial complexity.

• The class NP is defined to be the class of all decision problems that can be solved by
nondeterministic guess-and-verify algorithms with polynomial complexity. A nondeter­
ministic guess-and-verify algorithm can be viewed as being composed of two separate
stages, the first stage is the guessing stage and the second stage is the checking stage.

Given a problem instance $I$, the guessing stage merely “guesses” some structure $S$. Then
both $I$ and $S$ are provided as inputs to the checking stage, which proceeds to compute
in a normal deterministic manner, eventually halting with answer “yes” or “no”.

If for any instance of a decision problem $\pi'$, we can construct in polynomial time
an instance of a decision problem $\pi$ such that the instance of $\pi'$ has a “yes” answer iff
the instance of $\pi$ has a “yes” answer, then $\pi'$ is said to be Karp reducible to $\pi$, denoted
as $\pi' \preceq \pi$ (for detailed information on the Karp reducibility see [41]). If $\pi' \preceq \pi$ for all
$\pi' \in NP$ then any problem belonging to the class NP can be solved if $\pi$ can be solved
and $\pi$ is then called NP-hard. Finally $\pi$ is called NP-complete if $\pi$ is NP-hard and
$\pi \in NP$. NP-complete problems are thus the hardest problems in NP. A polynomial
solution of any problem in NP would imply that $P = NP$, since $P$ is a subset of NP.
It is not known whether $P = NP$, but it is generally believed that this is not the case.
Thus problems in NP are believed to be harder than those with polynomial algorithms.
It is a goal of the theory, when applied to a particular problem, to determine whether
it belongs to P, or to NP and not to P, or to neither.

Each optimization problem gives rise to a decision problem. A direct way of deriving
a related decision problem from a given instance of an optimization problem, is to
introduce a "threshold" value $K$ and ask the question: Does the given instance of the optimal problem have a feasible solution of value bounded by $K$? The optimization problem of finding the longest simple path in $G$, that is the maximum edge-length path passing through no vertex more than once is used here to illustrate the idea. A given instance of the longest simple path problem is a positive value edge-length for each $e \in E$, $V$ and the positive "threshold" value $K$. The question to be asked: Is there a simple path in $G$ between some two vertices of $V$ of length at least $K$?

If the corresponding decision problem can be demonstrated to be NP-complete, then the optimization problem is usually at least as difficult to solve as the corresponding decision problem; such an optimization problem is then termed NP-hard. In this way, even though the theory of NP-completeness is restricted to decision problems, the implications of the theory can be extended to optimization problems as well.

Suppose a problem is shown to be NP-complete. This problem might be modified, by relaxing some constraints or varying some parameters in the original problem. The simplified problem might be polynomially solvable. For locational decision problems, for example, the underlying networks might be restricted to those in which the graphs are planar, or bipartite, or acyclic. The complexities of those problems may be different in each case. The choice of which subproblems to analyze is determined by the application under consideration.

Given an optimization problem, we associate a complexity function for algorithms that solve the problem. Informally, the complexity function could be the number of operations taken to solve an instance of the problem, or the number of operations of a particular sort. Let $T(n)$ be the maximum complexity for problems of size $n$ of a given algorithm. A complexity function $T(n)$ is $O(g(n))$ if there exist a constant $c$ and a non-negative integer $n^*$ such that $|T(n)| \leq c|g(n)|$ for all values of $n \geq n^*$. A polynomial algorithm is defined to be one whose time complexity function is $O(p(n))$ for some polynomial function $p$. Any algorithm whose time complexity function cannot be so bounded is called a non-polynomial time algorithm. Polynomial algorithms are regarded as being "efficient".

This thesis will show that the $p$-median problem and the $p$-centre problem are NP-hard on general graphs and are polynomially solvable on acyclic graphs (trees). The thesis further presents polynomial algorithms for solving the continuous Max-Min tree partition problem and the continuous Min-Max tree partition problem respectively (at least, when the edge-lengths are rational).
Chapter 2

The $p$–Median Problem

This chapter deals with the $p$–median problem. $p$–Median problems are a class of optimal problems using the Mini-Sum criterion: Locate $p$ facilities so as to minimize the sum of the distances to each vertex from its nearest facility. In the next section, we will present a real-life scenario which can be modelled as a $p$–median problem. However, the $p$–median problem is not just applicable in the context of locating facilities for physical distribution; there are a large number of applications that can be modelled as $p$–median problems. Some generalizations of $p$–median problems will be presented at the end of this section. The $p$–median problem is the first optimal problem presented in this thesis, because the continuous formulation of the $p$–median problem is comparable to the discrete formulation of the $p$–median problem. Hence the $p$–median problem is a good example to illustrate the idea of comparability. Although the $p$–median problem can be solved in the discrete formulation, the general $p$–median problem on $G$ is NP-hard; a large section of this chapter is devoted to proving this fact.

Let $\mathcal{F}(G)$ be the class of subsets of $G$ with finitely many points, then the objective function $f : \mathcal{F}(G) \rightarrow \mathbb{R}$ is defined by $f(X_p) = \sum_{v_i \in V} w(v_i)d(v_i, X_p)$ where $X_p$ is a set of $p$ points in $G$. Any set of $p$ points in $G$, that minimizes $f$, is called an absolute $p$–median of $G$ (the continuous formulation). If the set of $p$ points that minimize $f$, is restricted to be from the set of vertices of $G$, it is called the vertex restricted $p$–median of $G$ (the discrete formulation). By a result of Hakimi [31] [32], there exists an absolute $p$–median consisting entirely of vertices of $G$. For this reason, the distinction between the continuous formulation and the discrete formulation is not usually of practical significance.

In this chapter, we will introduce the classical $p$–median problem described above and present Hakimi’s result (the Vertex Optimal Theorem). However, there exist many
interesting and natural generalizations of $p$-median problems, which will not be con­sidered in this thesis, such as: Minieka [48] extended the formulation of the absolute $p$-median problem on $G$, for the case where every point in $G$ is taken into consideration. Goldman [27] generalized the results to accommodate the case where one distinguishes a vertex as being a source or a destination. Hakimi and Maheshwari [33] dealt with the case of multiple commodities that go through multiple stages. To illustrate the term multiple commodities, we will use the following example: To locate $p$ emergency response centres, which house fire fighting, ambulances and police response units, on a network. Hence in this case, we wish to locate $p$ facilities to provide multiple services. It is clear that for each service there will generally be a different demand pattern and different operation costs. The formulation must take this into account. Mirchandani [50] discusses extensively a portion of the network location literature, which involves deterministic and probabilistic cases.

The organization of this chapter is as follows: Section 1 presents the formulation of the $p$-median problem. Section 2 proves the Vertex Optimal Theorem. Section 3 introduces the complexity of the $p$-median problem on $G$. Section 4 provides an algorithm for solving 1-median problem on $T$.

2.1 The $p$-Median

With the aim of providing a better insight into the formulation of a $p$-median problem, a basic simple scenario is presented: In a road transportation network, the manufacturing sector may wish to set up $p$ supply depots, such that the total transportation cost from the $p$ depots to all the demand points on the network is as small as possible. Now, suppose the network is modelled as a graph $G$, and the clients are modelled as vertices on $G$. The quantity demanded by each client is represented by the vertex weight and the transport cost per quantity at each route is represented by the edge-length. Hence, the $p$-median problem can be described as: Find the locations of supply depots, such that the total transportation cost is at minimum. So let $X_p$ be any set of $p$ points on $G$. Then the set of $p$ points $X_p^*$ on $G$ is a $p$-median of $G$, if for any $X_p$ on $G$, the following holds:

$$f(X_p^*) = \sum_{v_i\in V} w(v_i)d(v_i, X_p^*)$$
$$\leq \sum_{v_i\in V} w(v_i)d(v_i, X_p)$$
$$= f(X_p).$$
2.2 Comparability and the Vertex Optimal Theorem

The intention of this thesis is to formulate the location problems in a continuous fashion. However, this is not necessary in the $p$-median problem, since Hakimi [31] [32] has shown that for any absolute $p$-median of $G$, there exist an absolute $p$-median, with points situated only on the vertices of $G$. This theorem shows that a continuous formulation is "comparable" to a discrete formulation. Here is the Vertex Optimal Theorem:

**Theorem 2.2.1** There exist a subset $V_p$ of $V$ containing $p$ distinct vertices, $1 \leq p \leq n - m$, such that for every set of $p$ points $X_p$ on $G$, we have:

$$f(X_p) = \sum_{i=1}^{n} w(v_i) d(v_i, X_p)$$

$$\geq \sum_{i=1}^{n} w(v_i) d(v_i, V_p)$$

$$= f(V_p)$$

**Proof:** For a given $p$, let $G_p = \{X'_p | f(X'_p) \leq f(X_p), \forall X_p \}$ be the set of all $p$-medians of $G$. Now, let $X'_p \in G_p$ be the $p$-median with the least number of nonvertex centres. If $X'_p$ contains no nonvertex centre, then the theorem is proved. If not, then we have two cases to consider:

Case 1: Let $x \in X'_p$ be a nonvertex centre and let $x$ be the only centre on edge $e = \{u, v\}$. Now, let $V_v$ be the set of vertices which have the shortest path from $x$ in $X'_p$ passing through $v$, and let $V_u$ be the set of vertices not in $V_v$ which have the shortest path from $x$ in $X'_p$ passing through $u$ (by assigning $V_u$ first, we ensure that $V_u \cap V_v = \phi$). Now, let $d(u, x) = l$ then $d(x, v) = l(e) - l$. Now we have:

$$f(X'_p) = \sum_{v \in V} w(v_r) d(v_r, X'_p)$$

$$= \sum_{v \in V \setminus \{u, v\}} w(v) d(v, X'_p \setminus \{x\}) + \sum_{v \in V_u} w(v_j) d(v_j, x) + \sum_{v \in V_v} w(v_k) d(v_k, x)$$

$$= \sum_{v \in V \setminus \{u, v\}} w(v) d(v, X'_p \setminus \{x\}) + \sum_{v \in V_u} w(v_j) (d(v_j, u) + l)$$

$$+ \sum_{v \in V_v} w(v_k) (d(v_k, v) + (l(e) - l))$$
\[
= \sum_{v_i \in V_u \cup V_v} w(v_i) d(v_i, X'_p \setminus \{x\}) + \sum_{v_j \in V_u} w(v_j) d(v_j, u) + \sum_{v_j \in V_u} w(v_j) l \\
+ \sum_{v_k \in V_v} w(v_k) d(v_k, v) + \sum_{v_k \in V_v} w(v_k) (l(e) - l)
\]

Now, if \( \sum_{v_j \in V_u} w(v_j) \geq \sum_{v_k \in V_v} w(v_k) \), then we can transfer the centre from \( x \) to \( u \), since we have:

\[
f(X'_p) = \sum_{v_i \in V_u \cup V_v} w(v_i) d(v_i, X'_p \setminus \{x\}) + \sum_{v_j \in V_u} w(v_j) d(v_j, u) + \sum_{v_j \in V_u} w(v_j) l \\
+ \sum_{v_k \in V_v} w(v_k) d(v_k, v) + \sum_{v_k \in V_v} w(v_k) (l(e) - l)
\]

Similarly, if \( \sum_{v_j \in V_u} w(v_j) < \sum_{v_k \in V_v} w(v_k) \), then we can transfer the centre from \( x \) to \( v \).

Case 2: Suppose there are more than one centre on the edge \( e \). Let \( x \) be the centre closer to \( u \) than any other centre on \( e \). Let \( y \) be the next closest centre to \( u \) on \( e \). Now we can treat \( x \) on the segment \([u, y]\) similarly to the approach used in Case 1. Thus, \( x \) is either transferred to \( u \) or to \( y \). We continue this process, until all the non-centres on \( e \) are either transferred to \( u \) or to \( v \).

Now, in both cases, we are able to transfer one or many of the nonvertex centres of \( X'_p \) on \( e \) onto the vertex. Let \( V'_p \) denote the set of \( p \) points after transferring one of the nonvertex centres of \( X'_p \) onto a vertex. Since \( X'_p \) is a \( p \)-median, we have \( f(X'_p) = f(V'_p) \); thus \( V'_p \in G_p \), which contradict the fact \( X'_p \in G_p \) is the \( p \)-median with the least number of nonvertex centres.

2.3 The Complexity of the \( p \)-Median Problem

The problems of locating supply depots to minimize the transport cost, can be simplified, using Theorem 2.2.1, to restrict the search on the vertices of \( G \). However, it has been
shown to be NP-hard [40], even if the network considered is a planar graph of maximum vertex degree 3 and all edges and vertices are of the length 1 and weight 1 respectively.

Here is the outline of how Kariv and Hakimi [40] proved that the general problem of finding a p-median of a network is NP-hard. They used the NP-completeness of the vertex cover problem on a planar graph of maximum vertex degree 3, which was proved by Garey and Johnson [25]. Then Kariv and Hakimi [40] show that the problem of finding a dominating set on a planar graph of maximum vertex degree 3 is reduces to the cover problem, thus is NP-complete. They then proved that the dominating set problem is polynomial time reducible to the p-median problem. Therefore, there exists a polynomial time algorithm for the p-median only if P = NP, hence the p-median problem is NP-hard.

To start with, the vertex cover problem is defined as follow: Is there a vertex cover of size $k$ or less for $G$, that is, a subset $V' \subseteq V$ such that $|V'| \leq k$ and, for each edge $\{u, v\} \in E$, at least one of $u$ and $v$ belongs to $V'$? This problem is shown to be NP-complete in [24].

Garey and Johnson [25] shown that a stronger result can be derived from the NP-completeness of the vertex cover problem:

**Lemma 2.3.1** The following problem is NP-complete: Is there a vertex cover of size $k < n$ or less for a planar graph $G$ with maximum vertex degree 3?

**Proof:** Given a planar graph $G$ and an integer $k$, we construct a planar graph $G'$ with no vertex degree exceeding 3 and an integer $k'$ such that $G'$ has a vertex cover of size $k'$ iff $G$ has a vertex cover of size $k$.

Let $V = \{v_1, v_2, \ldots, v_n\}$. The construction begins with a fixed planar representation of $G = G_0$. For each integer $i$, from 1 up to $n$, we construct a planar representation for a graph $G_i$ from that for $G_{i-1}$ as follows:

1. Let $\{v_i, w_1\}, \{v_i, w_2\}, \ldots, \{v_i, w_p\}$ be the edges incident from $v_i$ in the order that they occur around $v_i$ in the planar representation of $G_{i-1}$.

2. Replace $v_i$ with a cycle consisting of the new vertices $u_i(j), v_i(j), 1 \leq j \leq n$ and the new edges $\{u_i(j), v_i(j)\}, 1 \leq j \leq n, \{v_i(j), u_i(j+1)\}, 1 \leq j \leq n - 1$, and $\{v_i(n), u_i(1)\}$.

3. Replace each edge $\{v_i, w_t\}$ by the edge $\{v_i(j), w_j\}$, add a new vertex $z_i$, and add the edge $\{u_i(1), z_i\}$.

17
Finally we set $G' = G_n$ and $k' = n^2 + k$. Observe that $G'$ has no vertex with degree exceeding 3.

Now suppose $V^*$ is a vertex cover for $G$ satisfying $|V^*| \leq k$. Then let

$$V_1^* = \{v_i(j)|v_i \in V^*, 1 \leq j \leq n\} \cup \{u_i(1)|v_i \in V^*\}$$

$$\cup \{u_i(j)|v_i \notin V^*, 1 \leq j \leq n\}$$

(2.1)

$$V' = \{v_i(j)|v_i \in V^*, 1 \leq j \leq n\} \cup \{u_i(1)|v_i \in V^*\}$$

where $V^* \cup U'$

(2.2)

and $U' = \{u_i(j)|v_i \notin V^*, 1 \leq j \leq n\}$.

We still need to check if $V_1^*$ is a vertex cover for $G'$ satisfying $|V_1^*| \leq k'$.

CLAIM 1: $|V_1^*| = |V^*| + n^2$.

Proof: There are $n|V^*|$ elements in the first subset of the right-hand side of equation 2.2, since for each $v_i \in V^*$ there are $n$ elements of $v_i(j)$. There are $|V^*|$ elements in the second subset of the right-hand side of equation 2.2. There are $n(n - |V^*|)$ elements in the last subset of the right-hand side of equation 2.2, since there are $n - |V^*|$ elements in $V \setminus V^*$, and for each $v_i \in V \setminus V^*$ there are $n$ elements of $v_i(j)$. There are thus a total of $n|V^*| + |V^*| + n(n - |V^*|)$ elements in $V_1^*$. Now, CLAIM 1 is proved.

CLAIM 2: If $V^*$ is a vertex cover for $G$ then $V_1^*$ is a vertex cover for $G'$.

Proof: We will examine each type of edges in $G'$ with respect to the vertices in $V^*$, to prove that every edge in $G'$ is covered by $V_1^*$. For any $v_i \in V$, we have:

- $\{u_i(j), v_i(j)\}, 1 \leq j \leq n$; $\{v_i(j), u_i(j + 1)\}, 1 \leq j \leq n - 1$; $\{v_i(n), u_i(1)\}$; $\{u_i(1), z_i\}$:
  For these four cases, if $v_i \in V^*$ then the first vertex of each type of edge is in $V_1^*$, since such a vertex is in $V'$ of equation 2.2, else by referring to $U'$ of equation 2.2, we have that the second vertex of each type of edge is an element of $V_1^*$.

- $\{v_i(j), w_j\}$: If $v_i \in V^*$ then $v_i(j) \in V_1^*$ because $v_i(j)$ is in $V'$ of equation 2.2, else since $\{v_i(j), w_j\}$ is an edge in $G$ and $V^*$ is a vertex cover for $G$, which implies that $w_j$ must be in $V^*$. By construction Step 3, we have $w_j = v_i(j'), 1 \leq l \leq n$, and by referring to $V'$ of equation 2.2, we have $v_i(j') \in V_1^*$.

So CLAIM 2 is proved, since every edge in $G'$ is covered by $V_1^*$.

Now, we have shown that $V_1^*$ is a vertex cover for $G'$ and $|V_1^*| \leq n^2 + k$. 18
Conversely, suppose \( V'_1 \) is a vertex cover for \( G' \) satisfying \( |V'_1| \leq k' \). Since the only vertices of \( G' \) that cover edges corresponding to edges of \( G \) are the \( v_i(j) \) vertices, we immediately know that the set \( V^* = \{ v_i | \exists j, 1 \leq j \leq n, v_i(j) \in V'_1 \} \) must form a vertex cover for \( G \). We shall show that \( |V^*| \leq k \). First we note that we may assume that \( u_i(1) \in V'_1 \) for every \( i \), since the edge \( \{ u_i(1), z_i \} \) must be covered and \( z_i \) only has degree 1. Define, for \( 1 \leq i \leq n \), \( S_i = V'_1 \cap \{ u_i(j), v_i(j) | 1 \leq j \leq n \} \). In order to cover all \( 2n \) edges in the cycle for \( v_i \) we must have \( |S_i| \geq n \). Since \( k' = n^2 + k \), this implies that at most \( k \) values of \( i \) can satisfy \( |S_i| > n \). Furthermore, since \( u_i(1) \in S_i \), the only set of exactly \( n \) vertices that covers all \( 2n \) edges in the cycle for \( v_i \) is \( \{ u_i(j) | 1 \leq j \leq n \} \). Thus if there exists a \( j \) for which \( v_i(j) \in S_i \), we must have \( |S_i| > n \). Since this occurs for at most \( k \) values of \( i \), we have \( |V^*| \leq k \), and \( V^* \) is the desired vertex cover for \( G \).

Since \( G' \) can clearly be constructed in polynomial time with respect to the size of \( G \), and has the desired vertex cover if \( G \) does, our transformation works as required, and the restricted problem is NP-complete.

Using the fact that the problem of finding a vertex cover of size \( k < n \) or less for a planar graph \( G \) with maximum vertex degree 3 is NP-complete, the following lemma can be proved:

**Lemma 2.3.2** The following problem is NP-complete: Given a planar graph \( G \) of maximum vertex degree 3 and a positive integer \( k' \), does there exist a subset \( V^* \) of size \( k' \) or less vertices, such that each vertex of \( G \) is either in \( V^* \) or is adjacent to a vertex of \( V^* \).

**Proof:** Let \( G \) be a planar graph with maximum degree 3 and \( k \) a positive integer. We construct a planar graph \( G' \) with no vertex degree exceeding 3 and an integer \( k' \), such that \( G' \) has a dominating set of size not greater than \( k' \) iff \( G \) has a vertex cover of size not greater than \( k \).

Let \( V = \{ v_1, v_2, \ldots, v_n \} \) and \( E = \{ e_1, e_2, \ldots, e_r \} \). The construction begins with a fixed planar representation of \( G = G_0 \). For each integer \( i \), from 1 up to \( r \), we construct a planar representation for a graph \( G_i \) from that for \( G_{i-1} \) as follows:

1. Let \( e_i = \{ v_i, v_j \} \) be the \( i \)-th edge of \( E \) in the planar representation of \( G_{i-1} \).
2. Introduce 4 new vertex \( v_{ij}, v_{ji}, u_{ij} \) and \( u_{ji} \). Now replace \( \{ v_i, v_j \} \) with the following 6 edges: \( \{ v_i, v_{ij} \}, \{ v_{ij}, u_{ij} \}, \{ v_{ij}, v_{ji} \}, \{ u_{ji}, v_{ji} \}, \{ u_{ij}, v_{ij} \} \) and \( \{ u_{ji}, v_j \} \).

Finally we set \( G' = G_r \) and \( k' = |E| + k \). We will call those vertices \( v_l, l = 1, 2, \ldots, n \)
in $G'$, the "pseudo" vertices. Let $S_i^{ij} = \{v_{ij}, v_{ji}, u_{ij}, u_{ji}, v_i, v_j| e_i = \{v_i, v_j\} \in E\}$. Note that both $G$ and $G'$ has no vertex with degree exceeding 3.

![Diagram](image)

Figure 2.1: An example of the construction on an edge $e_i = \{v_i, v_j\}$.

Now suppose $V^*$ is a vertex cover for $G$ satisfying $|V^*| \leq k$. Then let

$$V_i^* = V^* \cup \{v_{ij} \in G'| v_i \in V^* \text{ and } v_j \notin V^*\} \cup \{v_{jl} \in G'| v_j \in V^* \text{ and } v_l \notin V^*\} \cup \{v_{jl} \in G'| j < l \text{ and } v_i, v_j \in V^*\}. \quad (2.3)$$

$$= V^* \cup V'_j \cup V'_i \cup V'_{jl} \quad (2.4)$$

where $V^*, V'_j, V'_i, V'_{jl}$ represent the subsets in equation 2.3 respectively.

**CLAIM 1:** If $|V_i^*| = |V^*| + \tau$.

**Proof:** Since $V^*$ is a vertex cover for $G$, by definition, any edge $\{v_i, v_j\} \in E$ has either $v_j$ or $v_i$ as an element of $V^*$. This implies that the union of the last three subsets of the right-hand side of equation 2.3 is equal to the set consisting of one of the non-pseudo vertices from each $S_i^{ij}$. There are $\tau$ edges in $E$, which implies there are $\tau$ sets of $S_i^{ij}$. Thus, **CLAIM 1** is proved.

**CLAIM 2:** If $V^*$ is a vertex cover for $G$ then $V_i^*$ is a dominating set for $G'$.

**Proof:** We will prove the claim by using induction on the edges of $G'$ to prove that every vertex in $S_i^{ij}$ is adjacent to an element of $V_i^*$ or is in $V_i^*$.

For any $S_i^{ij}$ where $e_i = \{v_i, v_j\} \in E$ we have:

- If $v_j, v_l$ are both in $V^*$ and without loss of generality let $j < l$ then $v_j, v_l \in V_i^*$, which is obvious from equation 2.4. Furthermore, by referring to $V_i'$ of equation 2.4 we have $v_{jl} \in V_i^*$ and hence $u_{jl}$ and $u_{ij}$ are adjacent to $v_{jl}$ and $v_{ij}$ is adjacent to $v_l$.

- If $v_j \in V^*$ and $v_l \notin V^*$ ($v_l \in V^*$ and $v_j \notin V^*$) then $v_j$ ($v_l$) is in $V_i^*$, this is obvious.
from equation 2.4 and \( v_{ij} (v_{ji}) \) is in \( V'_i (V'_j) \) of equation 2.4. Now, \( v_i, u_{ij} \) and \( u_{ij} (v_{ij}, u_{ji}) \) are adjacent to \( v_{ij} (v_{ji}) \) and \( v_{ji} (v_{ij}) \) is adjacent to \( v_j (v_i) \).

Since \( S_i^{ij} \) is chosen arbitrarily, it is true for every \( S_i^{ij} \) that every vertex in \( S_i^{ij} \) is adjacent to an element of \( V_1^* \) or is in \( V_1^* \). Thus, CLAIM 2 is proved, since \( V_1^* \) is a dominating set of \( G' \).

Now, we have shown that \( V_1^* \) is a dominating set for \( G' \) and \( |V_1^*| \leq \tau + k \).

Conversely, suppose \( V_1^* \) is a dominating set for \( G' \) satisfying \( |V_1^*| \leq k' \), we will show that there exist a dominating set of \( G' \), \( V'_2 \) say, which is derived from \( V_1^* \) and \( |V_1^*| = |V_2^*| \). Furthermore, we will show that if \( V_2^* \) is such a dominating set then \( G \) has a vertex cover of size not greater than \( k \).

CLAIM 3: Given \( V_1^* \) a dominating set of \( G' \), the following are true: If \( v_j, v_i \in S_i^{ij} \) and \( v_j, v_i \notin V_1^* \) then there are at least two non-pseudo vertices in \( S_i^{ij} \) which are elements of \( V_1^* \).

Proof: We will prove this result by contradiction. It is obvious that \( |S_i^{ij} \cap V_1^*| \neq 0 \), since \( V_1^* \) is a dominating set of \( G' \) implies that \( v_{ij}, v_{ji}, u_{ij} \) and \( u_{ji} \) must either be adjacent to some element of \( V_1^* \) or are elements of \( V_1^* \). Assume \( |S_i^{ij} \cap V_1^*| = 1 \), then one of \( v_{ij}, v_{ji}, u_{ij} \) or \( u_{ji} \) is in \( S_i^{ij} \cap V_1^* \). If \( v_{ij} \in S_i^{ij} \cap V_1^* \) (respectively \( v_{ji} \in S_i^{ij} \cap V_1^* \) then \( v_{ji} \) (respectively \( v_{ij} \)) is not adjacent to any element of \( V_1^* \). If \( u_{ij} \in S_i^{ij} \cap V_1^* \) (respectively \( u_{ji} \in S_i^{ij} \cap V_1^* \) then \( u_{ji} \) (respectively \( u_{ij} \)) is not adjacent to any element of \( V_1^* \). Hence contradict the fact that \( V_1^* \) is a dominating set of \( G' \).

Now, we will derive \( V_2^* \) from \( V_1^* \) by replacing the elements in each \( S_i^{ij} \cap V_1^* \), \( i = 1, 2, \ldots, \tau \). Let \( S_i^{ij} \cap V_1^* \) be denote by \( V_1^* \).

For each \( S_i^{ij} \) do the following:

- If \( v_j \) or \( v_i \) is in \( V_1^* \) then let \( V_2^* = V_1^* \).
- If both \( v_j \) and \( v_i \) are not in \( V_1^* \) then by CLAIM 3, there are at least two elements of \( \{v_{ij}, v_{ji}, u_{ij}, u_{ji}\} \) are in \( V_1^* \), hence we need to consider four possible cases:

1 & 2. If \( v_{ij} \in V_1^* \) (\( v_{ji} \in V_1^* \)) and \( v_{ji} \notin V_1^* \) (\( v_{ij} \notin V_1^* \)) then replace the vertex \( w \), where \( w \neq v_{ij} \) (\( w \neq v_{ji} \)) and \( w \in V_1^* \), by \( v_j \) (\( v_i \)). Let \( V_2^* = \{v_j\} \cup V_1^* \setminus \{w\} \)

3. If \( v_{ij}, v_{ji} \in V_1^* \) then arbitrary choose one, \( v_{ji} \) say, and replace \( v_{ji} \) by \( v_j \). Let \( V_2^* = \{v_j\} \cup V_1^* \setminus \{v_{ji}\} \).
4. If $v_{ij}, v_{ji} \notin V_{2(i)}^*$ then arbitrary choose one, $u_{ji}$ say, and replace $u_{ji}$ by $v_{ji}$ and replace $u_{ij}$ by $v_{i}$. Let $V_{2(i)}^* = \{v_i, v_{ji}\}$.

Now 3 observations can be made:

[i]. For any $S_{ij}^t$, either $v_i$ or $v_j$ is in $V_{2(i)}^*$.

[ii]. $|V_{2(i)}^*| = |V_{1(i)}^*|, \forall i \in \{1, 2, \ldots, \tau\}$. This is true, since we are replacing the vertices in $V_{1(i)}^*$ by the same amount of vertices in $V_{2(i)}^*$.

[iii]. The vertices which are adjacent to some vertices of $V_{1(i)}^*$ or in $V_{1(i)}^*$ are still adjacent to some vertices of $V_{2(i)}^*$ or in $V_{2(i)}^*$.

Let $V_2^* = \cup_{i=1}^\tau V_{2(i)}^*$, thus $|V_2^*| = |V_1^*|$ by observation [iii]. If $V_{1(i)}^*$ is a dominating set of $G'$ then $V_{2(i)}^*$ is a dominating set of $G'$, using observation [iii].

Let $V^* = V_2^* \cap V$. $V^*$ is a cover for $G$. Since the only vertices of $G'$ that correspond to edges of $G$ are the pseudo vertices in each $S_{ij}^t$, by observation [i], we have that each edge is covered by $V^*$.

**CLAIM 4:** If $V_2^*$ a dominating set of $G'$, then $|V^*| \leq k$.

**Proof:** In each $S_{ij}^t$, the pseudo vertices are adjacent to $v_{ij}$ and $v_{ji}$ only. Thus to make sure that $V_2^*$ is a dominating set for $u_{ij}$ and $u_{ji}$, at least one of the non-pseudo vertex in $S_{ij}^t$ must an element of $V_2^*$. This implies there are at least $\tau$ elements in $V_2^*$ are the non-pseudo vertices. So we have $|V_2^* \cap V| \leq k$.

Since $G'$ can clearly be constructed in polynomial time with respect to the size of $G$, and has the desired dominating set if $G$ has the desired vertex cover. By Lemma 2.3.1 the vertex cover problem on $G$ is NP-complete, thus the dominating set problem is NP-complete.

**Theorem 2.3.3** The problem of finding a $p$-median is NP-hard even in the case when the network is a planar graph of maximum vertex degree 3 all whose edges are of the length 1 and all whose vertices have weight 1.

**Proof:** Let $G$ be a planar graph of maximum vertex degree 3, all of whose edges are of length 1 and all of whose vertices have weight 1. We need only to show that the problem of whether there exists a dominating set of cardinality $p$ in $G$ is polynomial time reducible to the problem of finding a $p$-median of $G$. Let $V_p$ be an arbitrary subset of $p$ vertices of $G$. The edges in $G$ are of the length one, which implies $f(V_p) = \sum_{v \in V} d(v, V_p) \geq n - p.$
Thus, if there exists any subset $V_p^*$ for which $f(V_p^*) = n - p$ holds, then $V_p^*$ is a $p$-median of $G$. On the other hand, the equation $f(V_p^*) = n - p$ is satisfied iff $d(v, V_p^*) = 1$ for each of the $n - p$ vertices not in $V_p^*$, namely, iff $V_p^*$ is a dominating set of cardinality $p$ in $G$. Therefore, there exists a dominating set of cardinality $p$ in $G$ iff $f(V_p^*)$ of a $p$-median $V_p^*$ of $G$ is $n - p$. This shows that the problem of finding a dominating set in $G$ is polynomial time reducible to the problem of finding a $p$-median of $G$, and with the result of Lemma 2.3.2, we have show that the latter problem is NP-hard.

2.4 The $p$-Median Problem on Tree Networks

In the previous section, we have shown that the general $p$-median problems are NP-hard, even when the network is a planar graph of maximum vertex degree 3. However, there are polynomial algorithms devised to solve the special case of $p$-median problems on a tree network. This is possible because of the acyclicity of a tree, which implies the uniqueness of path between any two points in a tree. Kariv and Hakimi [40] presented an algorithm which finds a $p$-median ($p > 1$) of a tree in time $O(n^2 p^2)$. This algorithm is very lengthy and hence we will not present it. However, Goldman's algorithm [28] for finding a 1-median in a tree will be introduced in the next section.

2.4.1 Algorithm for Finding a 1-Median

To present Goldman's algorithm [28], new notation is needed. Let $e = \{u, v\}$ be an edge in $T$. Define $T_v = (V_v, E_v)$ and $T_u = (V_u, E_u)$ to be the connected components of $T$ with $v$ a vertex in $T_v$, $u$ a vertex in $T_u$, where edge $e = \{u, v\}$ is removed from $T$.

The algorithm is based on the following two lemmas:

**Lemma 2.4.1** If $w(V_v) \geq w(V_u)$, then $T_v$ contains at least one optimal location.

**Proof:** It is sufficient to prove that for any vertex $y \in V_u$, we have $f(y) \geq f(v)$, and the proof is as follow:

$$f(y) = \sum_{x \in V} w(x)d(x, y) \quad \text{(by the definition of } f)$$

$$= \sum_{x \in V_v} w(x)d(x, y) + \sum_{x \in V_u} w(x)d(x, y) \quad \text{ (} \{V_v, V_u\} \text{ is a partition of } V)$$
\[
= \sum_{x \in V_v} w(x)(d(x, v) + d(v, y)) + \sum_{x \in V_u} w(x)d(x, y)
\]
\[
= w(V_v)d(v, y) + \sum_{x \in V_v} w(x)d(v, x) + \sum_{x \in V_u} w(x)d(x, y) \quad (\{v, u\} \text{ is the unique edge between } V_v, V_u)
\]
\[
\geq w(V_v)d(v, y) + \sum_{x \in V_v} w(x)d(v, x) + \sum_{x \in V_u} w(x)d(x, y) \quad (w(V_u) \leq w(V_v))
\]
\[
= \sum_{x \in V_v} w(x)(d(y, x) + d(v, y)) + \sum_{x \in V_u} w(x)d(x, v)
\]
\[
\geq \sum_{x \in V_v} w(x)d(x, v) + \sum_{x \in V_u} w(x)d(x, v) \quad \text{(triangle inequality)}
\]
\[
= f(v)
\]

Lemma 2.4.2 If \(w(V_v) \geq w(V_u)\), then finding a 1-median of \(T\) is equivalent to finding a 1-median for \(T_v\) except that \(w'(v) := w(v) + w(V_u)\).

Proof: By Lemma 2.4.1, the search for a 1-median of \(T\) can be restricted to the subset \(V_v\) of \(V\). We will show that this restricted problem is equivalent to finding a 1-median for \(T_v\), with the vertex weights modified as stated above. Let \(f'\) be the objective function for \(T_v\), with the vertex weights modified. Now for any vertex \(y \in V_v\), we have

\[
f(y) = \sum_{x \in V_v \setminus \{v\}} w(x)d(x, y) + w(v)d(y, v) + \sum_{x \in V_u} w(x)d(x, y)
\]
\[
= \sum_{x \in V_v} w(x)(d(x, v) + d(v, y)) + w(v)d(y, v) + \sum_{x \in V_v \setminus \{v\}} w(x)d(x, y)
\]
\[
= (w(V_v) + w(v))d(y, v) + \sum_{x \in V_v} w(x)d(x, v) + \sum_{x \in V_v \setminus \{v\}} w(x)d(x, y)
\]
\[
= f'(y) + \sum_{x \in V_u} w(x)d(x, v)
\]

Since the objective functions of the two problems differ only by the following constant \(\sum_{x \in V} w(x)d(x, v)\), the problems are equivalent.

GOLDMAN'S 1-MEDIAN ALGORITHM

1. If \(T\) consists of a single vertex then Stop; that vertex is the 1-median.
2. Search for an terminal vertex \( v_i \). If \( w(v_i) \geq \frac{w(V)}{2} \) go to Step 4, otherwise go to Step 3.

3. Let \( v_j \) be the adjacent vertex of \( v_i \). Modify \( T \) by deleting \( v_i \) and the link \([v_i, v_j]\) and then incrementing \( w(v_j) \) by \( w(v_i) \). Now, return to Step 1.

4. Stop; \( v_i \) is a 1-median.

In Step 2, if the algorithm used an efficient data structure, such as an array, then the search for an terminal vertex can be compute in \( O(1) \). This implies that the time complexity of this algorithm is \( O(n) \), since edge-lengths do not enter into the algorithm and there are \( n \) vertices in \( T \). We will prove the correctness of the algorithm in the next theorem.

**Theorem 2.4.3** Goldman's 1-median algorithm is valid.

**Proof:** This theorem is proved by induction on the number of vertices in the tree.

For any tree with one vertex \( v \), it is obvious that the algorithm is correct. This is because \( v \) is the 1-median of the tree and the algorithm finds \( v \), then terminates at Step 1.

Assume that the algorithm is correct for all trees with fewer than \( n \) vertices. Let \( T \) be a tree with \( n \) vertices. Let \( u \) be the terminal vertex chosen by the algorithm in Step 2 and let \( v \) be the vertex adjacent to \( u \). The proof needs to consider the following cases:

- If \( w(u) \geq \frac{w(V)}{2} \) then the algorithm terminates at Step 4 and \( u \) is the 1-median. Since \( u \) is a terminal vertex which implies \( w(T_u) \geq \frac{w(V)}{2} \). Hence \( w(T_u) \geq w(T_v) \), and by Lemma 2.4.1 \( T_u \) contains at least one 1-median, so \( u \) is a 1-median.

- If \( w(u) < \frac{w(V)}{2} \) then the algorithm will modify \( T \) by remove the vertex \( u \) and then incrementing \( w(v) \) by \( w(u) \). By Lemma 2.4.2 finding a 1-median of \( T \) is equivalent to finding a 1-median for the modified tree. Since the modified tree has number of vertices less than \( n \), by the induction hypothesis, the algorithm is correct for the modified tree. Hence the algorithm is correct for \( T \).

So the algorithm is valid for any tree with finitely many vertices. \( \blacksquare \)
Chapter 3

The $p$–Centre Problem

In the previous chapter, the $p$–median problem was discussed; this chapter we will deal with the $p$–centre problem. $p$–Centre problems are a class of optimal problems using the Min-Max criterion: Locate $p$ facilities so that the maximum distance to a demand from its nearest facility is minimized.

The $p$–median problem and the $p$–centre problem are studied extensively in the network locational literature. The purpose of these two chapters is to provide insights for solving other network locational problems.

In this chapter, various formulations of a $p$–centre problem will be given in the next section. Like the $p$–median problem, the $p$–centre problem is used in the modelling of a wide range of applications. In the next section, a real-life scenario will be used to illustrate the modelling of the $p$–centre problem.

Unlike the $p$–median problem, the absolute formulation of the $p$–centre problem is not comparable to the discrete formulation of the $p$–centre problem. This result will be proved in the second section of this chapter. Furthermore, two algorithms are reviewed, and the general $p$–centre problems on $G$ are shown to be NP-hard. The rest of this chapter is devoted to the continuous formulation of a $p$–centre problem and an efficient algorithm for solving the continuous $p$–centre problem on $T$ is presented.

Again, we will denote the objective function by $f$. To define $f$, two sets of points $Y$ and $S$ on $G$ are specified. Both $Y$ and $S$ may be finite or infinite in cardinality, and they represent the set of demand points and the set of potential supply points respectively. In the case of the $p$–centre problem, if $\mathcal{F}(G)$ is the class of subsets of $G$ with finitely many points then $f : \mathcal{F}(G) \to \mathbb{R}$ is defined as: $f(X_p) = \max_{y \in Y} \{ w(y)d(y, X_p) | X_p \subseteq S \}$,
where $X_p$ is any set of $p$ points in $S$. If $S = G$ and $Y = V$ then any set $X^*$ of $p$ points in $G$ that minimizes $f$ is called an absolute $p$-centre of $G$ ($f(X^*) = \min\{f(X_p) | X_p \subset G\}$). If $S = V$ and $Y = V$ then any set $V^*$ of $p$ points in $V$ that minimizes $f$ is called a discrete $p$-centre of $G$ ($f(V^*) = \min\{f(X_p) | X_p \subset V\}$). Furthermore, the value $f(X^*)$ (respectively $f(V^*)$) is called the absolute $p$-radius of $G$ (respectively the discrete $p$-radius of $G$), denoted by $r_p$ (respectively $r_p^{(v)}$). In the literature, $p$-centre problems on $G$ whose vertices are equally weighted are usually referred to as unweighted problems, and they are referred to as weighted problems otherwise.

The general problems were shown to be reducible to computationally finite problems. In the unweighted graph $G$, Minieka [49] identifies a finite point set in $G$, where there exists an optimal solution $X^*$ in this finite point set. This result enables the search for the solution of the $p$-centre problem to be restricted to finite point set in $G$. However, unlike the absolute $p$-median problem, the optimal solution of an absolute $p$-centre problem, in general, is not contained in the vertex set of $G$. The above result will be presented in the later section of this chapter.

Here is a brief overview of this chapter: Section 1 presents the formulation of the $p$-centre problem. Section 2 reviews algorithms for solving various formulations of the $p$-centre problem and proves the complexity of the $p$-centre problem on $G$. Section 3 presents the formulation of the continuous $p$-centre problem. Section 4 deals with the continuous $p$-centre problem on $T$.

### 3.1 The $p$-Centre

For purposes of insight, a basic scenario is introduced here: If some urban planners want to locate a fixed number of emergency centres on a network, such as fire departments, hospitals or police stations, then the time takes to provide the service needed from the closest facility to a potential site of trouble (the demand site) must be minimized, to achieve efficient responses. This problem can be modelled as a $p$-centre problem. Suppose the network is modelled as a graph $G$, and the demand sites are modelled as vertices in $G$. The time an emergency centre requires to provide the service needed at a demand site is represented by the edge-length. The $p$-centre problem can be described as follows: Find the locations of emergency centres, such that the maximum time it takes the closest emergency centre to provide the service needed at some demand site is a minimum.

Two variations of the $p$-centre problem can be formulated from the above scenario:
The discrete $p$–centre formulation and the absolute $p$–centre formulation, depending on where the potential emergency centres are located. Both formulations are defined in the introduction (refer to page 26 for more details) of this chapter. However, there exist many natural extensions on the formulation of the centre problems. In this section a few references will be given:

Hakimi [31] solved the discrete 1–centre problem in $O(n^3)$ time by computing and then by examining the distance matrix of $G$. In the same reference, Hakimi [31] defined and solved the absolute weighted 1–centre problem. The references [34] [39] [48] [26] [37] solve various forms of absolute $p$–centre problems on $G$ or $T$.

Handler and Rozman [37] modified their algorithm for solving the absolute $p$–centre problem on $G$ into an algorithm for solving the continuous $p$–centre problem on $G$. The continuous $p$–centre problem is that $p$–centre problem, where both the potential demand sites and the emergency centres can be located at any point of $G$. There are many references which give polynomial algorithms for the continuous centre problem on $T$, and we will pay particular attentions on those solved algorithms.

Now, a brief introductions of some references that will not be investigated in this thesis:

Goldman [29] investigated and gave a polynomial algorithm for the absolute 1–centre problem on $T$. The objective function $g$ used is an extension of $f$, where a nonnegative addend is associated with each vertex ($g(x) = \max\{a(v) + d(v, x) | v \in V\}$). Halpin [30] improved the time complexity of the above algorithm [29]. Dearing and Francis [18] considered the weighted version of the same problem, where the objective function is further extended for the weighted case ($g(x) = \max\{a(v) + w(v)d(v, x) | v \in V\}$). Dearing and Francis [18] provided a proof of the existence of a 1–centre in $G$ and provided an algorithm for finding such a 1–centre in $T$ with respect to $g$.

Then there are those problems with distance constraints [60] [23], where there is an upper bound imposed on the distance between any two facilities.

Megiddo and Supowit [47] formulated and investigated the complexity of the $p$–centre problem in the Euclidean plane and the complexity of the $p$–centre problem in the rectilinear metrics. The Euclidean $p$–centre problem is formulated with the distance function of $\mathbb{R}^2$ and the rectilinear $p$–centre problem is formulated with the following distance function: $|x_i - z_i| + |y_i - t_i|$.

There are other formulations (nonlinear or biobjective) of $p$–centre problems, such as [59] [61], but we will not investigate them.
3.2 Algorithmic Approach to the \( p \)-Centre Problem

In order to facilitate the identification of various formulations of the \( p \)-centre problem with respect to \( f \), Handler’s [37] shorthand categorization scheme will be adopted in the rest of this chapter: \( \{V, E\} / \{V, E\} / \{\{G, T\}\} \), where the first cell describes the facility location set, which would be either the set \( V \) of all the vertices of the network or the set \( E \) of all the points of the network. The second cell describes the demand set, which would also be in either \( V \) or \( E \). The third cell indicates the number of facilities that we have to select from the facility location set. The last cell describes the underlying structure of the physical network set, which could be either a general graph \( G \) or a tree \( T \). For example, the discrete \( p \)-centre problem on general graphs will be denoted as \( V/V/p/G \) and the continuous 1-centre problem on a tree will be denoted as \( E/E/1/T \).

3.2.1 Algorithm on weighted \( V/V/1/G \)

Hakimi [31] demonstrated that the problem of finding the discrete 1-centre of \( G \) can be solved in \( O(n^2) \), if the distance matrix \( D \) of \( G \) is given. If the distance matrix of \( G \) is unknown, Floyd’s algorithm [55] can be used to find the distance matrix of \( G \). Let \( D_E = (d_{ij}^E) \) be the \( n \times n \) matrix of the edge-length in \( G \), where

\[
d_{ij}^E = \begin{cases} 
0 & \text{if } i = j \\
\infty & \text{otherwise} \\
l(\{v_i, v_j\}) & \text{if } \{v_i, v_j\} \in E 
\end{cases}
\]

**Floyd’s Algorithm**

1. Let \( D^1 = D_E \) and \( k = 0 \).
2. Increase the value of \( k \) by one. If \( k = n + 1 \), let \( D = D^n \) and terminate; otherwise go to Step 3.
3. Let \( D^{(k+1)} \) be the matrix having the elements: \( d_{ij}^{(k+1)} = \min\{d_{ij}^k, d_{ik}^k + d_{kj}^k\} \). Find the values of all the elements of this matrix and return to Step 2.

Since there are \( n^2 \) elements in each matrix, there are \( O(n^2) \) computations in each stage. There are \( n \) stages, so the distance matrix \( D \) can be found in \( O(n^3) \) time. Note,
\[ D = (d_{ij}) \text{ where} \]
\[ d_{ij} = \begin{cases} 
0 & \text{for } i = j \text{ and } i = 1, 2, \ldots, n \\
\quad \quad d(v_i, v_j) & \text{for } i \neq j \text{ and } i, j = 1, 2, \ldots, n
\end{cases} \]

Now that the distance matrix \( D \) of \( G \) is computed, we can apply the method of Hakimi [31].

**A SIMPLE DISCRETE 1–CENTER ALGORITHM**

1. Initialise the data, input \( D \).
2. Compute the set \( S = \{d_i|i = 1, 2, \ldots, n\} \), where \( d_i = \max\{d_{ji} \in D| j = 1, 2, \ldots, n\}, 1 \leq i \leq n. \)
3. If \( d_i = \min S \) then \( v_i \) is a 1–centre of \( G \).

This takes \( O(n^2) \) operations to find the 1–centre.

### 3.2.2 Algorithm for unweighted \( V/E/p/G \) (Absolute \( p \–Centre)\)

An algorithm for solving the discrete \( p \–centre \) problem will not be presented. However, we will present an algorithmic scheme for solving the absolute \( p \–centre \) problem on \( G \), which can be applied for solving the discrete \( p \–centre \) problem.

In this section, we will present the theorem stated in [49]. This theorem shown the existence of a set of finite number of potential demand locations in \( G \), which contains an optimal solution for the absolute \( p \–centre \) problem on \( G \). Most of the notation and presentation are from the following two references [37] [26].

The following definition is needed for the algorithm:

**Definition 1** \( (v_i, y, v_j) \) denotes a local centre at a point \( y \in G \) with respect to a pair of vertices \( v_i, v_j \) iff

\( (i) \quad d(v_i, y) = d(v_j, y) \)

\( (ii) \quad \{a \in A_y| \lim_{\theta \to 0} \left[ (d(z, x^y) - d(z, y)) \right] < 0, \text{ for } z = v_i, v_j = \phi, \]

30
where $A_y$ is the set of edges or edge-segments incident at $y$ and $x_y^a$ is a point distant $\theta$ from $y$ on edge $a \in A_y$.

Informally, condition (ii) of definition 1 says that there is no direction from $y$ in which the minimum distance to both $v_i$ and $v_j$ is decreasing. Let $C_G$ be the set of all the local centres in $G$ ($C_G = \{y \in G | (v_i, y, v_j) \text{ exist for some } v_i, v_j \in V\}$). Note: $(v_i, y, v_i)$ exit iff $y = v_i$. Furthermore, $(v_i, v_i, v_i)$ is called the null centre at $v_i$. By this definition every $v \in V$ is a null centre, hence every vertex in $G$ is an element of $C_G$.

**Theorem 3.2.1** Any optimal set $X_p$ of no more than $p$ centres of $G$ can be replaced by another optimal set of no more than $p$ points $X_p^*$, where $X_p^* \subseteq C_G$.

**Proof:** Let $X_p$ be an optimal set of no more than $p$ centres in $G$. Now we define a set:

$$P(X_p) = \{P_i(X_p) | i = 1, 2, \ldots, p\}$$

where

$$P_i(X_p) = \{v \in V, x_i \in X_p \text{ and } d(v, x_i) = d(v, X_p)\}.$$

So $P_i(X_p)$ is the set of all the vertices which are closest to $x_i$ in $X_p$. Ties for the closest centre may be broken arbitrarily.

Now, $X_p^*$ is derived from $X_p$ in the following way:

- For any $x_i \in X_p \cap C_G$ then let $x_i \in X_p^*$.
- Now, for each $P_i(X_p)$ where $x_i \notin C_G$, assign the elements to $X_p^*$ as follows:
  
  (i) If $|P_i(X_p)| = 1$ then it will not effect the optimal value by assigning $x_i^* = v_k$ where $v_k \in P_i(X_p)$. Let $x_i^* \in X_p^*$.

  (ii) If $|P_i(X_p)| \geq 2$, then we must examine the connected subgraph $G_i$ of $G$ induced from the vertices in $P_i(X_p)$ (for example, $E_i \subseteq E$ and $G_i = (E_i, P_i(X_p))$). Now, find the 1-centre $x_i^*$ of $G_i$ and let $r_i^*$ be the 1-radius of $G_i$ deduced from $x_i^*$. Let $x_i^* \in X_p^*$, since this will not affect the optimal value of $X_p$ ($r_i^* \leq \max_{y \in P_i(X_p)} d(x_i, y) \leq r_p$).

  We still need to verify that $x_i^*$ of $G_i$ is in $C_G$. Assume $x_i^* \notin C_G$ then by definition of $C_G$ we have $x_i^* \notin V$ and for any $v_i, v_k \in P_i(X_p)$. So let $v_j \in P_i(X_p)$ be the unique vertex such that $d(x_i^*, v_j) = \max\{d(v_i, x_i^*) | v_i \in P_i(X_p)\} = r_i^*$, but since $v_j$ is unique, one can reduce $r_i^*$ by shifting some distance $\theta > 0$ towards $v_j$ and reduce the value of $r_i^*$, which contradicts the fact that $x_i^*$ is a 1-centre of $G_i$. The proof of the theorem is now complete.

31
So the unweighted absolute $p$-centre problem on $G$ can be reformulated for finding $X^*_p$:

$$f(X^*_p) = \min_{X_p \subseteq G} \max_{v \in V} \{d(v, X_p)\}$$

is reformulated into

$$f(X^*_p) = \min_{X_p \subseteq G} \max_{v \in V} \{d(v, X_p)\}.$$ 

Hence the search for $X^*_p$ can be restricted to be in the finite point set $C_G$.

Minieka [49] developed a finite procedure for determining a $p$-centre on $G$. Handler and Rozman [37] were able to improve the algorithm by eliminating some of the columns of the matrix $F$ (defined in the next paragraph) during the execution of the algorithm.

It is possible to index the elements of the set $C_G$, because $|C_G|$ is finite. Let $F$ be the $|V| \times |C_G|$-matrix, where $F = (d_{ij})$ and $d_{ij}$ denotes the shortest distance from vertex $v_i$ to the $j$-th potential site in $C_G$. Let $H_p$ be the index set of a set of columns in $F$. Now, Handler and Rozman [37] reformulate the unweighted absolute $p$-centre problem as

$$f(X^*_p) = \min_{H_p \subseteq S} \min_{v_i \in V} \min_{j \in H_p} d_{ij}.$$ 

Here is Minieka's algorithm which was presented by Handler and Rozman [37] (the proof is omitted):

**MINIEKA'S ALGORITHM**


2. (i) Let $d = \max_{v_i \in V} \min_{j \in H_p} d_{ij}$.

   (ii) Update the matrix $B = (b_{ij})$, where

   $$b_{ij} = \begin{cases} 
   0 & \text{if } d_{ij} \geq d \quad i = 1, 2, \ldots, n \\
   1 & \text{otherwise} \quad j = 1, 2, \ldots, |C_G|
   \end{cases}$$

3. Solve the Covering Problem:

   $$h = \min e^t x$$

   $$Bx \geq e$$

   $$x_j \in \{0, 1\}, \quad j = 1, 2, \ldots, |C_G|$$

   where $e^t = (1, 1, \ldots, 1)$.

4. If $h > p$, stop with $r_p = d$ the value of an optimal solution. Otherwise, an improved solution has been obtained. Update $H_p$ and return to Step 2.
While the above algorithm solves $f(X^*_p) = \min_{H_p \subseteq S} \max_{v \in V} \min_{j \in H_p} d_{ij}$ (hence the absolute $p$-centre) in a finite number of steps, the procedure is prohibitively time-consuming. This is because as the number of vertices increases, the number of local centres has an upper bound of $n + (\binom{n}{2})|E|$, which can be extremely large. Before we present any more algorithms on the variations of the $p$-centre problem, we examine the following question: Does there exist a polynomial algorithm for solving the $p$-centre problem on a general network? In fact, Kariv and Hakimi [39] have shown that the general $p$-centre problems are NP-hard.

### 3.2.3 The General $p$-Centre Problem is NP-Hard

Kariv and Hakimi [39] used the result of Lemma 2.3.2 to show that the general $p$-centre problems are NP-hard.

**Theorem 3.2.2** Problems of finding a discrete $p$-centre and an absolute $p$-centre are NP-hard even in the case when the network is a vertex-unweighted planar graph of maximum degree 3, all whose edges are of length 1.

**Proof:** Let $G$ be a planar graph of maximum vertex degree 3, all of whose edges are of length 1 and all of whose vertices have weight 1.

We need only to show that the problem of Lemma 2.3.2 is polynomial-time reducible to the problem of finding a discrete $p$-centre on $G$ in the following way:

Suppose that we can find a discrete $p$-centre $V^*_p$ and the vertex $p$-radius $r_p(v)$ of $G$.  

**Claim 1:** There exists a dominating set $V^*$ such that $|V^*| \leq p$ iff $r_p(v) \leq 1$.

**Proof:** This claim is trivially true, since if $V^*$ is a dominating set for $G$ and $|V^*| \leq p$, then for any $v \in V$, $v$ is either in $V^*$ or $v$ is adjacent to a vertex in $V^*$. This implies that $d(v, V^*) = 0$ or $d(v, V^*) = 1$, since the edges in $G$ have length 1. Hence, let $V^*$ be the discrete $p$-centre and we have $r_p(v) \leq 1$. Conversely, let $V^*_p$ be the discrete $p$-centre on $G$ and $r_p(v) \leq 1$. By definition, this implies $f(V^*_p) = \max\{w(v_i)d(v_i, V^*_p)\forall v_i \in V\} \leq 1$. We know that $\forall v_i \in V$, $w(v_i) = 1$ and $d(v_i, V^*_p) \geq 0$, hence $d(v, V^*) = 0$ or $d(v, V^*) = 1$ for any $v \in V$. This implies that any $v$ in $V$ is either adjacent to an element in $V^*_p$ or in $V^*_p$. Hence $V^*_p$ is a dominating set for $G$ and $|V^*_p| \leq p$.

Using similar argument, the absolute $p$-centre problem can be shown to be NP-hard. Suppose $X^*_p$ is an absolute $p$-centre on $G$ and an absolute $p$-radius $r_p$. Clearly, if $r_p > 1$, 33
then there does not exist a dominating set for $G$ with cardinality $\leq p$. On the other hand, if $r_p \leq 1$, then we can replace each nonvertex point of $X^*_p$ by the closest vertex to it (ties are broken arbitrarily). Now we obtain a discrete $p$-centre of radius $\leq 1$, then by CLAIM 1, there exists a dominating set $V^*$ such that $|V^*| \leq p$.

Thus, the problem of Lemma 2.3.2 is polynomial time reducible to both the problem of finding an absolute $p$-centre and the problem of finding a discrete $p$-centre, which means that the general $p$-centre problems are NP-hard.

Kariv and Hakimi [39] then describe an algorithm of complexity $O\left(\frac{r^p n^{2p-1}}{(p-1)! \log n}\right)$ for finding an absolute $p$-centre in a vertex weighted graph and an algorithm of complexity $O\left(\frac{r^p n^{2p-1}}{(p-1)!}\right)$ for finding an absolute $p$-centre in a vertex unweighted graph. We will not present those algorithms.

### 3.3 The Continuous $p$-Centre Problem

In the previous section, the demand set was always restricted to be in $V$. Such a restriction may not be feasible in modelling some of the problems that arise in practice. For example, problems such as locating a fixed number of fire hydrants along a street network, or emergency breakdown services on a highway network. Those types of centres are expected to service every points in the network, hence the continuous formulation of the $p$-centre problem is needed.

Handler and Rozman [37] have modified their algorithm for solving the absolute $p$-centre problem on $G$ to solve the continuous $p$-centre problem on $G$. However, Theorem 3.2.2 has shown that both the discrete and the absolute $p$-centre problem on $G$ are NP-hard. Thus we expect that the continuous $p$-centre problem on $G$ should be as difficult as the general $p$-centre problems on $G$. Indeed, Megiddo and Tamir [46] have proved that the continuous $p$-centre problem on $G$ is NP-hard.

There exists many algorithms [45] [14] [15], which solve the continuous $p$-centre problem on a tree in polynomial time. Handler [35] has shown an elegant algorithm for solving the continuous 1-centre problem on $T$ in $O(n)$ time. Furthermore, Handler [36] extended the 1-centre algorithm to solve the continuous 2-centre problem on $T$ in $O(n)$ time. The continuous $p$-centre problem on $T$ is shown to be equivalent to the dual problem of locating $p + 1$ points on $T$ [57], so as to maximize the minimum distance between any pairs of those $p+1$ points. Frederickson and Johnson [20] used some of their
previous results [21, 22] and provided an $O(n \min\{n, p\} \log(\frac{2pn}{(\min\{n, p\})^2}))$ algorithm for solving the continuous $p$-centre problem on $T$.

In the next section, we will present an efficient algorithm of Megido and Tamir [46].

### 3.4 The Continuous $p$–Centre Problem on a Tree Network

In this section, we will investigate the dual for the continuous $p$-centre problem on $T$, the $(p+1)$–dispersion problem on $T$.

A dual problem usually provides a better insight to the primal problem: A dual problem plays a rôle in the determination of upper bounds for the optimal solution. A reasonable upper bound on the optimal value of all the feasible solutions is useful, because it can verify the claim that a feasible solution is optimal or it is within a specified relative tolerance.

After the subsection on the $(p+1)$–dispersion problem on $T$, we will investigate the $O(n \log^3 n)$ algorithm [46] for the continuous $p$-centre problem on $T$.

The formulation of the continuous $p$-centre problem on $T$ is as follows: Let $X_p \subseteq T$ and $|X_p| = p$, then the objective function $f : \mathcal{F}(T) \rightarrow \mathbb{R}$ is defined as follows:

$$f(X_p) = \max\{d(x, X_p) | \forall x \in T\}$$  \hspace{1cm} (3.1)

and the continuous $p$-centre problem on $T$ is then expressed as:

$$r_p = \min\{f(X_p) | |X_p| = p, X_p \subseteq T\}$$  \hspace{1cm} (3.2)

### 3.4.1 The Dual of the $p$–Centre Problem

Duality is generally very useful for obtaining necessary and sufficient conditions for optimality. In Chapter 8 of [50], a physical interpretation for the $(p+1)$–dispersion problem on $T$ was given. Chandrasek and Daughety [14] provided a polynomial algorithm for solving the $(p+1)$–dispersion problem on $T$. The algorithm runs in $O(n^2 \log(p+1) \log n)$.

$(p+1)$–Centre dispersion problems on $T$ are a class of optimal problems using the Max-Min criterion: Locate $(p+1)$ facilities on $T$ so that they are as far apart as possible.
For example, find \( y_1, y_2, \ldots, y_{p+1} \) on \( T \) so as to maximize the minimum distance between pairs of points. Here we give the formulation of the \((p+1)\)-dispersion problem on \( T \):

Let \( Y_{p+1} \subseteq T \) and \( Y_{p+1} = \{y_1, y_2, \ldots, y_{p+1}\} \), then the objective function \( g : \mathcal{F}(T) \to \mathbb{R} \) is defined as follows:

\[
g(Y_{p+1}) = \min\left\{ \frac{d(y_i, y_j)}{2} \mid i < j \leq (p+1) \right\}
\]

and the \((p+1)\)-centre dispersion problem on \( T \) is then expressed as:

\[
r_{p+1}^D = \max\{g(Y_{p+1}) \mid |Y_{p+1}| = p+1, Y_{p+1} \subseteq T\}
\]  

We will present some of results proved in [57]. Shier [57] has shown that the furthest distance a point need be from its nearest \( p \)-centre is precisely the same as half the minimum distance between \( p+1 \) points located as far apart as possible in the tree. To start with, let \( N_r(x) = \{y \in T \mid d(x, y) \leq r\} \) denotes the neighbourhood of radius \( r \) about any point \( x \) in \( T \).

**Lemma 3.4.1** Given \( T \), any \( r > 0 \) and any two collections of the points in \( T \), say \( A, B \). The following is true:

\[
\min\{|A| \mid \bigcup_{x \in A} N_r(x) = T\} = \max\{|B| \mid x, y \in B \text{ and } x \neq y \Rightarrow d(x, y) > 2r\}. \tag{3.5}
\]

**Proof:** Suppose \( B = \{b_1, b_2, \ldots, b_k\} \) is any collection of the points with \( d(b_i, b_j) > 2r \) for \( i \neq j \). Thus at most one element of \( B \) is in any neighbourhood of radius \( r \), this implies that at least \( |B| \) such neighbourhoods are required to cover \( T \). Hence the left-hand side of equation 3.5 \( \geq \) the right-hand side of equation 3.5.

The reverse inequality is proved, using induction on the number of vertices on \( T \): If \( T \) has one single vertex, the reverse inequality holds. This is because the single vertex is cover by one neighbourhood and the right-hand side of equation 3.5 can only have maximum of one element, therefore both sides of equation 3.5 equal 1.

Assume that the reverse inequality holds for all trees with fewer than \( k \) vertices. Let \( T \) be a tree with \( k \) vertices. Assume that the diameter of \( T \), denoted by \( d(T) \) where \( d(T) = \max\{d(x, y) \mid x, y \in T\} \), is greater than \( 2r \). We can make this assumption, because if \( T \) has diameter \( \leq 2r \), then again both sides of equation 3.5 equal 1, because \( T \) can be cover by one neighbourhood with radius less than \( r \). Let \( P(v, u) \) be the path in \( T \) between \( v \) and \( u \) with \( d(v, u) = d(T) \). Let \( x \) be a point on \( P(v, u) \) and \( d(u, x) = r \).
Note that $u \in N_r(x)$ and since $d(T) > 2r$, $v \not\in N_r(x)$. Suppose that $T'$ is the smallest subtree of $T$ induced by all points $y \in T \setminus N_r(x)$. That is $y \in T'$ means there exists $z_1, z_2 \in T \setminus N_r(x)$ with $y \in P(z_1, z_2)$. Since path in $T$ is unique, we have $y \in T'$ iff $\exists z_y \in T$ with $d(x, z_y) > r$ and $y \in P(z_y, v)$.

Now let $A$ be a minimal cardinality set in $T'$ such that neighbourhoods $N_{r'}(x)$ defined in $T'$ satisfy $\bigcup_{z \in A} N_{r'}(z) = T'$. Then $A \cup \{x\}$ provides a set of points such that neighbourhoods of radius $r$ about these points cover $T$. If $A^*$ is a minimal cardinality set of $T$ with $\bigcup_{z \in A^*} N_r(z) = T$, then

$$|A^*| \leq |A| + 1 \quad (3.6)$$

On the other hand, let $B$ be a maximal cardinality set of points mutually distant in $T'$ by more than $2r$. Suppose $y \in B$ and $d(x, y) \leq r$. Then let $B_y$ denote the subtree determined by all points $s \in T'$ where $y \in P(s, v)$; certainly $z_y \in B_y$. It is claimed that $y$ is the only member of $B \cap B_y$. For suppose $w$ were another such point; then $d(w, y) > 2r$ since $w, y \in B$, and

$$d(w, v) = d(w, y) + d(y, v) > 2r + d(x, v) - d(x, y) \geq r + d(x, v) = d(u, v).$$

However, this would contradict the fact that $u$ and $v$ are points of $T$ furthest apart. Accordingly, we can interchange $y$ and $z_y$ in $B$ without changing the fact that points in $B$ are mutually distant by more than $2r$. By repeatedly applying this argument, one can ensure that all points $y \in B$ satisfy $d(x, y) > r$. Thus, by the observation near the end of the previous paragraph, $d(u, y) = d(u, x) + d(x, y) > 2r$, and so $B \cup \{u\}$ is a set of points in $T$ mutually distant by more than $2r$. If $B^*$ is a maximal cardinality set in $T$ with this property then $|B^*| \geq |B| + 1 \geq |A| + 1 \geq |A^*|$ using equation 3.6 and the inductive hypothesis for $T' \subset T$. Thus equation 3.5 follows by induction. \[ \]

Shier [57] used Lemma 3.4.1 to prove the duality between the $p$-centre location problem and the $(p + 1)$-centre dispersion problem. Here is Shier’s Min-Max Theorem:

**Theorem 3.4.2** Given $T$, we have $r_p = r^p_{p+1}$ (see equation 3.2 and equation 3.4).

**Proof:** Let $X$ be a set of $p$ points in $T$. Then the $p$ neighbourhoods of radius $r = f(X)$ centred about these $p$ points cover $T$. Let $Y$ be any collection of $p + 1$ points in $T$. Then
there must be at least two distinct points \( v_1, v_2 \in Y \) that lie in the same neighbourhood, say \( N_r(u_i) \). Thus \( d(v_1, v_2) \leq d(v_1, u_i) + d(u_i, v_2) \leq 2r \), and this implies that

\[
g(Y) \leq \frac{d(v_1, v_2)}{2} \leq r = f(X).
\]

Accordingly, \( \max\{g(Y) ||Y|| = p + 1\} \leq \min\{f(X) ||X|| = p\} \) and so \( r_p \geq r_{p+1}^D \).

To prove the reverse inequality: Let \( Y^* \) be a set of \( p + 1 \) points that maximizes \( g(Y) \) and let \( r_{p+1}^D = g(Y^*) \). Since \( Y^* \) maximizes \( g \), there does not exist a set \( Y \subseteq T \) of \( p + 1 \) points with \( \frac{d(x, y)}{2} > r_{p+1}^D \), \( \forall x, y \in Y, x \neq y \). Hence

\[
\max\{|B| |x, y \in B \text{ and } x \neq y \Rightarrow d(x, y) > 2r_{p+1}^D\} = q \leq p.
\]

By Lemma 3.4.1, there is a set of \( q \) distinct points \( A = \{x_1, x_2, \ldots, x_q\} \) such that neighbourhoods of radius \( r_{p+1}^D \) cover \( T \). Accordingly, \( r_p \leq r_{p+1}^D \).

Now, we have \( r_p = r_{p+1}^D \).

### 3.4.2 The Algorithmic Approach to the Continuous \( p \)-Centre Problem on \( T \)

Shier's duality result is useful, since now we know that the continuous \( p \)-centre problem on \( T \) is solvable by a sequence of covering problems; by solving a finite sequence of covering problems and takes the limits:

**A Covering Problem**

\[
\begin{align*}
\text{minimize} & \quad |X| < n \\
\text{subject to} & \quad \min_{x \in X} \{d(v_i, X)\} \leq r, \quad i = 1, 2, \ldots, n, \\
& \quad X \subseteq T.
\end{align*}
\]

Furthermore, Theorem 3.4.2 implies that the optimal solution to the continuous \( p \)-centre problem is an element in the set

\[R' = \left\{ \frac{d(v_i, v_j)}{2} | i, j = 1, 2, \ldots, n \right\}.
\]

An efficient implementation of the approach to solve the continuous \( p \)-centre problem on \( T \) as a sequence of covering problems is presented by Megiddo and Tamir [46]. We will give a detailed investigation on this algorithm.

38
To start with, given $r > 0$, Chandrasekaran and Tamir [15] consider the problem of covering $T$ with a minimum number of $r$-neighbourhoods. This number is denoted by $M(r)$. It is clear that $M(r)$ is a monotone, nonincreasing, step function. Note that $M(r)$ is called the domination number, when $r = 1$.

If $T$ is a rooted tree, then a terminal vertex is a vertex with degree one. A maximal set of terminal vertices which are connected to the same vertex, say $s$, is called a cluster $C(s)$. Here is an algorithm [14] for finding $M(r)$ when $r$ is given:

**CLUSTER ELIMINATION ALGORITHM**

1. Choose a cluster $C(s)$.

2. Let $\{ (s, i) \in E| i \in C(s) \}$ be the set of edges connecting the terminal vertices to their father vertex $s$. For each $i \in C(s)$ let $d(s, i) = 2rk_i + b_i$ where $k_i$ is a nonnegative integer and $0 < b_i \leq 2r$. Set $d(s, i) \leftarrow b_i$ for $i \in C(s)$. (At this point $k_i$ facilities have already been located on edge $(s, i)$ with distance between adjacent facilities equal to $2r$. Also note that the trimmed edges have positive edge-lengths.)

3. Let

$$\alpha = \min_{i \in C(s)} \{ d(s, i)| d(s, i) > r \} = d(s, i_1^*)$$

and

$$\beta = \max_{i \in C(s)} \{ d(s, i)| d(s, i) \leq r \} = d(s, i_2^*)$$

In case of a tie $i_1^*$ (respectively $i_2^*$) can be chosen as the smallest index for which the minimum (respectively maximum) is attained. Also if $\alpha$ (respectively $\beta$) is defined on an empty set, it is taken to be $+\infty$ (respectively $-\infty$). (Note that at least one of $\alpha, \beta$ is finite.)

(i) If $(\alpha + \beta) > 2r$, then for each $i \in C(s)$ such that $d(s, i) > r$, locate a facility on $(s, i)$ at a distance $r$ from the terminal vertex $i$ (of the reduced cluster obtained in Step 2). Remove all the edges $(s, i)$ in $C(s)$ except $(s, i_2^*)$. If $s$ now becomes a terminal vertex, locate a facility at $s$ and terminate. Otherwise remove vertex $s$ from the path, $P(t, i_2^*)$, where $t$ is the father vertex of $s$ (Now we have the edge $(t, i_2^*)$) and go to Step 4.

(ii) If $(\alpha + \beta) \leq 2r$, then for each $i \neq i_1^*, i \in C(s)$ with $d(s, i) > r$, locate a facility on $(s, i)$ at a distance $r$ from the terminal vertex $i$. Remove all
the edges except \((s,i^*_t)\). If \(s\) is now a terminal vertex, locate a facility on \((s,i^*_t)\) at a distance \(r\) from \(i^*_t\) and terminate. Otherwise remove vertex \(s\) from the path, \(P(t,i^*_t)\), where \(t\) is the father vertex of \(s\) (Now we have the edge \((t,i^*_t)\)) and go to Step 4.

4. Choose a cluster of the remaining tree and return to Step 2.

The reasoning for this algorithm is clear. For \(i \in C(s)\), if \(d(s,i) > 2r\), then a facility must be located on the edge \((s,i)\) at a distance not greater than \(r\) from the terminal vertex \(i\). Since this facility may be located at a distance equal to \(r\) (from \(i\)) Step 2 follows. It is clear that after reductions in Step 2 are done, there must be a facility on each edge \((s,i)\) with \(d(s,i) > r\) and if \((\alpha + \beta) \leq 2r\), one of these serves the edges \((s,i)\) with \(d(s,i) \leq r\). If not, we need one additional facility and this has to serve the edge \((s,i^*_t)\) and hence the reductions in Step 3 are justified. It is clear that the above algorithm takes \(O(\max\{n,M(r)\})\) time, if the output is to be the \(M(r)\) facility locations. However, if in Step 2 only the number \(k\) of facilities are recorded, then this algorithm is \(O(n)\).

Chandrasekaran and Tamir [15] proved that the jump points of the \(M(r)\) are of the form \(d(;x;y)\) where \(l\) is integral and \(x, y\) are terminal vertices. In particular the \(p\)-radius \(r_p\) belongs to the set

\[
R = \left\{ \frac{d(x,y)}{2l} \mid 1 \leq l \leq p \text{ and } x,y \text{ are terminal vertices in } T \right\}.
\]

Here is the result of Chandrasekaran and Tamir [15]:

**Theorem 3.4.3** Let \(r_p\) be the solution to the continuous \(p\)-centre problem (see equation 3.2). Then \(r_p \in R\) where

\[
R = \left\{ \frac{d(x,y)}{2l} \mid 1 \leq l \leq p \text{ and } x,y \text{ are terminal vertices in } T \right\}.
\]

**Proof:** Let \(S = \{x_1,x_2,\ldots,x_p\}\) be the set of points on \(T\) at which the \(p\) optimal supply centres are located. Define \(Y = \{y \mid y \in T \text{ and } d(y,S) = r_p\}\), and let \(S'\) be the subset of supply points serving the members of \(Y\),

\[
S' = \{x \mid x \in S, d(x,y) = r_p \text{ for some } y \in Y\}.
\]

Claim that, without loss of generality, it can be assumed that each member of \(S'\) is the midpoint of a simple path of length \(2r_p\), connecting two points of \(Y\). Suppose that
$x \in S'$ does not have the above property. Then, this supply centre $x$ can be slightly perturbed to $x'$ such that the optimality is not affected; all points $y$ in $Y$ served by $x$ satisfy $d(x', y) < r_p$, and no additional points are added to $Y$. Therefore, $x$ can be omitted from $S'$ and all points $y$ in $Y$ served by $x$ can be omitted from $Y$. Note that the minimality of $r_p$ ensures that the set $S'$ remaining after this process is not empty.

To complete the proof the theorem, each member of $Y$ must be shown to be either a terminal vertex or a midpoint of simple path, which connecting two points of $S'$ and has length $2r_p$.

Let $y \in Y$. Then there exists $x_i \in S'$ with $d(y, x_i) = r_p$. If $y$ is not a terminal vertex there exists $z \neq y$, $z \in T$, and $y$ is on the simple path between $z$ and $x_i$. Considering only the subpath connecting $z$ and $y$, we observe that all points on this subpath but $y$ are not served by $x_i$, since they are at a distance greater than $r_p$ from $x_i$. So, let $x_k$ be the point in $S$, closest to $y$, and serving at least one point which is not $y$, on the above subpath. Clearly $d(y, x_k) = r_p$, since $y$ is in $Y$, and therefore $x_k \in S'$.

Moreover, since $d(x_i, u) \leq r_p$ for some $u \neq y$ on that subpath, $y$ is the only intersection point of the path connecting $y$ and $x_k$ and the path connecting $y$ and $x_i$. Hence $y$ is on the simple path between $x_i$ and $x_k$ with $d(y, x_i) = d(y, x_k) = r_p$.

Using the above properties satisfied by the members of $Y$ and $S'$, we start with $x \in S'$ and consider the path of length $2r_p$, which connects two points of $Y$ and has $x$ as its midpoint. If at least one of these endpoints is not a terminal vertex, the path can be extended by $2r_p$ such that the new path will still connect two members of $Y$. Continuing this process, the no-cycle property of a tree ensures that we find a simple path of the tree connecting two terminal vertices and having total length of $2lr_p$, $1 \leq l \leq p$. This completes the proof.

Theorem 3.4.3 implies that the optimal solution to the continuous $p$-centre problem is the smallest $r$ in $R$ for which the solution to respective covering problem (see page 38) is at most $p$. The cardinality of $R$ is $O(n^2p)$ for a particular $p$-centre problem on $T$. Thus, the question that needs to be answered: Is there an efficient search for value $r$ in the set $R$, without explicitly generating the whole set $R$? The answer to this question is "yes", [45] [20] both gave representations on efficient search in the set $R$.

It was shown that, for a fixed valued $r$, $M(r)$ can be solved using the cluster elimination algorithm. Megiddo [43] presented a general method of solving parametric combinatorial problems, which simulates the computation of $M(r)$ where $r$ belongs to a certain interval and is not just fixed at an unique value. He presented a general result
relating the time complexity of the ratio minimization problem to that of the linear
minimization problem, where both are subjected to the same constraints.

Megiddo and Tamir [46] presented an $O(n \log^3 n)$ algorithm for finding the continuous
$p$-centre. The basic idea of their approach is to obtain an interval containing $r_p$, then
the interval is repeatedly narrowed, such that each "reduced" interval contains $r_p$ and
$M(r) \leq p$. It then finds $r_p$ exactly.

Megiddo and Tamir [46] define the following step function with jump at $\lfloor \frac{d(x,y)}{2l} \rfloor$,
l $= 1, 2, \ldots$:

$$k_{x,y}(r) = \lfloor \frac{d(x,y)}{2r} \rfloor, \forall x, y \in V \text{ and } r > 0.$$  

Furthermore, they assume that $T$ is rooted and if $(i,j) \in E$ then tail($(i,j)$) is not an
element of the set of point of $(i,j)$.

The determination of the final interval is carried out in two phases:

**Phase 1 of Megiddo and Tamir's Algorithm Scheme**

**PHASE 1**: In this phase, an interval $(a_0, b_0]$ is found such that $r_p \in (a_0, b_0]$ and at
least the $k_{ij}(r)$'s corresponding to edges $(i,j)$ are constant over $(a_0, b_0]$ (for each edge
$(i,j)$ the function $k_{ij}(r)$ is constant for $r \in (a_0, b_0]$ ). To find this interval, the following
lemmas are needed:

**Lemma 3.4.4** In order for every point to be within a distance of $r$ from at least one
centre, at least $k_{ij}(r)$ centres must be located on the edge $(i,j)$.

**Proof**: Let $(i,j) \in E$ and let $k = \lfloor \frac{d(i,j)}{2r} \rfloor$. By definition of floor, we have

$$d(i,j) = 2kr + b, \text{ where } k \in \{0, 1, 2, \ldots \}, b \in [0, 2r).$$

It is obvious that $(i,j)$ can not be covered by less than $k$ neighbourhoods. Hence it
is sufficient to show there exists an neighbourhood cover for $(i,j)$, where $k$ centres of
the neighbourhoods are located on $(i,j)$.

If $b = 0$ then place $k$ centres $\{c_1, c_2, \ldots, c_k\}$ inductively in the following way: place
c_1 at distance $r$ from $i$, then place $c_{l+1}$ at distance $2r$ from $c_l$, where $l = 1, 2, \ldots, k - 1$.
Now $(i,j)$ is covered by $N_r(c_{l''})$, where $l'' = 1, 2, \ldots, k$.  

42
If \( b \neq 0 \) then place \( k + 1 \) centres \( \{c_1, c_2, \ldots, c_{k+1}\} \) inductively in the following way: place \( c_1 \) at \( i \ (i \notin (i, j)) \), then place \( c_{l+1} \) at distance \( 2r \) from \( c_l \), where \( l = 1, 2, \ldots, k \). Now \((i, j)\) is covered by \( N_r(c_{l'}) \), where \( l' = 1, 2, \ldots, k + 1 \) and \( c_1 \notin (i, j) \).

Hence this proves the lemma.

**Lemma 3.4.5** To satisfy the requirement in Lemma 3.4.4 with respect to points of the edge \((i, j)\) it is sufficient to allocate \( k_{ij}(r) + 1 \) centres to that edge.

**Proof:** It is obvious from the construction of the covers on \((i, j)\), that if \( b \neq 0 \) then there is a need to place \( k + 1 \) centres \( \{c_1, c_2, \ldots, c_{k+1}\} \) to cover \((i, j)\), hence it is sufficient to allocate \( k + 1 \) centres to that edge.

**Lemma 3.4.6** If \( m(r) = \sum_{(i,j) \in E} k_{ij}(r) \) then
\[
m(r) \leq M(r) \leq m(r) + (n - 1).
\]

**Proof:** For each \((i, j)\) in \( E \), let \( M((i, j)) \) be the number centre that is located on \((i, j)\) with respect to \( M(r) \) \( (\sum_{(i,j) \in E} M((i,j)) = M(r)) \). By Lemma 3.4.4, we have \( k_{ij}(r) \leq M((i, j)) \) for all the edges in \( E \), this implies \( m(r) = \sum_{(i,j) \in E} k_{ij}(r) \leq \sum_{(i,j) \in E} M((i,j)) = M(r) \). By Lemma 3.4.5, we know it is sufficient to allocate \( k_{ij}(r) + 1 \) centres on each edge \((i, j) \in E \). Now, there are \((n - 1)\) edges in \( T \), so \( M(r) \leq \sum_{(i,j) \in E} (k_{ij}(r) + 1) = m(r) + (n - 1) \). So the lemma is trivially true.

With equation 3.7 and the definition of \( M(r) \), the following bounds can be established:
\[
p - 2(n - 1) < m(r_p) \leq p \tag{3.8}
\]
\[
p - (n - 1) < m(r') \leq p \quad \text{where} \quad r' = \sum_{(i,j) \in E} \frac{d(i,j)}{2p} \tag{3.9}
\]

Now, using the cluster elimination algorithm to compute \( M(r') \) and determine whether \( r' \geq r_p \) or \( r' < r_p \).

(i) If \( r' \geq r_p \) then the procedure continuously decrease \( r \), starting from \( r = r' \) until \( m(r) = p + 1 \). From equation 3.9, this procedure will approach at most \( p + 1 - m(r') < n \) jump points of the function \( m \). All these jumps are at points
of the form \( \frac{d(i,j)}{2(k_{ij}(r') + l)} \) where \( 1 \leq l \leq p + 1 - m(r') \). As matter of fact, these jumps occur at the \( p + 1 - m(r') \) largest elements of the set

\[
R_0 = \{ \frac{d(i,j)}{2(k_{ij}(r') + l)} | (i,j) \in E, l = 1, 2, \ldots, p + 1 - m(r') \}.
\]

Since \( R_0 \) is naturally partitioned into \( n - 1 \) sorted subsets, corresponding to the \( n - 1 \) edges in \( T \), these jumps can be found and sorted in \( O(n + \min\{n, p\} \log n) \) time using a standard priority queue [1].

(ii) If \( r' < r_p \) then the procedure continuously increase \( r \), starting from \( r = r' \) until \( m(r) = \max\{0, p - 2(n - 1)\} \). From equation 3.8, this procedure will approach at most \( m(r') - \max\{0, p - 2(n - 1)\} \leq \min\{2n, p\} \) jump points of the function \( m \). Using the scheme of Step (i), similarly all these jumps are found in \( O(n + (m(r') - \max\{0, p - 2(n - 1)\}) \log n) \) time.

Now, all the jumps are found and sorted, the procedure continuous to search for consecutive jump points \( a_0, b_0 \) such that \( r_p \in (a_0, b_0] \). The time used by PHASE 1 is \( O(\min\{n, p\} \log n + n \log \min\{n, p\}) \).

**Phase 2 of Megiddo and Tamir’s Algorithm Scheme**

At the end of PHASE 1, the interval \( (a_0, b_0] \) is obtained. Before we continue to examine Phase 2 of the algorithm scheme, we will present the algorithm on decomposition of trees from [45].

In [45], a procedure was introduced to decompose a tree \( T \) into three or fewer subtrees. The subtrees have the following properties:

- Precisely one vertex is shared by all the subtrees.
- Each subtree has no more than \( \frac{3}{2} + 1 \) vertices.

Let \( N(i) \) be the set of vertices which are the neighbours of the vertex \( i \). Furthermore, \( K(i, j) \) is the number of vertices in the subtree \( T_i \), where edge \( \{i, j\} \) is removed (see page 23 for more details). All the values for \( K(i, j) \)s can be recursively computed in \( O(n) \). And we will need the following observations to prove the correctness of this procedure:

1. If \( i \) is a terminal vertex where \( N(i) = \{j\} \) then \( K(i, j) = 1 \).
2. \( K(i, j) + K(j, i) = n \) for all pairs of neighbours.
For all $j$, $\sum_{i \in \mathcal{N}(j)} K(i, j) = n - 1$.

(4) If $j, k \in \mathcal{N}(i)$ ($j \neq k$) then $K(j, i) < K(i, k)$.

Assume that all the $K(i, j)$s are known, the following process can be used to find a vertex $x$ such that $K(i, x) \leq n/2$, $\forall i \in \mathcal{N}(x)$. This vertex $x$ will be refer as the centroid of $T$.

CENTROID SEARCH ALGORITHM

0. $x \leftarrow 1$

1. if $K(i, x) \leq \frac{n}{2}, \forall i \in \mathcal{N}(x)$ then Stop. else (there is precisely one $i \in \mathcal{N}(x)$ such that $K(i, x) > \frac{n}{2}$) $x \leftarrow i$ go to Step 1.

Note: Centroid Search Algorithm is a special case of Goldman’s 1–Median Algorithm on $T$, where the vertices in $T$ all have weight of one.

CLAIM 1: $x$ is a centroid.

Proof: The procedure generates a path $1 = x_1, 2 = x_2, \ldots, k = x_k$ such that $K(x_{j+1}, x_j) > \frac{n}{2}$ where $j = 1, 2, \ldots, k - 1$ and $x_k = x$. By observations (2) and (4), the function $m(x_j) \equiv \max_{i \in \mathcal{N}(x_j)} K(i, x_j)$ is monotone decreasing along that path with finite vertices. Hence an $x_k = x$ is reached for which $m(x) \leq \frac{n}{2}$, so $x$ is the centroid for $T$.

CLAIM 2: Set $\mathcal{N}(x)$ can be partitioned into two subsets $\mathcal{N}_1, \mathcal{N}_2$ such that $\sum_{i \in \mathcal{N}_i} K(i, x) \leq \frac{2n}{3}$.

Proof: Assume $\mathcal{N}(x) = \{v_1, v_2, \ldots, v_q\}$. By observation (3) there is $v_s \in \mathcal{N}(x)$ such that

$$\sum_{i=1}^{s-1} K(v_i, x) \leq \frac{n-1}{2} \quad \text{and} \quad \sum_{i=s+1}^{q} K(v_i, x) \leq \frac{n-1}{2}.$$ 

Now the $\mathcal{N}(x)$ is partitioned into 3 subsets: $\mathcal{N}_1' = \{v_i \in \mathcal{N}(x) | 1 \leq i \leq s - 1\}$, $\mathcal{N}_2' = \{v_s\}$ and $\mathcal{N}_3' = \{v_i \in \mathcal{N}(x) | s + 1 \leq i \leq q\}$. Now, let $m = \max\{\sum_{i=1}^{s-1} K(v_i, x), \sum_{i=s+1}^{q} K(v_i, x), K(s, x)\}$. We have $\frac{2n}{3} \leq m \leq \frac{n-1}{2}$, since observation (3). So let $\mathcal{N}_1$ be the set $\mathcal{N}_1', i = 1, 2, 3$ where $\sum_{v \in \mathcal{N}_1} K(v, x) = m$ and $\mathcal{N}_2$ be the union of the other two sets. Now $\sum_{i \in \mathcal{N}_i} K(i, x) \leq \frac{2n}{3}, i = 1, 2$.

Finally, the partition of $\mathcal{N}(x)$ induces a decomposition of $T$ into two subtrees $T_i, i = 1, 2$ where each subtree has at most $2n/3$ of the vertices of $T$; let $T_i$ be the subtree consisting
of $x$ and all the vertices accessible from $x$ via a member of $N_x$. Obviously, $x$ is the only vertex of $T$ that belongs to more than one such subtree, and in each subtree there are not more than $\frac{2m^2}{3}$ vertices. This decomposition runs in time $O(n)$. Furthermore, this decomposition may proceed into the subtrees and their components and so on, and that the whole hierarchy will be called a total centroid decomposition. The total centroid decomposition takes $O(\log n)$ phases.

**PHASE 2:** To start with, the algorithm is given a tree $T$ and an interval $(a, b]$ such that $a < r_p \leq b$ and function $k_{ij}(r)$ (for $i, j \in E$) are known to be constant over $(a, b]$. The aim of **PHASE 2** is to find $(a', b'] \subseteq (a, b]$ such that for all $x, y \in V$ the function $k_{xy}(r)$, will be constant on $(a', b']$. The algorithm goes as follows: Given a tree $T$, we applied the centroid decomposition on $T$ and partition $T$ into two subtrees $T_1, T_2$ and let $c$ be the centroid of $T$. Then algorithm applies the routine recursively to the trees $T_1$ and $T_2$. Thus the algorithm obtains an interval $(a_0, b_0]$ ($a_0 < r_p \leq b_0$) such that $k_{xy}(r)$ is constant on $(a_0, b_0]$. Hence, the constant value over $(a_0, b_0]$ of $k_{xy}(r)$ for $x \in V_1$ and of $k_{xy}(r)$ for $y \in V_2$ is also assumed to be known. Moreover, $k_{xc}(r) + k_{cy}(r) \leq k_{xy}(r) + k_{cy}(r) + 1$. Thus, the function $k_{xy}(r)$ ($x \in T_1, y \in T_2$) may have at most one jump in the interval $(a_0, b_0]$, namely, when it jumps from $k_{xc}(r) + k_{cy}(r) + 1$ to $k_{xc}(r) + k_{cy}(r)$. This occurs at the value

$$d(x, y) = 2(k_{xc}(r) + k_{cy}(r) + 1).$$

Denote $a_x = \frac{1}{2}d(x, c)$, $b_y = \frac{1}{2}d(c, y)$, $c_x = k_{xc}(r)$ and $d_y = k_{cy}(r) + 1$. Now search for $r_p$ within the set

$$R' = \{a_x + b_y | x \in V_1, y \in V_2\}.$$

In other words, the algorithm searches for $\alpha', \beta' \in R' \cup \{a_0, b_0\}$ such that $a_0 \leq \alpha' < r_p \leq \beta' \leq b_0$ and $(\alpha', \beta') \cap R' = \phi$. Then Megiddo and Tamir [46] give a search scheme for $r_p$, which performed in $O(n \log^2 n)$ time. Here is the searching algorithm scheme:

**The Searching Algorithm Scheme on** \{\(\frac{a_i + b_j}{c_i + d_j}\)\}

Let $S = \{\frac{a_i + b_j}{c_i + d_j} | 1 \leq i, j \leq n\}$. The aim of this searching algorithm scheme is to find $r_p$ in $S$. Notice that the elements in $S$ can be described as the solution of $(c_i x - a_i) + (d_j x - b_j) = 0$. The search will be conducted in two stages:
Stage 1. In this stage, the algorithm searches for $r_p$ among the points of intersection of lines $y = c_i x - a_i$ with each other. This algorithm scheme simulates Preparata’s [51] parallel sorting scheme in a serial manner (this method was introduced in [44]). Preparata’s parallel sorting scheme employs $n \log n$ processors during $O(\log n)$ steps. Now, the serial algorithm trying to sort the set $\{c_i x - a_i | i = 1, 2, \ldots, n\}$, where $x$ is not known. Whenever a processor in Preparata’s algorithm has to compare some $c_i x - a_i$ with $c_j x - a_j$, $i \neq j$, this algorithm will compute the critical value $x_{ij} = \frac{a_i - a_j}{c_i - c_j}$. After execution of a single step in Preparata’s scheme will create the production of $n \log n$ points of intersection of lines $y = c_i x - a_i$ with each other. The crucial point is that these values are produced independently of each other since the processors work “in parallel”. Thus, the $k$-th processor does not have to know the outcome of the comparison for which processor $k - 1$ is responsible. Therefore, the scheme starts the testing only when all the $n \log n$ processors have produced critical values. Given these $n \log n$ critical values and an interval $(s_0, t_0]$ which contains $r_p$, the algorithm can in $O(n \log n)$ time narrow down the interval so that it will still contain $r_p$ but no intersection point in its interior. This requires the finding of medians in the sets of cardinalities $n \log n, \frac{1}{2}n \log n, \frac{1}{4}n \log n, \ldots$ plus $O(\log n)$ evaluations of $M(r)$. Then the algorithm proceeds to the next step in Preparata’s scheme.

The time used by each step is $O(n \log n)$, hence the entire stage 1 takes $O(n \log^2 n)$ time.

Stage 2. To start with, the algorithm assumes, without loss of generality, that for $x \in [s_1, t_1], c_i x - a_i \leq c_{i+1} - a_{i+1}, i = 1, 2, \ldots, n - 1$. Let $j$ ($1 \leq j \leq n$) be fixed and consider the set $S_j$ of $n$ lines $S_j = \{y = c_i x - a_i + d_j x - b_j | i = 1, 2, \ldots, n\}$. Since $S_j$ is “sorted” over $[s_1, t_1]$, the algorithm can find in $O(\log n)$ evaluations of $M(r)$ a subinterval $[s_j^*, t_j^*]$ such that $s_j^* < r_p \leq t_j^*$, and such that no member of $S_j$ intersects the $x$-axis in the interior of this interval. However, the algorithm scheme works on the $S_j$’s in parallel. Specifically, there will be $O(\log n)$ steps. During a typical step, the median of the remainder of every $S_j$ is selected (in constant time) and its intersection points with the $x$-axis is computed. The set of these $n$ points is then searched for $r_p$ and the interval is updated accordingly. This enables the algorithm to discard a half from each $S_j$. A step in stage 2 takes $O(n \log n)$ time, hence stage 2 is carried out in $O(n \log^2 n)$ time.

At the end of stage 2, the algorithm will obtain values $\{s_j^*, t_j^* | j = 1, 2, \ldots, n\}$. Let $s = \max_{1 \leq j \leq n} s_j^*$ and $t = \max_{1 \leq j \leq n} t_j^*$, thus $s < r_p \leq t$ and $S \cap (s, t) = \phi$. 47
At the end of this algorithm, we obtained an interval $[a, b]$ such that all the $k_{xy}$'s are constant over $(a, b)$ and $r_p \in (a, b)$. Thus, we have $r_p = b$, since at least one of the function $k_{xy}$ must jump at $r_p$. Hence the continuous $p$–centre problem is solvable by Megiddo and Tamir's algorithm [46] in $O(n\log^3 n)$ time, since PHASE 2 is organized as $O(\log n)$ stages determined by the total centroid decomposition on $T$. 
Chapter 4

Continuous Min-Max Tree Partition

In the previous two chapters, \( p \)-median and \( p \)-centre problems were investigated and some algorithms for solving those problems were explored. Two distinctive observations can be made:

(i) In both cases, the location problems were shown to be \( \text{NP-hard} \) for general networks. However, polynomial-time algorithms are constructed for those location problems on a tree. The reason for this is the convexity of most location problems on trees.

(ii) In general, most location problems in tree networks can be solved by using some sort of subtree covering approach [3], such as tree decomposition or tree partition.

In Dearing, Francis and Lowe [19], there is an in-depth discussion on the convexity of location problems on tree networks. In this chapter and the following one, we will investigate some algorithmic approaches to some optimization problems on tree networks. We will apply shifting algorithms [5] [12] [13] to solve the \( p \)-partition problem on tree.

There are some relevant references on variants of these problems, such as Lucertini, Perl and Simeone [42] who examine the problem involving the minimization of the difference between the largest and the smallest weight of a component; De Simone, Lucertini, Pallottino and Simeone [56] consider the sum of the absolute deviations of the component weights from their mean \( \frac{W}{p} \), where \( W \) is the total weight of the tree; Agasi, Becker and Perl [2], Becker and Schach [9], Perl and Snir [54] all investigated some sort of constrained partition problems.

This chapter is organized in the following way: In the next section, a practical
problem will be presented to motivate the investigation of the \( p \)-partition on networks. The discrete optimal (Min-Max or Max-Min) \( p \)-partition problems on \( G \) are shown to be \( \text{NP-hard} \), even if \( G \) is a grid graph of 3 rows and \( p = 2 \). Section 2 provides an introduction for the continuous Min-Max \( p \)-partition on \( T \). Section 3 presents a pseudo-polynomial algorithm for the problem with rational edge-lengths. Section 4 makes the necessary improvements to give a polynomial algorithm. Section 5 proves the correctness of the polynomial algorithm. Section 6 considers the complexity.

4.1 \( p \)-Partition Problems on \( G \)

A central coordinator wants to allot the maintenance of highway (telephone or pipeline) networks among \( p \) service units with equal work-capacities. If one assumes that the workload for the maintenance of a subgraph is proportional to the length of the subgraph, then an obvious way to solve this problem is to cut \( G \) into \( p \) connected subgraphs whose lengths are equal. Partitioning a network \( G \) into \( p \) equal components is not always possible; the tree network in Figure 4.1 is an example which illustrates this. Thus, if equal partition is not possible, we need to consider other criteria. In this chapter we will consider the Min-Max criterion: Find a \( p \)-partition \( P \) of \( G \) such that the maximum component in \( P \) is less or equal to the maximum component of \( P', \forall P' \in \Pi(G,p) \). In the next chapter, we will consider the Max-Min criterion: Find a \( p \)-partition \( P \) of \( G \) such that the minimum component in \( P \) is greater or equal to the minimum component of \( P', \forall P' \in \Pi(G,p) \). Figure 4.1 provides an example that illustrates that the \( p \)-partition obtained from the above two criteria can be different.

![Figure 4.1](image-url)

Figure 4.1: Example of 4-partitions on a tree with respect to the Max-Min criterion and the Min-Max criterion.
A theorem from [6] will now be presented. This theorem proves that the vertex restricted Max-Min $p$-partition problem on $G$ is NP-hard, even if $G$ is a grid graph of 3 rows and $p = 2$. With a simple modification, this theorem can also show that the vertex restricted Min-Max $p$-partition problem on $G$ is NP-hard.

The Subset Sum problem [41] is defined as follows: Given a finite set $S$, integer weight function $w : S \rightarrow N$, and target integer $B$, does there exist a subset $S' \subset S$ such that $\sum_{a \in S'} w(a) = B$? This problem is shown to be NP-complete in [41].

Here is some of the notation used in [6]: Let $G$ be $n \times m$ undirected grid graph with $V = \{(i,j)\}$ where $(i,j)$ is the vertex in the $i$-th row and $j$-th column, $1 \leq i \leq n$ and $1 \leq j \leq m$. An edge $e \in E$ iff it joins $(i,j)$ to $(k,l)$ for some $i, j, k, l$ where $|i - k| + |j - l| = 1$. Figure 4.2 gives an example of a grid graph.

![Figure 4.2: Example of a 2-partition on a 3 x 5 grid graph.](image)

Theorem 4.1.1 [6] is used to show NP-hardness of Max-Min partition problem on $G$, however we extend the proof to prove the NP-hardness of Min-Max partition problem on $G$. Note: the $P_2$ is the decision problem correspond to the Max-Min partition problem on $G$ and respectively the $P'_2$ is the decision problem correspond to the Min-Max partition problem on $G$.

**Theorem 4.1.1** Let $P_2$ (respectively $P'_2$) be the problem of finding whether, given $G$ and $K$, there exists a 2-partition of $G$ satisfying $w(C_i) \geq K$ (respectively $w(C_i) \leq K$), $(i = 1, 2)$. The problem $P_2$ (respectively $P'_2$) for $3 \times M$ graphs is NP-complete.

**Proof:** Clearly $P_2$ (respectively $P'_2$) is in NP. A polynomial time reduction from Subset Sum is exhibited. Let $s_1, \ldots, s_M$ be an instance of Subset Sum. Construct (in polynomial time) an $3 \times M$ grid graph with $w((2,j)) = s_j (j = 1, \ldots, M)$ and $w((i,j)) = 0$ otherwise (see Figure 4.2).
CLAIM 1: For any nonempty proper subset $S$ of the index set $\{1, 2, \ldots, M\}$ with complement $S'$, there is a 2-partition of $G$ with $\{(2, j)\}_{j \in S}$ in one component and $\{(2, j)\}_{j \in S'}$ in the other component.

Proof: Let $U$ be the set of vertices in the first row of $G$ and let $L$ be the set of vertices in the third row. The required 2-partition is then $\{(2, j) \in S \cup U, (2, j) \in S' \cup L\}$.

From CLAIM 1, we see that the instance of Subset Sum has a solution iff there is a solution of the instance of $P_2$ (respectively $P_2'$ for this network with $K = (1/2)\sum_{j=1}^{M} s_j$. Hence the reduction is proved.

Theorem 4.1.1 shows that the optimal $p$-partition problems are NP-hard and we will focus our attention on finding polynomial algorithms that solve the continuous optimal $p$-partition problems on $T$ in rest of the thesis.

4.2 Continuous Min-Max $p$-Partition on $T$

Becker, Simeone and Chiang [7] [8] have investigated the continuous Max-Min partition problem on tree with rational valued edge-lengths. A pseudo-polynomial time algorithm is implemented, then a polynomial-time algorithm is derived from it. In this chapter, we present a similar approach to the problems of continuous Min-Max partition on a tree.

We can formulate the objective function $f$ for Min-Max $p$-partitions using the definition of a $p$-partition of $T$ in Chapter 1. Let $f : \Pi(T, p) \to \mathbb{R}$ be the objective function, where $f(P) = \max\{w(C_i) | C_i \in P\}$. Now, the Min-Max $p$-partition problem on $T$ is finding $P^* \in \Pi(T, p)$ such that $f(P^*) = \min\{f(P) | P \in \Pi(T, p)\}$, and we call such a partition an optimal $p$-partition.

Becker, Perl and Schach [10] [11] presented an algorithm for discrete Min-Max $p$-partition on tree, where the weight function $w(C_i)$ is the sum of all the vertex weight in the component $C_i$. The discrete partition formulation, however, usually cannot achieve an efficient way of allocating equal work loads or resources, since edge-splitting is forbidden among different units; this may result in poor work load balance. We thus survey the algorithm for Continuous Min-Max Tree Partition. In continuous Min-Max $p$-partition on $T$, the weight function, denoted by $l(C_i)$ instead of $w(C_i)$, is the sum of the edge-lengths of all the edge segments of the component $C_i$.

A labelled $p$-partition is a $p$-partition whose cuts are labelled 1, 2, ..., $p-1$. It will
often be convenient to think of a $p$-partition as being labelled. We will assume that for
every edge $e$ the length $l(e)$ of $e$ is a positive rational number. Any point $x$ of a rooted
tree $T$ is uniquely determined by an ordered pair $(\text{edge}(x), \text{dist}(x))$, where $\text{edge}(x)$ is
the edge containing $x$ and $\text{dist}(x)$ is the distance of $x$ from the head($x$). We represent
the point $x$ by this pair. Hence a labelled $p$-partition whose cuts are $c_1, \ldots, c_{p-1}$ (where
$i$ is the label of $c_i$) can be thought of as a $2 \times (p - 1)$ matrix

$$P = \begin{bmatrix} e_1 & e_2 & \cdots & e_{p-1} \\ d_1 & d_2 & \cdots & d_{p-1} \end{bmatrix},$$

where $e_k = \text{edge}(c_k)$ and $d_k = \text{dist}(c_k)$.

Now, we can show the existence of an optimal $p$-partition:

**Theorem 4.2.1** An optimal $p$-partition always exists.

**Proof:** Let $e_{\text{max}}$ be the maximum length of a leaf-edge, and let $\theta = e_{\text{max}}/p$. Barring
the trivial case $p = 1$, there is a $p$-partition $P_0$ such that $f(P_0) \geq \theta$ (put all $p - 1$ cuts
on the leaf edge of maximum length). Hence we may restrict ourselves, only to those
partitions for which the distance of any two cuts along a same edge is at least $\theta$, since
at least one of these partitions is optimal.

Two (labelled) $p$-partitions $P$ and $P'$ will be said to be similar if

(i) $e_i = e'_i$ for all $i = 1, \ldots, p - 1$;

(ii) if $e_i = e_j$ then $d_i < d_j$ if and only if $d_i' < d_j'$.

Clearly, similarity is an equivalence relation, and there are a finite number of equiv­
calance classes. Thus, in order to prove the theorem, it will be enough to establish the
following

**CLAIM:** In each equivalence class there exists a $p$-partition which is optimal over all
partitions in the class.

So, consider any given equivalence class, and choose any $p$-partition $P$ in the class.
Let $I$ be the set of those $1 \leq i \leq p - 1$ such that there is some $1 \leq j \leq p - 1$, for
which $e_i = e_j$ and $d_j < d_i$. Among such indexes $j$, the one for which $d_j$ is largest will
be denoted next($i$). That is, for $i \in I$, next($i$) is the label of the cut following the $i$-th
cut along the edge $e_i$ in the down-direction. Notice that, in view of (i) and (ii), the set
$I$ and the function next($i$) are independent of the chosen partition in the equivalence
class.
There is a one-to-one correspondence between the partitions 

\[ P = \begin{bmatrix} e_1 & \ldots & e_{p-1} \\ d_1 & \ldots & d_{p-1} \end{bmatrix} \]

in the class and the vectors \( d = (d_1, \ldots, d_{p-1}) \) belonging to the set

\[ \Omega \equiv \{ d \in \mathbb{R}^{p-1} : 0 \leq d_i \leq l(e_i), i = 1, \ldots, p - 1; d_i - d_{\text{next}(i)} \geq 0, i \in I \}. \]

This follows from the fact that the vector \((e_1, \ldots, e_{p-1})\) is the same for all partitions in the class, by (i) above. In view of the above remark, the smallest length of a component of a partition in the class is a function \( \Phi(d) \) of the vector \( d \) corresponding to the partition. We will show that \( \Phi(d) \) is continuous in \( \Omega \).

First of all, let us show that the length of each (directed) segment \( s \) of the partition corresponding to \( d \) is a linear function of \( d \). We distinguish four possible cases:

1. The tail of \( s \) is a cut, say the \( i \)-th one, and the head of \( s \) is \( \text{head}(e_i) \). Then the length of \( s \) is given simply by \( d_i \).

2. The tail of \( s \) is a cut \( i \) and head of \( s \) is the cut \( \text{next}(i) \). Then the length of \( s \) is \( d_i - d_{\text{next}(i)} \).

3. The head of \( s \) is a cut \( i \) and the tail of \( s \) is \( \text{tail}(e_i) \). Then the length of \( s \) is \( l(e_i) - d_i \).

4. Segment \( s \) coincides with some edge \( e_i \). Then the length of \( s \) is \( l(e_i) \).

Let \( \varphi_0(d) \) be the length of the top component, and \( \varphi_i(d) \) \((i = 1, \ldots, p)\) the length of the down-component of cut \( i \). Then each \( \varphi_i(d) \) \((i = 0, \ldots, p - 1)\) is a linear function of \( d \), being the sum of the lengths of a finite number of segments.

It follows that \( \Phi(d) = \max\{\varphi_0(d), \varphi_1(d), \ldots, \varphi_{p-1}(d)\} \) is a piecewise-linear concave function, and thus it is continuous in \( \Omega \).

But then the function \( \Phi(d) \), being continuous on the compact set \( \Omega \), has a minimum in \( \Omega \) by Weierstrass' Theorem. This proves the claim and the theorem.

4.3 A Nonpolynomial Algorithm

For the remainder of this chapter we will assume that all the trees \( T \) have the following structure: The lengths of all edges are rational and the weights of all the vertices are
zero, unless specified otherwise.

The approach to the algorithm is similar to the approaches taken in the previous chapter: Under the assumption that all the edge-lengths are rational, we identified a set of potential sites in $T$, where we can place the $p - 1$ cuts of an optimal $p$-partition. Thus, the continuous Min-Max tree partition problems can be reduced to the discrete Min-Max tree partition problems. The latter problem is then solved by the algorithm of Becker, Perl and Schach [10]. In spite of the fact that this algorithm is polynomial in $n$ and $p$ [11], the resulting algorithm for continuous Min-Max partitioning is not polynomial, since the reduction itself is not polynomial.

Given a dissected tree $T$, a cut $c$ is said to be an endcut if $c$ is an endpoint of some edge, and a midcut if $c$ belongs to the interior of some edge. Let $P \in \Pi(T, p)$ and let $q$ be the number of endcuts of $P$ plus one. One can associate with $P$ the $q$-partition $P_E$ whose cuts are the endcuts of $P$. The set of midcuts of $P$ will be denoted by $M(P)$.

An optimal $p$-partition is said to be tuned if, for each component $C$ of $P_E$, the $m_C$ midcuts of $P$ that lie within $C$ define an optimal $(m_C + 1)$-partition $M_C$ of $C$.

**Theorem 4.3.1** Among the optimal $p$-partitions of $T$ there is always a tuned one. Furthermore, a tuned $p$-partition $P$ of $T$ has the following property: each component $C$ of $P_E$ is subdivided into components of equal length by the midcuts of $P$ that lie within $C$.

**Proof:** Let $P^*$ be any arbitrary optimal $p$-partition of $T$, the existence of such optimal $p$-partition of $T$ is guaranteed by Theorem 4.2.1. If $P^*$ is tuned, then we have found a tuned optimal $p$-partition of $T$. Suppose $P^*$ is not a tuned partition, then we can obtain a tuned $p$-partition of $T$ as follows: For each component $C$ of $P_E$, let $m_C$ be the number of midcuts of $P^*$ that lie in $C$, and let $M_C^*$ be the $(m_C + 1)$-partition of $C$ defined by these midcuts. For all $C$ such that $m_C \geq 1$, replace the cuts of $M_C^*$ by the cuts of an optimal $(m_C + 1)$-partition $M_C'$ of $C$ (some of the cuts of $M_C'$ may turn out to be endcuts). In this way one obtains a new $p$-partition $P'$ of $T$. The partition $P'$ is still optimal, since for every $C$ the length of the largest component of $M_C'$ is at least as small as the length of the largest component of $M_C^*$. One always has $|M(P')| \leq |M(P^*)|$. If $|M(P')|$ is strictly less than $|M(P^*)|$, the process is restarted with $P'$ playing the rôle of $P^*$. Eventually, after at most $p - 1$ iterations one obtains an optimal $p$-partition $P$ such that either (i) $M(P) = \emptyset$ or (ii) $P$ has the same number of midcuts as the immediately preceding partition. In both cases, $P$ is tuned.
We will now show that $P$ has the property stated above. Assume that for some component $C$ of $P_E$ for which $m_C \geq 1$, the components of an optimal $(m_C + 1)$-partition $M_C$ do not have equal lengths.

Let $L$ be the largest length of a component of $M_C$. Now, label all the components in $M_C$ “red”, if the length of the corresponding component is $L$, and “blue” otherwise. Let $\epsilon > 0$ be sufficiently small, so that if one or more cuts of $M_C$ are shifted by no more than $\epsilon$ in either direction along the edges containing them, then they are still midcuts and do not collide with each other. It suffices to take $\epsilon = \frac{1}{4}s_{\text{min}}$, where $s_{\text{min}}$ is the minimum length of a segment of $M_C$.

Since we are assuming that the components of $M_C$ do not have equal lengths, by connectedness of $M_C$, there exist a red component having a blue neighbour. Let $L' < L$ be the length of such a blue component. Shift the midcut separating the blue component and the red one towards the latter by $\delta = \min\{(L - L')/2, \epsilon\}$. The shift causes the length of the red component to become smaller than $L$, while the length of the blue component remains smaller than $L$. Hence the red component is relabelled as blue. Repeat this process until all the red components in $M_C$ are blue. In this way, one obtains an $(m_C + 1)$-partition of $C$ whose components have length less than $L$. This contradicts the optimality of $M_C$. Hence $P$ does have the property required in the statement of the theorem.

At the beginning of this section, we made the assumption that all edge-lengths are rational. Since an optimal $p$-partition remains such when all edge-lengths are multiplied by a positive number, we may assume, without loss of generality, that all edge-lengths are integers.

For $k \in \mathbb{N}$ let $\mathbb{N}/k$ be the set of all rational numbers $s/k$ such that $s \in \mathbb{N}$. Let $Q_p = \bigcup_{1 \leq k \leq p} \mathbb{N}/k$.

**Lemma 4.3.2** Let $T$ be a tree with integral edge-lengths. Let $P$ be a $p$-partition each of whose components has length a number in $\mathbb{N}/k$. Let $c_i, i = 1, \ldots, p - 1$ be the cuts of the partition. Then $\text{dist}(c_i) \in \mathbb{N}/k, i = 1, \ldots, p - 1$ (see page 53 for $\text{dist}(c)$).

**Proof:** By induction on $p$. If $p = 1$ there are no cuts, and the result is trivial. Note for what follows that $\mathbb{N}/k$ is closed under addition, and also under subtraction when the result is non-negative, and it contains the positive integers.

Assume true for all trees cut into $< p$ components. Let $T$ be a tree whose edges have integral lengths and which is cut into $p$ components whose lengths are in $\mathbb{N}/k$. If
all cuts are endcuts, the result is obvious. Suppose that there is at least one midcut. Let $c$ be a midcut which has no midcut as descendant, and let it be situated on edge $e = (u, v)$. All cuts below $c$ are endcuts, and hence if $c_i$ is one of these, $\text{dist}(c_i) = 0$ or $= l(\text{edge}(c_i))$, so that $\text{dist}(c_i)$ is an integer. Since the edge-lengths are integers, the down-component of $c$ has length $j + \text{dist}(c)$ for some integer $j$. Since this length is in $\mathbb{N}/k$, it follows that $\text{dist}(c)$ is in $\mathbb{N}/k$. All the other cuts lying on $e$ have the lengths of their down-components in $\mathbb{N}/k$, and so their distances from $u$ are in $\mathbb{N}/k$. Let $c_0$ be the cut on $e$ which is closest to $u$. Then the length of the segment $[u, c_0]$ is in $\mathbb{N}/k$. Therefore, if we delete the down-component of $u$ to obtain tree $T'$, then the edges of $T'$ also have integer lengths, and $T$ has the property of the statement of the lemma iff $T'$ has this property. By the inductive hypothesis, each cut $a$ of $T'$ has $\text{dist}(a) \in \mathbb{N}/k$, and the result follows.

**Theorem 4.3.3** If all edge-lengths are integers, there is an optimal $p$-partition such that the length of each segment belongs to $\mathbb{Q}_p$.

**Proof:** Let $P$ be the tuned optimal $p$-partition whose existence is guaranteed by Theorem 4.3.1, and let $C$ be any component of the associated partition $P_E$. If $C$ has no midcut in its interior then $C$ can be partitioned into edges and the thesis follows. If $C$ contains $k - 1$ midcuts then by Theorem 4.3.1 they divide $C$ into components of equal length $\frac{l(C)}{k}$. By Lemma 4.3.2, the length of every segment of $C$ belongs to $\mathbb{N}/k$.

We are now going to exhibit a (nonpolynomial) reduction from CONTINUOUS MIN-MAX TREE PARTITION to the MIN-MAX TREE PARTITION problem stated in the introduction of this chapter, under the assumption that all edge-lengths be integral. Here is the reduction of $T$ to a tree $T'$ with integral edge-lengths:

**REDUCTION**

1. Multiply by $p!$ all edge-lengths. Let $\lambda(e) = p!l(e)$ be the scaled length of $e$, for each edge $e$.
   (Notice that, as a consequence of Theorem 4.3.3, after the scaling there is an optimal $p$-partition of $T$ whose segments have integral lengths).

2. Assign a zero weight to every node of $T$.

3. For each edge $e$ of $T$, insert $\lambda(e)$ nodes with weight one along $e$.

4. Delete all nodes of degree one or two having zero weight, except for root.
The resulting tree $T'$ has $O(p!n l_{\text{max}})$ nodes, where $l_{\text{max}}$ is the maximum length of an edge of $T$. Let $X_k$ be the set of those points $x$ of $T$ such that $\text{dist}(x) \in N/k$ ($k = 1, 2, \ldots$).

Here we will give a formal definitions of descendant (we have discussed descendant informally on page 10). Let $x, y$ be two points of $T$. The point $y$ is a descendant of the point $x$ if either $\text{edge}(x) = \text{edge}(y)$ and $\text{dist}(x) \geq \text{dist}(y)$ or $\text{edge}(x) \neq \text{edge}(y)$ and $\text{edge}(y)$ is a descendant of $\text{edge}(x)$.

A example on reduction on a tree:
(a) A 3-partition of a tree $T$.
(b) Corresponding 3-partition of $T'$.

Figure 4.3: Example of reduction on a tree with 3-partitions, (a) is the original tree, (b) is the equivalent tree after reduction.

Lemma 4.3.4 There exists a one-to-one correspondence $\sigma$ between $X_{p!}$ and the edge-set of $T'$, such that, for all $x, y \in X_{p!}$, $y$ is a descendant of $x$ iff $\sigma(y)$ is a descendant of $\sigma(x)$ in $T'$. Furthermore, if $c_1, \ldots, c_{p-1}$ belong to $X_{p!}$ and are the cuts of a $p$-partition of $T$, then the $p$-partition of $T'$ whose cuts are $\sigma(c_1), \ldots, \sigma(c_{p-1})$ has the following property: for $i = 1, \ldots, p-1$ the weight of the down-component of $\sigma(c_i)$ is equal to the length of the down-component of $c_i$ multiplied by $p!$. A similar property holds for the root component.
The construction of $\sigma$ is straightforward. We indicate it on the small example of Figure 4.3. The actual proof will be presented in the Appendix.

**Theorem 4.3.5** If all edge-lengths are integers, CONTINUOUS MIN-MAX TREE PARTITION is (non-polynomially) reducible to MIN-MAX TREE PARTITION.

**Proof:** By Theorem 4.3.3 one does not lose in optimality by looking only at those $p$-partitions whose segments have their length in $\mathbb{Q}_p$. Consider any such partition $P$ and any cut $c$ of $P$. Since $\text{dist}(c)$ is the sum of the lengths of some segments, one has $\text{dist}(c) \in \mathbb{Q}_p \subseteq \mathbb{N}/p!$. Hence $c \in X_p$. The theorem then follows from Lemma 4.3.2.

As a consequence of the above theorem, one can solve CONTINUOUS MIN-MAX TREE PARTITION (under the assumption that all edge-lengths are integers) by constructing $T'$ as indicated above and then solving the corresponding MIN-MAX TREE PARTITION PROBLEM on $T'$ by the algorithm of Becker, Perl and Schach [10]. We call this algorithm for the CONTINUOUS MIN-MAX problem the **discretized shifting algorithm**. For future reference, we recall here the algorithm of Becker, Perl and Schach, quoting directly from their paper. (The tree $T$ is assumed to be rooted at one of its leaves).

**SHIFTING ALGORITHM**

1. Assign all $p-1$ cuts to the unique edge incident with the root $r$. Initialise the data and set $\text{BEST-MINMAX-SO-FAR} \leftarrow \infty$, and set $\text{BEST-PARTITION-SO-FAR}$ equal to the starting configuration.

2. While the top component is not a heaviest component, perform Step 3 to 5.

3. Find a cut $c$ with a heaviest down-component and down-shift it from its current edge to son-edge $e$ containing no cuts, maximizing the weight $\text{RDC}(c)$ of the resulting down-component of the shifted cut $c$. If no such vacant edge exists, then stop.

4. Traverse the path from $\text{tail}(e)$ to the root in the bottom-up direction until a vertex $v$, which is head of a cut, is encountered. For each vertex $w$ on that path having a cut incident from $w$, perform the following: If the down-component of a cut incident from $w$ is lighter than the down-component of the vacant son-edge $e_v$ of $w$ on the path, then side-shift that cut to edge $e_v$. If more than one cut incident from $w$ can be side-shifted, choose a cut with a lightest down-component.

59
5. Update and set HEAVIEST equal to the weight of the largest component in the current partition \( P \). If \( \text{HEAVIEST} < \text{BEST-MINMAX-SO-FAR} \), then set \( \text{BEST-MINMAX-SO-FAR} \leftarrow \text{HEAVIEST} \) and set \( \text{BEST-PARTITION-SO-FAR} \leftarrow P \). Return to Step 2.

Becker, Perl and Schach [10] further defined a terminating position to be a partition at which the algorithm terminates. A final value of BEST-PARTITION-SO-FAR is called a resulting partition of the algorithm.

Stage 1 Stage 2 Stage 3 Stage 4 Stage 5 Stage 6

Figure 4.4: Example of the discrete shifting algorithm on a tree with 4-partitions.

Perl and Vishkin [53] have shown that the above algorithm can be implemented, with a suitable data structure, so as to run in \( O(R(p-1)(p-1 + \log_2 d) + n) \) time, where \( R \) is the number of edges in the radius of \( T \) and \( d \) is the maximum degree in \( T \).

Since \( T' \) has \( O(p!nl_{\text{max}}) \) nodes, the resulting algorithm for CONTINUOUS MIN-MAX TREE PARTITION has a time-complexity \( O(R(p-1)(p-1 + \log_2 d) + p!nl_{\text{max}}) \). Hence the algorithm is pseudo-polynomial for every fixed \( p \).

4.4 A Polynomial Algorithm

The discretized shifting algorithm is not an efficient one for the edge-length trees, because each cut moves very slowly down a long edge-length. Hence we derive from the discretized shifting algorithm a much faster algorithm, which runs in polynomial time.
We will down-shift cuts simultaneously in stages. Group all cuts that have the heaviest down component in each stage and label them active. In each stage down shift only the active cuts with respect to the bottleneck, until a bottleneck point is reached.

A bottleneck of a down-shift stage is a real number which restricts the distance traveled by any cut. The point at which the constraints are violated is also referred to as a bottleneck point.

At any iteration of the discretized shifting algorithm, call a cut \( c' \) active if it is eligible for down-shifting, that is, if \( c' \) has the heaviest down-component. The subgraph induced by the set of all active cuts is a rooted forest \( F \) in the cut-tree. Any connected component of \( F \) will be call a fleet. Its leaves will be call the front cuts, its root the rear cut, and the father of its root the pier of the fleet. A cut will be called passive if it is not active and it is the father of some active cut, that is, it is the pier of some fleet. Finally, a cut is neutral if it is neither active nor passive.

For the sake of reference, we shall classify down-shifts in the discretized algorithm as follows: A down-shift of a cut from edge \( e \) to a son-edge \( g \) will be said to be of the first kind if the common endpoint \( j \) of \( e \) and \( g \) is a fork; otherwise (that is, if \( j \) has outdegree 1) the down-shift is called of the second kind. In the discretized shifting algorithm, long sequences of consecutive down-shifts of the second kind typically occur, and they follow a definite pattern.

We define the speed \( s_c \) of active cut \( c \) as the number of those descendants of \( c \) in the cut-tree that belong to the same fleet as \( c \) (including \( c \) itself), then \( c \) is down-shifted \( s_c \) times, when the frontcut is down-shifted once. It follows from the definition that the speed of \( c \) is always equal to \( 1 + \) the sum of the speeds of the sons of \( c \) that belong to the same fleet as \( c \).

In the discretized shifting algorithm, there is a second type of shift, the side-shift, where a cut is shifted from its present edge to a brother-edge. The purpose of the side-shift is to correct some previous shifts which are now wrong because of the last down-shift. There are two important properties of the side-shifts in the discretized shifting: Firstly, a side-shift only occurs at a fork of a tree, where cuts are incident directly from the fork. Secondly, a side-shift is needed, only after some down-shifts are made.

Any edge (fork), in the path from any front cut to the pier of a fleet of active cuts in the tree, will be called the neutral edge (the neutral fork). Any neutral fork, with non-active cuts incident directly from it, is called the passive fork, similarly, any neutral
edge e incident from the passive fork, with cut at tail(e), is called the passive edge. Any passive fork incident with cuts eligible for side-shifting, is called the active fork. (A cut is eligible for side-shift, if the down-component of the cut is less or equal to some down-component of a passive edge.)

Let us now examine the effect of the shifting of the cuts in the tree T' (read page 58 on T') on the corresponding cuts in the tree T.

Call a cut c in T blocked if c = head(e) for some edge e, and unblocked otherwise. A down-shift of a cut c in the tree T is an operation which replaces c by another cut ĉ in the down-tree of c. We will need to consider only two special types of down-shifts, namely "jumps" and "slides". A jump of a blocked cut c = head(e) replaces c by ĉ = tail(g), where g is a son-edge of e.

Next, let c = (edge(c), dist(c)) be an unblocked cut, and let 0 ≤ α ≤ dist(c). The slide of c by α replaces c by ĉ = (edge(c), dist(c) - α).

A side-shift of a cut c, in the tree T, is an operation which replaces c by another cut ĉ in the down-tree of a brother-edge. Hence, a side-shift of a cut c = tail(e) replaces c by ĉ = tail(e'), where e' is a brother-edge of e.

A down-shift of the first kind and a side-shift in the tree T' induce, via the mapping σ⁻¹ (see page 59 for more details) a jump and a side-shift in T respectively. One or more successive down-shifts of the second kind of a cut c' in T' induce a slide of c = σ⁻¹(c') in T. By definition, a stage consists of either of a jump, a slide or a side-shift.

When the initial set A' of active cuts in T' is down-shifted to the final set B' of cuts, the corresponding sets of cuts in T change from A = σ⁻¹(A') to B = σ⁻¹(B'). For the time being, define the cuts in A to be active (a definition of "active cut" which is intrinsic to T will be given later). Since the notion of speed depends only on the set of active cuts and on the cut-tree, and since the latter is invariant under the mapping σ, the speed of any cut in A is well-defined, and it is equal to the speed of σ(c). The crucial observation here is that one can go from A to B by simultaneously down-shifting all cuts in A, each at its own speed. More precisely, each cut c ∈ A is made to slide by a distance proportional to its speed. The constant of proportionality, which is the distance moved by the front cut, is calculated by the algorithm in polynomial time, and at that stage, the shift of all the cuts in A could be made simultaneously. We can think of this as skipping over some of the motion through the individual vertices of the discretized shifting algorithm. Now we define the notion of "speed" as that of "speed relative to the front cuts". In other words, speed is the ratio of the distance travelled by a cut to
the distance travelled by its front cut. Hence, by definition, the speed of the front cuts of $A$ is $1$. We view sliding of cuts in the tree $T$ as a synchronous continuous process over the length of the tree. We give a simple example in Figure 4.5 to demonstrate this.

![Figure 4.5: A simple example: there are two cuts in a tree with just one edge $e$, where $l(e) = 6$. Now, let cut $c_1$ be placed three units away from head($e$) and let $c_2$ be placed at tail($e$). When $c_1$ is shifted one unit down towards head($e$), then $c_2$ needs to shift two units down, so that the down components of both $c_1$ and $c_2$ remain equal.](image)

However, during simultaneously down-shifting of all the cuts in $A$, we need to monitor all those cuts that have been effected by down-shifting of cuts in $A$. Hence we need to side-shift those cuts which need to be corrected.

The above considerations suggest a continuous shifting procedure for solving CONTINUOUS MIN-MAX PARTITION. The procedure works directly on the tree $T$ and consists of a finite number of stages. Each stage consists either of a single jump or of one or more concurrent slides, then is followed by a sequence of correction stages if corrections are needed, each correction stage consists of a single side-shift. At the beginning of each stage, a set of active cuts is identified. Then terms such as passive, neutral, fleet, pier, passive fork, passive edge and speed are well-defined. At each stage, if there is at least one blocked active cut, then one such cut jumps from the head of its current edge $e$ to the tail of a suitably chosen son-edge $g^*$ of $e$. If, on the other hand, all active cuts are unblocked, then they are simultaneously and continuously down-shifted, each at its own speed, until an “event” occurs (there are five different types of events). After the down-shifting, we search for the cuts which need to be side-shifted, we do the search from bottom-up direction. The procedure is designed in such a way that the following property holds: If $A$ and $B$ denote the initial set of active cuts and the final set of shifted cuts, respectively, then there exists a sequence of consecutive shifts in the discretized shifting algorithm that changes $\sigma(A)$ into $\sigma(B)$. Such a property is crucial in proving the correctness of the procedure.

In order to be more specific, we need to define a notion of down-component for
continuous down-shifts of cuts in the tree $T$. For each edge $e$ of $T$, let $h(e)$ be the length of the down-component of tail($e$). The function $h(e)$ can be recursively computed as follows.

If edge $e$ bears some cut, let $c_f$ be the first cut along $e$, that is, the cut at maximum distance from head($e$). Then $h(e) = l(e) - \text{dist}(c_f)$.

If edge $e$ bears no cut and $e$ is a leaf-edge, then $h(e) = l(e)$; otherwise $h(e) = l(e) + \sum_{g \in \text{Son}(e)} h(g)$, where $\text{Son}(e)$ is the set of all son-edges of $e$.

We prescribe that a blocked active cut $c$, when it has to jump, always jumps to $c' = \text{tail}(g^*)$, where $h(g^*) = \max_{g \in \text{Son}(e)} h(g)$. (If there are ties choose any such $g^*$) We write $c' = \text{jump}(c)$. We also say that $c$ jumps over $j$, the unique vertex where $c$ incident to and $c'$ incident from.

The down-component $R$ of an arbitrary cut $c$ is then defined as follows.

- When $c$ is unblocked, $R$ is equal to the down-component of $c$.
- When $c$ is blocked, $R$ is equal to the union of all the down-component of edges, where each edge is incident from $c$ (If edge($c$) is a leaf-edge we set $R = \phi$).
- The length of $R$ will be denoted by $DC(c)$. When $R = \phi$ we set $DC(c) = 0$.

We prescribe that a cut, $c = \text{tail}(e)$ say, incident from a passive fork $v_f$, when it has to side-shift, always shifts to $c' = \text{tail}(e')$, where $\text{tail}(e') = \text{tail}(e) = v_f$ and $DC(c') \geq DC(c)$. In fact, if $DC(c') = DC(c)$ then $s_{c'} > s_c$.

We define $\text{HEAVIEST} = \max\{DC(c) : c \text{ is a cut }\}$ and $\text{TopComponent} = DC(r_0)$. At the beginning of each stage, a cut $c^*$ is said to be active if $DC(c^*) = \text{HEAVIEST}$.

Let $PFork$ be the set of all passive forks and $PEdge$ be the set of all passive edges at the beginning of each slide stage and let $AFork$ be the set of all the active forks at the end of a slide stage. If $v_i \in AFork$ then let $E_{v_i} = \{e \in PEdge|\text{tail}(e) = v_i\}$. The speed of an edge $e$, denoted by $s_e$, is the sum of all the speed of the active cuts under the down-component of tail($e$).

If a blocked active cut exists, then one such cut $c^*$ is replaced by the unblocked cut $c^{**} = \text{jump}(c^*)$.

If all active cuts are unblocked, then they are simultaneously and continuously down-shifted, each at its own speed, along their edges. Throughout this sliding process, $DC(c)$
decreases for all active cuts \( c \), it remains constant for all neutral cuts \( c \), and it is non-decreasing for all passive cuts \( c \). Furthermore, the value of \( DC(c) \) remains identical for all the active cuts \( c \).

Now for each sliding stage, define a bottleneck \( b \) as follows: \( b \) is the distance travelled by any front cut from the beginning to the end of the stage. Then the overall effect of the continuous down-shifting process, during the stage, on each individual cut \( c \) is tantamount to the slide of \( c \) by \( b.\) (speed of \( c) = b.s.c.\)

The sliding stage ends when one of the following five types of event occurs.

1. Some active cut becomes blocked.
2. Some neutral cut becomes active.
3. Some passive cut becomes active.
4. Some non-active cut needs to be side-shifted.
5. A certain stopping condition is met.

If (5) occurs, then the algorithm halts.

If one defines the \( i \)-bottleneck \( b_i \) as the distance travelled by any front cut from the beginning of the stage to the first occurrence of an event of type \( i \), \( (i = 1, 2, 3, 4, 5) \), see above, then

\[
b = \min\{b_1, b_2, b_3, b_4, b_5\}.
\]

Continuous Shifting Algorithm Scheme

1. Initialise the data.
   Let \( \text{BEST-MINMAX-SO-FAR} := \infty \), and let \( \text{BEST-PARTITION-SO-FAR} \) equal to the starting configuration.

2. If there is no cut at \( r_0 \), introduce one if fewer than \( p - 1 \) cuts have previously been introduced. If \( p - 1 \) cuts have been introduced, introduce the dummy cut which remains forever at the root. If there is some blocked cut in \( A \), then go to 3 else go to 4.

3. Jump section.
   (a) If \( \text{TopComponent} \geq \text{HEAVIEST} \), then STOP.
(b) If there is some blocked cut in \( A \), then choose one, \( c \) say, and let \( c := \text{jump}(c) \).

(c) **Side-Shifting section.**
Traverse the path from \( \text{tail}(\text{jump}(c)) \) to the root in the bottom-up direction until a vertex \( v \), which is head of a cut, is encountered. For each neutral fork \( w \) on that path, perform the following: If the down-component of a cut incident from \( w \) is lighter than the down-component of the vacant son-edge \( e_s \) of \( w \) on the path, then side-shift that cut to edge \( e_s \). If more than one cut incident from \( w \) can be side-shifted, choose a cut with a lightest down-component.

(d) Update the data and set \( \text{HEAVIEST} \) equal to the weight of the largest component in the current partition \( P \).
If \( \text{HEAVIEST} \leq \text{BEST-MINMAX-SO-FAR} \), then
Let \( \text{BEST-MINMAX-SO-FAR} \leftarrow \text{HEAVIEST} \), and
let \( \text{BEST-PARTITION-SO-FAR} \leftarrow P \).
Go to 2.

4. **Sliding section.**

(a) If \( \text{TopComponent} \geq \text{HEAVIEST} \), then STOP.

(b) Sliding routine:
while there is some \( c \in A \) do
\[ c := (\text{edge}(c), \text{dist}(c) - b.s_c); \]
\[ A := A \setminus \{c\}; \]
endwhile

(c) **Side-Shifting Section.**
if \( b_4 \leq b \) then
while \( AFork \neq \emptyset \) do
Set \( v_i \) to be the active fork with the largest index in \( AFork \).
\{ The choice of \( v_i \) is made in the bottom-up direction \}
while there is some \( e \in E_{v_i} \) do
\[ A_e := \{ c | c \notin A, c \text{ incident from } v_i \text{ and } DC(e) = DC(c) \} \]
if \( A_e \neq \emptyset \) then
If \( c \in A_e \) is neutral then
\[ c := \text{tail}(e); \]
set \( c \) as passive;
else
    choose $c'$ where $s_{c'} = \min\{s_c | c \in A_e\}$;
    if $s_{c'} < s_{\text{tail}(e)}$ then
        $c' := \text{tail}(e)$;
    endif
endif

$E_v := E_v \setminus \{e\}$;
endwhile

$A_Fork := A_Fork \setminus \{v_i\}$;
endwhile
end if

(d) Update the data and set HEAVIEST equal to the weight of the largest component in the current partition $P$.
If $\text{HEAVIEST} \leq \text{BEST-MINMAX-SO-FAR}$, then
Let $\text{BEST-MINMAX-SO-FAR} \leftarrow \text{HEAVIEST}$, and
let $\text{BEST-PARTITION-SO-FAR} \leftarrow P$.
Go to 2.

Here we give the algorithm for calculating the speed of a cut in each stage:

Algorithm for calculating the speed of a cut

for $i = 1..k$ do  \{ $k$ is the number of cuts that have been introduced\}
begin
    speed(i) := 1;
    if $\text{son}(i) \neq \phi$ then
        for each $c_k \in \text{son}(i)$ do
            if $c_k \in A$ then speed(i) := speed($c_i$) + speed($c_k$);
        endif
    endif
end

Note that we introduced the cuts from 1 to $p - 1$ in that order during the continuous shifting algorithm, and hence the for loop does a bottom-up search on the cut-tree. Similar method is use for the bottom-up search of side-shifting cuts in the set $A_Fork$, when the vertex is indexed from top-down.

See Figures 4.6–4.8 for an example of the working of the algorithm.

Notice that each iteration of the loop corresponds to a single stage and a sequence of
Figure 4.6: The continuous shifting algorithm continues into Figure 4.7.
Figure 4.7: The continuous shifting algorithm continues into Figure 4.8. Note the algorithm side-shifts a cut at Stage 16.
correction side-shifts. The role of the dummy cut $c_p$ is to ensure that the optimality test (TopComponent $\geq$ HEAVIEST) is performed when $c_p$ becomes active, that is, when the length of the top-component becomes equal to HEAVIEST. Consider, for example, the problem of finding an optimal $p$-partition for the trivial tree consisting of a single edge $e$. When all the $p-1$ cuts become unblocked nothing would prevent them from being down-shifted all the way to head($e$), were it not for the presence of the dummy cut $c_p$ above the root. As soon as the lengths of the $p$ components become equal, $c_p$ becomes active and the go to loop halts since the condition TopComponent = HEAVIEST holds.

Let us give explicit expressions for the five bottlenecks. Let Active, Passive, Neutral, and Cut denote the sets of all active, passive, neutral, and arbitrary cuts, respectively, at the beginning of the stage. Let $L_c = \text{length of the maximum down-component of cut } c \text{ at the beginning of the stage};$  
$\text{MAXNEUTR} = \max \{L_c : c \in \text{Neutral} \} \text{ (} < \text{HEAVIEST)}.$

For any unblocked passive cut $c$, let $\text{AS}_c = \text{set of all active sons of } c \text{ in the cut-tree};$  
$S_c = \sum_{q \in \text{AS}_c} s_q$, where $s_q$ is the speed of $q;$  

For any blocked passive cut $c$ and for any son-edge $g$ of edge($c$), let $\text{AS}_c(g) = \text{set of all active sons of } c \text{ in the down-tree of } g$ and let
\[ S_c(g) = \sum_{q \in AS_c(g)} s_q. \]

Finally, for any edge \( e \), let
\[ AS_e = \text{set of all active cuts immediately below the down-component of } e; \]
\[ S_e = \sum_{c \in AS_e} s_c, \text{ where } s_c \text{ is the speed of } c; \]

**Proposition 4.4.1** One has
\[
\begin{align*}
    b_1 &= \min\{ \frac{\text{dist}(c \text{ to the next marker (cut or fork))}}{s_c} : c \in \text{Active} \}; \\
    b_2 &= \text{HEAVIEST - MAXNEUTR}; \\
    b_3 &= \min\{ \beta_c : c \in \text{Passive} \}; \\
    b_4 &= \min\{ \beta_u : u \in PFork \}; \\
    b_5 &= \text{HEAVIEST - TopComponent},
\end{align*}
\]

where
\[
\beta_c = \frac{\text{HEAVIEST} - L_c}{1 + S_c}, \quad \forall \text{ unblocked passive } c,
\]
\[
= \min\{ \frac{\text{HEAVIEST} - h(g)}{1 + S_c(g)} : g \in \text{Son(edge(e))}, \quad \forall \text{ blocked passive } c.
\]

and
\[
\beta_u = \min\{ \beta_e : e \in PEdge \text{ and } \text{tail}(e) = u \}; \\
\beta_e = \min\{ \beta^*_e : c \notin \text{Active}, c = \text{tail}(e') \text{ where } e' \text{ is a brother of } e \};
\]

where
\[
\beta^*_e = \frac{L_c - DC(e)}{S_e}, \quad \forall \text{ neutral } c \text{ incident from } u, \\
= \frac{L_c - DC(e)}{S_e - S_c}, \quad \forall \text{ passive } c \text{ incident from } u \text{ and } S_e > S_c.
\]

**Proof:** In any stage, let \( t \) be the distance travelled by the front cuts (\( t = 0 \) at the beginning of the stage) the length of the down-component of a cut \( c \) is given by
• HEAVIEST - t, for all c ∈ Active;
• Lc, for all c ∈ Neutral;
• Lc + Sc t, for all unblocked c ∈ Passive:
• \( \max\{h(g) + Sc(g)t; g \in Son(edge(c))\} \), for all blocked c ∈ Passive.

Moreover, if the active cut c slides along \( e = edge(c) \), then when its front cut has slid a distance \( t \), its distance from head(\( e \)) is given by dist(\( c \)) - Sc t.
The bottleneck \( b_1 \) is the smallest \( t \) for which such distance becomes zero. Hence Equation 4.1 holds.
The bottleneck \( b_2 \) is the solution to the equation (in the unknown \( t \))

\[
\text{HEAVIEST} - t = \text{MAXNEUTR}.
\]

Hence Equation 4.2 holds.

Similarly, \( b_3 \) is the solution to

\[
\text{HEAVIEST} - t = \text{TopComponent}.
\]

Hence Equation 4.5 holds.

Now, for every c ∈ Passive define \( \beta_c \) to be the distance travelled by all the active sons of c from the beginning of the stage to the first occurrence of the event "c becomes active".

First of all, Equation 4.3 holds.

If c is any unblocked passive cut, then \( \beta_c \) is the unique solution to the equation

\[
\text{HEAVIEST} - t = Lc + Sc t.
\]

Hence Equation 4.6 holds.

If c is any blocked passive cut, then \( \beta_c \) is a solution to the nonlinear equation

\[
\text{HEAVIEST} - t = \max\{h(g) + Sc(g)t : g \in Son((edge)(c))\}\].

One can check that the unique solution to this equation is given by

\[
\beta_c = \min\{\epsilon_c(g) : g \in Son(edge(c))\},
\]

where \( \epsilon_c(g) \) is the unique solution to the equation

\[
\text{HEAVIEST} - t = h(g) + Sc(g)t,
\]

72
that is,

\[ \varepsilon_c(g) = \frac{\text{HEAVIEST} - h(g)}{1 + S_c(g)}. \]

(See Figure 4.9 for a geometric interpretation.) Hence Equation 4.7 hold.

Finally, for every passive fork \( u \), define \( \beta_u \) to be the distance travelled by all the active cuts immediately below some down-component of the passive edge incident from \( u \), from the beginning of the stage to the first occurrence of the event, where some non-active cut incident from \( u \) needs to be side-shifted.

Hence Equation 4.9 and Equation 4.10 hold.

Now, any non-active cut incident from some fork is either a neutral cut or an unblocked passive passive cut. Let \( c \) be any non-active cut which incident at the tail(\( e' \)) and \( e' \) is a brother-edge of the passive edge \( e \):

If \( c \) is a neutral cut, then \( \beta_c \) is the unique solution to the equation

\[ h(e) + S_c t = L_c. \]
Hence Equation 4.12 hold. If \( c \) is any passive cut, then \( \beta_c \) is the unique solution to the equation

\[
h(c) + S_c t = L_c + S_c t.
\]

However, \( h(c) \leq L_c \) at the beginning of any stage, so a side-shift is needed iff \( S_c > S_e \). Hence Equation 4.13 hold.

### 4.5 Correctness of the Continuous Shifting Algorithm

We will prove the correctness of the continuous shifting algorithm, under the assumption that all edge-lengths are rational, using simulation: we show that each stage of the execution of the continuous shifting algorithm can be simulated by a sequence of consecutive moves of the discretized shifting algorithm.

Throughout this section we use lower-case letters to denote sets and functions referring to the tree \( T \) (for example, active, \( dc(c) \), head(\( c \)), ...) and upper-case letters for \( T' \) (for example, ACTIVE, \( DC(c') \), HEAD(\( c' \)), ...).

A fork in \( T \) clearly maps to a zero weighted node, a FORK, in \( T' \) (see page 57 for details). Hence, let the set of all the forks in \( T \) (\( T' \)) be denoted by fork(\( T \)) (FORK(\( T' \))). A cut \( c' \) in \( T' \) will be said to be blocked if HEAD(\( c' \)) is a fork or a leaf, and unblocked otherwise, that is, if HEAD(\( c' \)) has outdegree one. We shall never need to consider blocked cuts whose head is a leaf. Clearly, a cut \( c \) in \( T \) is blocked iff \( \sigma(c) \) is blocked in \( T' \).

The notion of unit cycle is introduced, to identify a special sequence of consecutive down-shifts of the second kind in \( T' \). A prerequisite for a unit cycle to start is that at the beginning all cuts to be down-shifted are unblocked and active. Let \( L \geq 1 \) be the common weight of the down-components of these cuts, and let \( A' \) be their index-set. We shall further assume that during the cycle, for each \( h \in A' \), the \( h \)-th cut remains unblocked. The level of a cut \( c' \in A' \) is the distance of \( c' \) from a front cut of \( A' \), that is, the largest number of edges in a path from \( c' \) to a front cut in the cut-tree.

**UNIT CYCLE**

begin
while there is some \( h \in A' \) such that \( DC(c'_h) = L \) do
select a lowest-level $h \in A'$ such that $DC(c'_h) = L$;  
while $h \in A'$ do  
\hspace{1em} down-shift $c'_h$ by 1 unit;  
\hspace{1em} $h := \text{FATHER}(h)$;  
endwhile  
endwhile  
end

A sequence of consecutive shifts in the tree $T'$ is said to be admissible if at each iteration the cut that is chosen to be down-shifted is active or to be side-shifted is in need of correction. Hence an admissible sequence can always be interpreted as a sequence of consecutive shifts in the discretized shifting algorithm.

A collective shift is a transformation from a labelled $p$-partition $P_1$ of $T$ to a labelled $p$-partition $P_2$ of $T$ such that, for each $k = 1, \ldots, p - 1$, the $k$-th cut of $P_2$ is a shift (down-shift or side-shift) of the $k$-th cut of $P_1$. Let $A_1$ and $A_2$ be the set of cuts of $P_1$ and $P_2$, respectively. We shall say that a collective shift can be simulated if there exists an admissible sequence of consecutive shifts that transforms $\sigma(A_1)$ into $\sigma(A_2)$. Examples of collective shifts are the jumps, the concurrent slides and the side-shifts that take place in jumping, sliding and side-shifting stages, respectively. We shall show that those stages can be simulated.

**Lemma 4.5.1** If $c_1, \ldots, c_{p-1}$ belong to $X_{pl}$ and are the cuts of a $p$-partition of $T$ and if in addition all edge-lengths are integers, one has

$$ dc(c_k) \in X_{pl}, \quad k = 1, \ldots, p - 1. $$

**Proof:** Let $P$ be the $p$-partition defined by the cuts $c_1, \ldots, c_{p-1}$. Since these belong to $X_{pl}$ and since all edge-lengths are integers, all segments of $P$ have their length in $X_{pl}$. It follows that, for each $k = 1, \ldots, p - 1$, the value of $dc(c_k)$, being the sum of the lengths of a finite number of segments, also belongs to $X_{pl}$. \hfill \Box

**Lemma 4.5.2** If all edge-lengths are integers, then the partition obtained at the end of each stage of the continuous shifting algorithm has the property that all its cuts belong to $X_{pl}$.

**Proof:** By induction on the number $v$ of stages. At the beginning of the algorithm, the initial partition obviously has the required property. Assume that such property holds
true at the beginning of stage $v \geq 1$. If the stage is a jumping one or a side-shifting one, then the property remains true immediately after the stage. If the stage is a sliding one then in view of Lemma 4.5.1 and of Proposition 4.4.1 the bottlenecks $b_1, b_2, b_3, b_4, b_5$ belong to $X_{pl}$. Hence also the bottleneck $b$ belongs to $X_{pl}$. By the inductive hypothesis, at the beginning of stage $v$ the cuts of the current partition belong to $X_{pl}$. Since each active cut is made to slide by an integral multiple of $b$ it follows that also the cuts of the partition obtained after sliding stage (and hence also at the beginning of stage $v + 1$) must belong to $X_{pl}$.

Lemma 4.5.3 If $c_1, \ldots, c_{p-1}$ belong to $X_{pl}$ and are the cuts of a $p$-partition of $T$ and if $c'_1 = \sigma(c_1), \ldots, c'_{p-1} = \sigma(c_{p-1})$ are the corresponding cuts in $T'$, then one has

$$DC(c'_h) = p! \ dc(c_h), \ h = 1, \ldots, p - 1$$

Proof: The lemma easily follows from Lemma 4.3.4 and from the definition of the function $dc(c)$.

Lemma 4.5.4 Under the hypotheses of Lemma 4.5.3,

(a) A cut $c_h$ is an active cut in $T$ iff $c'_h$ is an active cut in $T'$.

(b) $\text{ACTIVE} = \sigma(\text{active}), \ \text{PASSIVE} = \sigma(\text{passive}), \ \text{NEUTRAL} = \sigma(\text{neutral})$.

Proof:

(a) If $c_h \in \text{active}$, then $dc(c_h) \geq dc(c_k), \ k = 1, \ldots, p - 1$, then by Lemma 4.5.3 one has $DC(c'_h) = pldc(c_h) \geq pldc(c_k) = DC(c'_k)$. Hence $c'_h$ is active. Conversely, if $DC(c'_h) = pldc(c_h) < pldc(c_j) = DC(c'_j)$, a contradiction.

(b) From (a) one has $\sigma(\text{active}) = \text{ACTIVE}$. Now, since the notions of passive and neutral cut depend only on the notions of active cut and on the cut-tree, and since the latter is invariant under $\sigma$ after Lemma 4.3.4, one must also have $\sigma(\text{passive}) = \text{PASSIVE}$ and $\sigma(\text{neutral}) = \text{NEUTRAL}$.

Lemma 4.5.5 If throughout a unit cycle all cuts $c'_h, h \in A'$ are unblocked and if initially their down-component has weight $L \geq 1$ then

(a) throughout the execution of the unit cycle, when a cut is chosen to be down-shifted the weight of its down-component is either $L$ or $L + 1$.

(b) at the end of the unit cycle, for each $h \in A'$ one has $DC(c'_h) = L - 1$. 76
Proof:

(a) Right after the beginning of the inner while, a cut $c'_h$ such that $DC(c'_h) = L$ is down-shifted: this causes $DC(c'_m)$, where $c'_m$ is the father of $c'_h$, to become equal to $L + 1$. At each subsequent iteration of the inner while a cut $c'_k$ such that $DC(c'_k) = L + 1$ is down-shifted, causing the length of the down-component of the father of $c'_k$ to become $L + 1$.

(b) The outer while is repeated until all cuts $c'_h$ ($h \in A'$) for which $DC(c'_h) = L$ have been down-shifted. Since these cuts remain unblocked throughout the unit-cycle, at the end of the outer while the length of their resulting down-component must be $L - 1$.

Lemma 4.5.6 During a unit cycle, each cut $c'_h$, $h \in A'$, is down-shifted a number of times equal to its speed.

Proof: Each front cut is down-shifted exactly once and its speed is 1. Any other cut is down-shifted every time any of its sons has been down-shifted. This implies the lemma.

Let $P$ be the $p$-partition at the beginning of a given sliding stage, let $c_1, \ldots, c_{p-1}$ be its cuts, let HEAVIEST be the largest length of a down-component of a cut of $P$, and let $A$ be the index-set of the active cuts.

For any $t$, $0 \leq t \leq b$, let $P(t)$ be the partition obtained from $P$ after each cut $c_h, h \in A$, is made to slide by $t \cdot s_c$. Let $c_1(t), \ldots, c_{p-1}(t)$ be the cuts of $P(t)$.

Lemma 4.5.7 For each $0 \leq t < b$, the index-set of the active cuts of $P(t)$ remains equal to $A$, and all these active cuts remain unblocked.

Proof: For each $0 \leq t < b$ and for each $h \in A$ one has $dc(c_h(t)) = HEAVIEST - t$.

From the definition of the bottleneck $b$, it follows that for each $0 \leq t < b$ and for each $h \in A$, all cuts $c_h$ are unblocked, and

$$HEAVIEST - t > dc(c_h(t)), \quad k \notin A.$$ 

Hence the lemma follows.
Lemma 4.5.8 During any given sliding stage for each $0 \leq t < b$ such that $t \in X_{pl}$, the collective slide from $P(t)$ to $P(t + \frac{1}{pl})$ can be simulated.

Proof: If $P$ is the $p$--partition at the beginning of the sliding stage, then by Lemma 4.5.2, all its cuts $c_1, \ldots, c_{p-1}$ belong to $X_{pl}$. Since, for each $h \in A$, $c_h(t)$ is obtained from $c_h$ through a down-shift by $t \cdot \text{speed}(c_h) \in X_{pl}$. By Lemma 4.5.7, all active cuts of $P(t)$ are unblocked and they are precisely the cuts $c_h(t)$, $h \in A$. All these cuts have a down-component with length $\text{HEAVIEST} - t$. Since all cuts of $P(t)$ belong to $X_{pl}$, there is a corresponding partition $P' = P'(t)$ whose cuts are $c'_1 = \sigma(c_1(t)), \ldots, c'_{p-1} = \sigma(c_{p-1}(t))$. By Lemma 4.5.4 all cuts $c'_h$, $h \in A'$, are active and unblocked. Thus, if $L = \max\{DC(c'_k) : k = 1, \ldots, p-1\}$, by Lemma 4.5.3, one must have $L = p!(\text{HEAVIEST} - t)$.

CLAIM: For each $k \notin A$, if $c'_k = \sigma(c_k(t + \frac{1}{pl}))$, then one has $DC(c'_k) \leq L - 1$.

We shall first prove the CLAIM under the following hypothesis.

(H) : In the partition $P(t + \frac{1}{pl})$ all cuts $c_h(t + \frac{1}{pl})$, $h \in A$, remain active and unblocked.

Under this assumption, one must have, for all $h \in A$, $k \notin A$, $dc(c_k(t + \frac{1}{pl})) \leq dc(c_k(t + \frac{1}{pl})) = \text{HEAVIEST} - t - \frac{1}{pl}$.

Hence, taking into account Lemma 4.5.3,

$$DC(c'_k) = p! \ dc(c_k(t + \frac{1}{pl})) \leq p!(\text{HEAVIEST} - t) - 1 = L - 1.$$ 

Notice that H certainly holds for all $0 \leq t < b - \frac{1}{pl}$, in view of Lemma 4.5.7. Furthermore, H is also satisfied for $t = b - \frac{1}{pl}$ when the sliding stage is not of type 1.

Hence the only remaining case is when $t = b - \frac{1}{pl}$ and the sliding stage is of type 1.

In this case, by the definition of bottleneck one has

$$\text{HEAVIEST} - b \geq dc(c_k(b)), \ \forall k \notin A.$$ 

Hence, for $k \notin A$, one has

$$DC(c'_k) = p! \ dc(c_k(b)) \leq p!(\text{HEAVIEST} - (b - \frac{1}{pl})) - 1 = L - 1.$$ 

Thus the CLAIM is proved.
Next, consider a unit cycle with starting partition $P'$. Taking into account Lemma 4.5.6, one sees that the partition of $T'$ obtained at the end of the unit cycle actually coincides with the partition $P''$ whose cuts are $c'_1, \ldots, c'_{p-1}$. Notice that throughout the unit cycle the $k$-th cut, for $k \notin A$, is never down-shifted. Hence the weight of its down-component is non-decreasing during the cycle, and since it does not exceed $L$ at the end, it remains $\leq L$ throughout the cycle. Now, observe that by Lemma 4.5.5 (a) every time a cut is chosen to be down-shifted the weight of its down-component is either $L$ or $L+1$. Therefore, the down-shifts of the unit cycle form an admissible sequence that transforms $\{\sigma(c_1(t)), \ldots, \sigma(c_{p-1}(t))\}$ into $\{\sigma(c_1(t + \frac{1}{p!})), \ldots, \sigma(c_{p-1}(t + \frac{1}{p!}))\}$. \[\]

**Lemma 4.5.9** Every jump can be simulated.

**Proof:** When a blocked cut jumps from head($e$) to tail($g^*$), where $g^* \in Son(e)$ and $h(g^*) = \max\{h(g) | g \in Son(e)\}$, its image in $T'$ is down-shifted from edge $\sigma(\text{head}(e))$ to edge $\sigma(\text{tail}(g^*))$. In view of Lemma 4.3.4, for each $g \in Son(e)$ the weight of the down-component of $\sigma(\text{tail}(g))$ is equal to $plh(g)$. The lemma easily follows. \[\]

**Lemma 4.5.10** Every side-shift can be simulated.

**Proof:** There are two type of down-shifts in the continuous shifting algorithm, thus we need to show that, for each type of down-shift, the side-shift stage can be simulated.

First, after a jump stage: When a cut is side-shifted from tail($e$) to tail($e^*$), where $e^* \in Brother(e)$ and $DC(e) < DC(e^*)$, its image in $T'$ is side-shifted from edge $\sigma(\text{tail}(e))$ to edge $\sigma(\text{tail}(e^*))$. In view of Lemma 4.3.4, for each $e' \in Brother(e)$ the weight of the down-component of $\sigma(\text{tail}(e'))$ is equal to $plh(e')$. In this case the lemma easily follows.

Second, after a slide stage: When a cut, $c$, is side-shifted from tail($e$) to tail($e^*$), where $e^* \in Brother(e)$, $dc(e) = dc(e^*)$ and $S_c > S_e$, its image in $T'$ is side-shifted from edge $\sigma(\text{tail}(e))$ to edge $\sigma(\text{tail}(e^*))$. In view of Lemma 4.3.4, for each $e' \in Brother(e)$ the weight of the down-component of $\sigma(\text{tail}(e'))$ is equal to $plh(e')$. In this case, the discretized shifting algorithm will not side-shift $c'$, since $DC(c') = DC(e^*)$, but condition $S_c > S_e$ insured that $DC(e^*)$ will be greater than $DC(c')$ in the future. Hence the lemma follows. \[\]

**Lemma 4.5.11** When the continuous shifting algorithm stops, the discretized shifting algorithm also does.

79
Proof: The continuous shifting algorithm stops whenever for some partition $P$ the following condition holds: TopComponent $\geq$ HEAVIEST.

Let $L$ be the largest weight of a down-component of a cut in the corresponding partition $P'$ of $T'$ and let $W$ be the weight of the Top-Component of $P'$ of $T'$. By Lemma 4.5.3 one has:

$$L = pl \text{HEAVIEST} \leq pl \text{TopComponent} = W$$

So, the discretized shifting algorithm stops, since the stopping condition $L \leq W$ is satisfied.

Theorem 4.5.12 The continuous shifting algorithm is correct.

Proof: By Lemma 4.5.9 each jumping stage can be simulated. Each sliding stage can therefore also be simulated, since it can be thought of as a sequence of $plb$ consecutive collective slides

$$P = P(0) \rightarrow P(\frac{1}{pl}) \rightarrow P(\frac{2}{pl}) \rightarrow \ldots \rightarrow P(b - \frac{1}{pl}) \rightarrow P(b),$$

each of which can be simulated, in view of Lemma 4.5.8.

By Lemma 4.5.11 when the continuous shifting algorithm stops then the discretized shifting algorithm stops as well.

Hence a run of the continuous shifting algorithm can be simulated by a run of the discretized shifting algorithm. If $P$ and $P''$ are the final partitions generated by the continuous and the discretized algorithm, respectively, then the cuts of $P''$ are the images under $\sigma$ of the cuts of $P$. Since $P''$ is optimal, $P$ must also be optimal by Lemma 4.3.4.

4.6 Complexity of the Continuous Shifting Algorithm

We shall prove that the time-complexity of the continuous shifting algorithm is $O((n^2p + np^2)(p + d))$, where $d$ is the maximum degree.

A sliding stage is said to be of type $i$ if it ends with an event of type $i$, that is, if $b = b_i$ ($i = 1, \ldots, 5$). A sliding stage is hybrid if two or more among the four bottlenecks coincide, that is, if the stage is of more than one type.
Theorem 4.6.1 The continuous shifting algorithm runs in $O((n^2p + np^2)(p + d))$ time.

Proof: We shall first establish a polynomial upper bound on the number of stages.

CLAIM 1: The total number of jumping stages is at most $(n - 1)(p - 1)$.

Proof: Each jumping stage consists of a single jump. Since all cuts move down, no cut ever jumps over a fork more than once. Furthermore, no cut jumps over any leaf other than root. It follows that the total number of jumps (or equivalently, of jumping stages) never exceeds $(n - 1)(p - 1)$.

CLAIM 2: The total number of side-shift stages is at most $(d - 1)(n - 1)(p - 1)$, where $d$ is the maximum degree.

Proof: Each side-shift stage consists of a single side-shift. If a cut is to be side-shifted, then it must be incident from a fork. Firstly, a cut is incident from a fork, after a jump stage. Secondly, a cut will not be side-shifted to the same edge twice. There will be at most $(n - 1)(p - 1)$ cuts incident from the forks during the implementation of the algorithm, since there are at most $(n - 1)(p - 1)$ jumping stages. Furthermore, each cut can be side-shifted at most $d - 1$ times, at each fork. So, there are at most $(d - 1)(n - 1)(p - 1)$ side-shift stages.

CLAIM 3: The total number of sliding stages is at most $(n - 1)(p - 1)(2p - 2 + d) + 1$.

Proof: Let us assume for the moment that there is no hybrid stage.

When an active cut becomes blocked, before getting blocked again it must jump over some fork. Then, in view of CLAIM 1 and CLAIM 2, the number of stages of type 1 is at most $(n - 1)(p - 1)$ and the number of stages of type 4 is at most $(d - 1)(n - 1)(p - 1)$. Next, observe that a stage of type 2 or 3 always results in a unit increase of the number of active cuts (all the cuts that were active at the beginning of any such stage remain active at the end of the stage). As a consequence, any stage of type 2 or 3 is followed by at most $p - 2$ stages of these two types: afterwards, either a jump or a stage of type 1 or 5 must necessarily take place. Therefore, the number of stages of type 2 or 3 is bounded above by $2(n - 1)(p - 1)^2$. Finally, there is at most one stage of type 5, since right after it the algorithm halts.

The above bounds hold a fortiori if some sliding stage is hybrid. Thus the total number of sliding stages is at most $d(n - 1)(p - 1) + 2(n - 1)(p - 1)^2 + 1 = (n - 1)(p - 1)(2p - 2 + d) + 1$. 
(n = no. of nodes, p = no. of components = no. of cuts + 1)

Complexity of single stage

<table>
<thead>
<tr>
<th>Computation</th>
<th>Order of complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization,</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Active, Fleets, Passive, Neutral</td>
<td>(O(p))</td>
</tr>
<tr>
<td>(A_{fork}, P_{fork}, P_{edge}, A_{Edge})</td>
<td>(O(p))</td>
</tr>
<tr>
<td>(\text{jump}(c), \forall \text{ blocked } active \ c)</td>
<td>Overall (O(n))</td>
</tr>
<tr>
<td>speed ((c), \forall c \in \text{ Active} )</td>
<td>Overall (O(p))</td>
</tr>
<tr>
<td>(b_1)</td>
<td>(O(p))</td>
</tr>
<tr>
<td>(\text{MAXNEUTR})</td>
<td>(O(p))</td>
</tr>
<tr>
<td>(b_2, b_3, \text{BEST-MINMAX-SO-FAR, BEST-PARTITION-SO-FAR})</td>
<td>(O(1))</td>
</tr>
<tr>
<td>(A_{Sc}, S_c, \forall \text{ unblocked } passive \ c)</td>
<td>Overall (O(p))</td>
</tr>
<tr>
<td>(A_{Sc}(g), S_c(g), \forall \text{ blocked } passive \ c, \forall g \in \text{Son}(\text{edge}(c)))</td>
<td>Overall (O(n))</td>
</tr>
<tr>
<td>(\beta_c, \forall c \in \text{ Passive} )</td>
<td>Overall (O(p))</td>
</tr>
<tr>
<td>(\beta_u, \forall u \in P_{fork}, b_4)</td>
<td>Overall (O(n))</td>
</tr>
<tr>
<td>(b_3)</td>
<td>(O(p))</td>
</tr>
<tr>
<td>(b)</td>
<td>(O(1))</td>
</tr>
<tr>
<td>(\text{Sliding, Side-Shift})</td>
<td>(O(p))</td>
</tr>
<tr>
<td>(\text{Cut-tree})</td>
<td>(O(n))</td>
</tr>
<tr>
<td>(h(e), e \in E)</td>
<td>overall (O(n))</td>
</tr>
<tr>
<td>(\text{DC}(i), \forall i)</td>
<td>overall (O(p))</td>
</tr>
<tr>
<td>(\text{RDC}(e), \forall e \in P_{edge})</td>
<td>overall (O(n))</td>
</tr>
<tr>
<td>(\text{HEAVEST, TOPCOMP})</td>
<td>(O(p))</td>
</tr>
</tbody>
</table>

**Overall complexity of a single stage** \(O(n + p)\)

**No. of stages** \(O((np)(p + d))\)

**Overall complexity of the algorithm** \(O((n^2p + np^2)(p + d))\)

Table 4.1: COMPLEXITY OF THE CONTINUOUS SHIFTING ALGORITHM

Next, we analyze the complexity of any single stage. Our estimate is based on a somewhat crude implementation of the algorithm (in particular, we assume that all relevant values are computed from scratch in each stage, instead of being updated from the previous stage). Thus, it is not unlikely that a better bound can be achieved. On the other hand, our primary purpose here is to establish the polynomial complexity of the algorithm rather than looking after the details of an efficient implementation.

Table 4.1 summarizes the order of complexity of the different computations required by the algorithm. It turns out that the running time of the continuous shifting algorithm is \(O((n^2p + np^2)(p + d))\). Hence the algorithm is strongly polynomial.
Chapter 5

Continuous Max-Min Tree Partition

Becker, Simeone and Chiang [7] [8] give a greedy continuous down-shifting algorithm for solving the continuous Max-Min p-partition problem on a tree with rational valued edge-lengths. For practical purposes, this result is sufficient, since the assumption made is not unreasonable. However, it is possible to extend the algorithm for solving the continuous Max-Min p-partition on the tree with real valued edge-lengths. In this chapter, we will present the results from [7] [8].

In the previous chapter, we have solved the p-partition problem on $T$ with respect to the continuous Min-Max criterion. In this chapter we will solve the continuous Max-Min p-partition problem on $T$ with rational valued edge-lengths, algorithmically. To start with, the investigation of the continuous p-partition problem was motivated in the previous chapter. We used an example in Figure 4.1 to illustrate that the resulting p-partitions obtained from using those two criteria, may have different magnitude. Furthermore, in Theorem 4.1.1 the discrete Max-Min p-partition on $G$ was shown to be NP-hard.

One thing that needs to be clarified, is that some of the proof which will be presented in this chapter, is similar but not identical to the proof presented in Chapter 4. The obvious reason is that the criterion considered in the Max-Min partition problem is different to the one considers in the Min-Max partition problem. The algorithm presented in Chapter 4 considers the down-component of a cut and the algorithm which will be presented in this chapter, considers the resulting down-component of a cut. Given a continuous p-partition on $T$, the resulting down-component of a cut $c$ is different from the down-component of $c$ if only if $c$ is incident to a fork (This is not true in the discrete formulation).
The following is a brief summary of this chapter: Section 1 contains formulation and notation. Section 2 presents a pseudo-polynomial algorithm for solving the continuous Max-Min \( p \)-partition problem on \( T \) with rational edge-lengths. Section 3 makes the necessary improvements to give a polynomial algorithm and considers the complexity of the continuous algorithm. Section 4 proves the correctness of the polynomial algorithm.

5.1 Terminology of the Continuous Down-Shifting Algorithm

Again, the objective function for the continuous Max-Min \( p \)-partition is denoted by \( f; f : \Pi(T,p) \rightarrow \mathbb{R} \), where \( f(P) = \min\{l(C_i) | C_i \in P\} \) and \( l(C_i) \) is the sum of the edge-lengths of all the edge segments of the component \( C_i \) in the \( p \)-partition \( P \). Now, the problems of continuous Max-Min \( p \)-partition on \( T \) is finding \( P^* \) in \( \Pi(T,p) \) such that \( f(P^*) = \max\{f(P) | P \in \Pi(T,p)\} \). Furthermore, in this chapter we will call such partition as "optimal" \( p \)-partition (take note that the usage of optimal \( p \)-partition here is different to the one used in the previous chapter).

In the previous chapter, we have mentioned that the approach we used for solving the problems of continuous Min-Max tree partition is similar to the one presented in [7] and [8]. Thus, in this chapter, we will retain most of the notation and terminology used in the previous chapter (such as passive cut, down-component, neutral cut, jump, speed of a cut and fleet). We will specify any terminology that needs to be changed.

5.2 A Nonpolynomial Algorithm

The polynomial down-shifting algorithm [7] [8] is derived from the discrete down-shifting algorithm. Here is a brief description of how the discrete down-shifting algorithm is implemented on a tree \( T \): Under the assumption that the edge-lengths of \( T \) are rational. The tree \( T \) can be reducible to a vertex weighted tree \( T' \) (see page 57 on the reduction). The discrete Max-Min tree partition problem is then solved by the algorithm of Perl and Schach [52].

In this section, the algorithm for solving the continuous Max-Min tree partition will be presented. Although the results in this section consider Max-Min tree partition, the approach used to verify this algorithm is same as Chapter 4. The proofs in this section
have been omitted, because the similarity between the proofs in this section and the proofs in Chapter 4. However, we will state the results in this section and give the corresponding proofs from Chapter 4.

First, the following theorem is proved:

**Theorem 5.2.1** An optimal $p$-partition always exists.

This theorem provides the basis for searching an algorithm, which will solve the continuous Max-Min tree partition problem. The proof of this theorem is similar to the one proved in Theorem 4.2.1. The only change that needs to be made, is that the continuous function, $\Phi(d) = \max\{\varphi_0(d), \varphi_1(d), \ldots, \varphi_{p-1}(d)\}$ in $\Omega$, is redefined as $\Phi'(d) = \min\{\varphi_0(d), \varphi_1(d), \ldots, \varphi_{p-1}(d)\}$. Then it follows that the function $\Phi'(d)$, being continuous on the compact set $\Omega$, has a maximum in $\Omega$ by Weierstrass' Theorem. This proves the claim and the theorem.

The reduction relies on Theorems 5.2.2 and 5.2.4 below, the main results of this section.

We used similar terminology, used in the previous chapter, for the continuous Max-Min tree partition. Given a dissected tree, a cut $c$ is said to be an endcut if $c$ is an endpoint of some edge, and a midcut if $c$ belongs to the interior of some edge. Let $P$ be any given $p$-partition of the tree $T$, and let $q$ be the number of endcuts of $P$ plus one. One can associate with $P$ the $q$-partition $P_E$ whose cuts are the endcuts of $P$. The set of midcuts of $P$ will be denoted by $\text{Midcut}(P)$.

An optimal $p$-partition is said to be tuned if, for each component $C$ of $P_E$, the $m_C$ midcuts of $P$ that lie within $C$ define an optimal $(m_C + 1)$-partition $M_C$ of $C$.

**Theorem 5.2.2** Among the optimal $p$-partitions of $T$ there is always a tuned one. Furthermore, a tuned $p$-partition $P$ of $T$ has the following property: each component $C$ of $P_E$ is subdivided into components of equal length by the midcuts of $P$ that lie within $C$.

Since Theorem 5.2.1 proved the existence of Max-Min optimal partitions on $T$, Theorem 5.2.2 can be proved in a similar fashion to Theorem 4.3.1, where in the proof of Theorem 5.2.2 $L$ is the smallest length of a component of $M_C$.

Assume the condition given in the previous chapter, that all edge-lengths are rational. Since an optimal $p$-partition remains such when all edge-lengths are multiplied by a
positive number, we may assume, without loss of generality, that all edge-lengths are integers.

For \( k \in \mathbb{N} \) let \( \mathbb{N}/k \) be the set of all rational numbers \( s/k \) such that \( s \in \mathbb{N} \). Let \( Q_p = \bigcup_{1 \leq k \leq p} \mathbb{N}/k \).

Similarly we can prove the following result identically to the one proved in Lemma 4.3.2, hence we will only state the lemma here:

**Lemma 5.2.3** Let \( T \) be a tree with integral edge-lengths. Let \( P \) be a \( p \)-partition each of whose components has length a number in \( \mathbb{N}/k \). Let \( c_i, i = 1, \ldots, p-1 \) be the cuts of the partition. Then \( \text{dist}(c_i) \in \mathbb{N}/k, i = 1, \ldots, p-1 \).

The next theorem is fundamental for the reduction algorithm. Using Theorem 5.2.2 and Lemma 5.2.3, Theorem 5.2.4 can be proved similarly using the proof of Theorem 4.3.3.

**Theorem 5.2.4** If all edge-lengths are integers, there is an optimal \( p \)-partition such that the length of each segment belongs to \( Q_p \).

Now we are able to apply the (nonpolynomial) reduction from the continuous Max-Min tree partition to the discrete Max-Min tree partition problem. Under the assumption that all edge-lengths be integral, the reduction of \( T \) to a tree \( T' \) used the same reduction algorithm defined on page 57. The resulting tree \( T' \) has \( O(plnl_{max}) \) nodes, where \( l_{max} \) is the maximum length of an edge of \( T \). Used the same notation used on page 58, we have:

**Lemma 5.2.5** There exists a one-to-one correspondence \( \sigma \) between \( X_{pt} \) and the edge-set of \( T' \), such that, for all \( x, y \in X_{pt} \), \( y \) is a descendant of \( x \) iff \( \sigma(y) \) is a descendant of \( \sigma(x) \) in \( T' \). Furthermore, if \( c_1, \ldots, c_{p-1} \) belong to \( X_{pt} \) and are the cuts of a \( p \)-partition of \( T \), then the \( p \)-partition of \( T' \) whose cuts are \( \sigma(c_1), \ldots, \sigma(c_{p-1}) \) has the following property: for \( i = 1, 2, \ldots, p-1 \) the weight of the down-component of \( \sigma(c_i) \) is equal to the length of the down-component of \( c_i \) multiplied by \( pl \). A similar property holds for the root component.

The construction of \( \sigma \) is straightforward and is identical to the one stated in Lemma 4.3.4. The actual proof will be presented in the Appendix. Furthermore, using Theorem 5.2.4 and Lemma 5.2.5, the next theorem can be proved similarly to Theorem 4.3.5.
Theorem 5.2.6 If all edge-lengths are integers, CONTINUOUS MAX-MIN TREE PARTITION is (non-polynomially) reducible to MAX-MIN TREE PARTITION.

As a consequence of the above theorem, one can solve the continuous Max-Min tree partition (under the assumption that all edges are integers) by constructing $T'$ as indicated above and then solving the corresponding discrete Max-Min tree partition problem on $T'$ by the algorithm of Perl and Schach [52]. We call this algorithm for the CONTINUOUS MAX-MIN problem the discretized shifting algorithm. For future reference, we recall here the algorithm of Perl and Schach, quoting directly from their paper. (The tree $T$ is assumed to be rooted at one of its leaves).

![Figure 5.1: Example of the down-shifting algorithm on a tree with 4 partitions.](image)

**DOWN-SHIFTING ALGORITHM**

1. Assign all $p - 1$ cuts to the unique edge incident with the root $r$.
2. Find the weight $W_{\text{min}}$ of the current lightest component.
3. Find a shift of a cut $c$ to a son-edge $e$ containing no cuts, maximizing the weight $RDC(c)$ of the resulting down-component of the shifted cut $c$.
4. If $RDC(c) \geq W_{\text{min}}$ then perform the shift of the cut $c$ to the edge $e$ and return to Step 2.
5. Terminate. The weight $W_{\text{min}}$ is the weight of the lightest component of the Max-Min $p$-partition obtained.
Perl and Vishkin [53] have shown that the above algorithm can be implemented, with a suitable data structure, so as to run in \(O(np^2 + n \log n)\) time.

Since \(T'\) has \(O(p!nl_{\text{max}})\) nodes, the resulting algorithm for CONTINUOUS MAX-MIN TREE PARTITION has a time-complexity \(O(p!nl_{\text{max}}(p^2 + \log_2(p!nl_{\text{max}})))\). Hence the algorithm is pseudo-polynomial for every fixed \(p\).

### 5.3 A Polynomial Algorithm

The motivation for finding the polynomial algorithm for the continuous Max-Min tree partition problem is similar to the one provided for the continuous Min-Max continuous tree partition: The discretized down-shifting algorithm is not an efficient algorithm, since the algorithm takes small steps along very long edges. Our strategy will be to group sequences of consecutive down-shifts into “stages”, in such a way that

1. each stage can be performed in polynomial time;
2. there are polynomially many stages.

However, the polynomial down-shifting algorithm has one type of shifts, the down-shift. Thus, there are some changes made: A cut \(c\) is called an active cut, if it is eligible for down-shifting, that is, if \(c\) maximizes the weight \(RDC(c)\) of the down component results from down-shifting \(c\). Now, the terminology of fleet, front cut, rear cut, pier, passive cut, neutral cut and speed is similarly defined with respect to the definition of the active cut as in previous chapter (see page 61 for the exact terminology).

For the sake of a clearer presentation of this chapter, we will re-define some terminology, from previous chapter, which will be used in the algorithm:

Call a cut \(c\) in \(T\) blocked if \(c = \text{head}(e)\) for some edge \(e\), and unblocked otherwise. A down-shift of a cut \(c\) in the tree \(T\) is an operation which replaces \(c\) by another cut \(\hat{c}\) in the down-tree of \(c\). We will need to consider only two special types of down-shifts, namely “jumps” and “slides”. A jump of a blocked cut \(c = \text{head}(e)\) replaces \(c\) by \(\hat{c} = \text{tail}(g)\), where \(g\) is a son-edge of \(e\).

Next, let \(c = (\text{edge}(c), \text{dist}(c))\) be an unblocked cut, and let \(0 \leq \alpha \leq \text{dist}(c)\). The slide of \(c\) by \(\alpha\) replaces \(c\) by \(\hat{c} = (\text{edge}(c), \text{dist}(c) - \alpha)\).

We need to define a notion of resulting down-component for continuous down-shifts of cuts in the tree \(T\). For each edge \(e\) of \(T\), let \(h(e)\) be the length of the down-component
of tail(e). The function \( h(e) \) can be recursively computed as follows.

If edge \( e \) bears some cut, let \( f \) be the first cut along \( e \), that is, the cut at maximum distance from head(e). Then \( h(e) = l(e) - \text{dist}(f) \).

If edge \( e \) bears no cut and \( e \) is a leaf-edge, then \( h(e) = l(e) \); otherwise \( h(e) = l(e) + \sum_{g \in \text{Son}(e)} h(g) \), where \( \text{Son}(e) \) is the set of all son-edges of \( e \).

We prescribe that a blocked active cut \( c \), when it has to jump, always jumps to \( c' = \text{tail}(g^*) \), where \( h(g^*) = \max_{g \in \text{Son}(e)} h(g) \). (If there are ties choose any such \( g^* \)) We write \( c' = \text{jump}(c) \). We also say that \( c \) jumps over \( j \), the unique fork containing both \( c \) and \( c' \).

The resulting down-component \( R \) of an arbitrary cut \( c \) is then defined as follows.

- When \( c \) is unblocked, \( R \) is equal to the down-component of \( c \).
- When \( c \) is blocked, \( R \) is equal to the down-component of \( \text{jump}(c) \). (If edge(\( c \)) is a leaf-edge we set \( R = \phi \)).
- The length of \( R \) will be denoted by \( RDC(c) \). When \( R = \phi \) we set \( RDC(c) = 0 \). Notice that if \( c \) is blocked and edge(\( c \)) is not a leaf-edge then \( RDC(c) = RDC(\text{jump}(c)) \) by definition.

We define \( LARGEST = \max\{RDC(c) | c \text{ is a cut} \} \), and \( \text{SHORTEST} = \min\{DC(c) | c \text{ is a cut} \} \). At the beginning of each stage, a cut \( c^* \) is said to be active if \( RDC(c^*) = LARGEST \).

If a blocked active cut exists, then one such cut \( c^* \) is replaced by the unblocked cut \( c^{**} = \text{jump}(c^*) \).

If all active cuts are unblocked, then they are simultaneously and continuously down-shifted, each at its own speed, along their edges. Throughout this sliding process, \( RDC(c) \) decreases for all active cuts \( c \), it remains constant for all neutral cuts \( c \), and it is non-decreasing for all passive cuts \( c \). Furthermore, the value of \( RDC(c) \) remains identical for all the active cuts \( c \).

Now for each sliding stage, define a bottleneck \( b \) as follows: \( b \) is the distance traveled by any front cut from the beginning to the end of the stage. Then the overall effect of the continuous down-shifting process, during the stage, on each individual cut \( c \) is tantamount to the slide of \( c \) by \( b.(\text{speed of } c) = b.s_c \).
The sliding stage ends when one of the following four types of event occurs.

1. Some active cut becomes blocked.

2. Some neutral cut becomes active.

3. Some passive cut becomes active.

4. A certain stopping condition is met.

If (4) occurs, then the algorithm halts.

If one defines the \( i \)-bottleneck \( b_i \) as the distance traveled by any front cut from the beginning of the stage to the first occurrence of an event of type \( i \), \( i = 1, 2, 3, 4 \), see above, then

\[ b = \min\{b_1, b_2, b_3, b_4\}. \]

Continuous Down-Shifting Algorithm Scheme

1. Initialise the data.

2. Jump section.

   (a) If LARGEST < SHORTEST, then STOP.

   (b) If there is some blocked cut in \( A \), then choose one, \( c \) say, and let \( c := \) jump\((c)\).

   (c) If there is no cut at \( r_0 \), introduce one if fewer than \( p-1 \) cuts have previously been introduced. If \( p-1 \) cuts have been introduced, introduce the dummy cut which remains forever at the root.

   (d) Update the data.

3. If there is still some blocked cut in \( A \), then go to 2 else go to 4.

4. Sliding section.

   (a) If LARGEST = SHORTEST, then STOP.

   (b) Sliding routine:

   \[
   \text{while there is some } c \in A \text{ do}
   \]

   \[
   c := (\text{edge}(c), \text{dist}(c) - b, s_c);
   \]
\[ A := A \setminus \{c\}; \]

\hspace*{1cm} \text{endwhile}

(c) Update the data.

5. Go to 2.

Again we give the algorithm for calculating the speed of a cut in each stage:

Algorithm for calculating the speed of a cut

for \( i = 1 \ldots k \) do \( \{k \text{ is the number of cuts that have been introduced}\} \)
begin
\[ \text{speed}(i) := 1; \]
if \( \text{son}(i) \neq \phi \) then
for each \( c_k \in \text{son}(i) \) do
if \( c_k \in A \) then
\[ \text{speed}(i) := \text{speed}(c_i) + \text{speed}(c_k); \]
endif
end
end

See Figures 5.2-5.3 for an example of the working of the algorithm.

Let us give explicit expressions for the four bottlenecks. Let \( \text{Active}, \text{Passive}, \text{Neutral}, \) and \( \text{Cut} \) denote the sets of all active, passive, neutral, and arbitrary cuts, respectively, at the beginning of the stage. Let \( L_c = \text{length of the maximum resulting down-component of cut } c \text{ at the beginning of the stage} \);
\[ \text{MAXNEUTR} = \max\{L_c : c \in \text{Neutral}\} (\text{< LARGEST}). \]
For any unblocked passive cut \( c \), let
\[ \text{AS}_c = \text{set of all active sons of } c \text{ in the cut-tree}; \]
\[ S_c = \sum_{q \in \text{AS}_c} s_q, \text{ where } s_q \text{ is the speed of } q; \]
Finally, for any blocked passive cut \( c \) and for any son-edge \( g \) of edge\( (c) \), let
\[ \text{AS}_{c}(g) = \text{set of all active sons of } c \text{ in the down-tree of } g \text{ and let} \]
\[ S_c(g) = \sum_{q \in \text{AS}_{c}(g)} s_q. \]

Proposition 5.3.1 \textbf{One has}
\[ b_1 = \min\{\frac{\text{dist}(c \text{ to the next marker (=cut or fork))}}{s_c} : c \in \text{Active}\}; \quad (5.1) \]
Figure 5.2: The continuous down-shifting algorithm continues into Figure 5.3.
Figure 5.3: The continuous down-shifting algorithm.
\[ b_2 = \text{LARGEST} - \text{MAXNEUTR}; \quad (5.2) \]
\[ b_3 = \min\{\beta_c : c \in \text{Passive}\}; \quad (5.3) \]
\[ b_4 = \text{LARGEST} - \text{SHORTEST}, \quad (5.4) \]

where

\[
\beta_c = \frac{\text{LARGEST} - L_c}{1 + S_c}, \quad \forall \text{ unblocked passive } c \quad (5.5)
\]
\[
= \min\left\{\frac{\text{LARGEST} - h(g)}{1 + S_c(g)} : g \in \text{Son}(\text{edge}(c))\right\}, \quad \forall \text{ blocked passive } c. \quad (5.6)
\]

The proof of Proposition 5.3.1 is similar to the proof of Proposition 4.4.1, thus the proof will be omitted (see page 71 for detailed proof). Notice, from the Proposition 5.3.1, that \(b_1, b_2\) and \(b_3\) are always positive, while \(b_4\) is always non-negative.

We will give a detailed formal description of the continuous down-shifting algorithm in the Appendix.

For every fixed \(p\), in a tree \(T\) with \(n - 1\) edges, it was shown [7] that the time complexity of the improved continuous algorithm is \(O(n^2 p^2 + np^3)\), the actual proof is so similar to those proved in Theorem 4.6.1, we will show it at the Appendix.

The final result in [7] implies that a better bound on the running time can be achieved, provided that certain "pathologies" do not occur during the execution of the algorithm:

**REGULARITY HYPOTHESIS:** After each jumping stage and after each stage of type 1, the number of active cuts never decreases by more than one.

The above hypothesis is violated only in two special cases:

1. Before some jumping stage, there are at least 3 active cuts, and after the jump the resulting down-component of the cut that jumps becomes shorter than the resulting down-component of its father.
2. At the end of some stage of type 1, at least 2 active cuts become blocked.

**Theorem 5.3.2** Under the regularity hypothesis, the total number of stages is \(O(np)\). Hence the running time of the continuous down-shifting algorithm is \(O(n^2 p + np^3)\).

**Proof:** We shall make use of an amortized complexity argument. Let
\( v = \) total number of stages:
\( v_1 = \) total number of stages of type 1;
\( v_{2,3} = \) total number of stages of type 2 or 3 (or both);
\( v_{\text{jump}} = \) total number of jumps;
\( \alpha_0(\equiv 1) = \) initial number of active cuts;
and for \( i = 1, \ldots, v \):
\( \alpha_i = \) number of active cuts at the end of the \( i \)-th stage;
\( \Delta_i = \alpha_i - \alpha_{i-1} \).

Then one must have
\[ \Delta_i \geq 1, \text{ at the end of stage } i \text{ of type 2 or 3}; \]
and under the regularity hypothesis,
\[ \Delta_i \geq 1, \text{ at the end of each jumping or type 1 stage } i. \]

It follows that
\[
p - 1 \geq \alpha_v = \alpha_0 + \Delta_1 + \ldots + \Delta_v \geq 1 + v_{2,3} - v_1 - v_{\text{jump}} \geq 1 + v_{2,3} - 2(n-1)(p-1).
\]
Therefore, \( v_{2,3} \leq (2n - 1)(p - 1) - 1 = O(np) \). Since \( v_1 = O(np) \), \( v_{\text{jump}} = O(np) \) and \( v_4 \leq 1 \), the theorem follows.

### 5.4 Correctness of the Algorithm

In [7], the correctness of the continuous down-shifting algorithm, under the assumption that all edge-lengths are rational, is proved through simulation: we show that each stage of the execution of the continuous down-shifting algorithm can be simulated by a sequence of consecutive moves of the discretized down-shifting algorithm.

The approach used to prove the correctness of the continuous down-shifting algorithm is similar to the one shown in Chapter 4. The main difference, between the two algorithms, is that the criterion for cuts to be shifted in the continuous down-shifting algorithm is different from that of the continuous shifting algorithm.
In the continuous down-shifting algorithm, a cut is active if the cut has largest resulting down component. With this in mind, the notation and terminology will be similar to the one used on page 74. Use lower-case letters to denote sets and functions referring to the tree $T$ (for example, active, $de(c)$, head($c$), ...) and upper-case letters for $T'$ (for example, ACTIVE, $DC(c')$, HEAD($c'$), ...).

The unit cycle has the identical function as the one defined on page 74, but the cuts, which the index set $A'$ represent, are those cuts $c$ with $RDC(c) = L$. A prerequisite for a unit cycle to start is that at the beginning all cuts to be down-shifted are unblocked and active. Let $L \geq 1$ be the common weight of the resulting down-components of these cuts, and let $A'$ be their index-set. We shall further assume that during the cycle, for each $h \in A'$, the $h$-th cut remains unblocked. The level of a cut $c' \in A'$ is the distance of $c'$ from a front cut of $A'$, that is, the largest number of edges in a path from $c'$ to a front cut in the cut-tree.

**UNIT CYCLE**

begin
while there is some $h \in A'$ such that $RDC(c'_h) = L$ do
    select a lowest-level $h \in A'$ such that $RDC(c'_h) = L$ ;
    while $h \in A'$ do
        down-shift $c'_h$ by 1 unit ;
        $h :=$ FATHER($h$) ;
    endwhile
endwhile
end

A sequence of consecutive down-shifts in the tree $T'$ is said to be admissible if at each iteration the cut that is chosen to be down-shifted is active. Hence an admissible sequence can always be interpreted as a sequence of consecutive down-shifts in the discretized down-shifting algorithm.

A collective down-shift is a transformation from a labelled $p$-partition $P_1$ of $T$ to a labelled $p$-partition $P_2$ of $T$ such that, for each $k = 1, \ldots, p - 1$, the $k$-th cut of $P_2$ is a down-shift of the $k$-th cut of $P_1$. Let $A_1$ and $A_2$ be the set of cuts of $P_1$ and $P_2$, respectively. We shall say that a collective down-shift can be simulated if there exists an admissible sequence of consecutive down-shifts that transforms $\sigma(A_1)$ into $\sigma(A_2)$. Examples of collective down-shifts are the jumps and the concurrent slides that take place in jumping and sliding stages, respectively. We shall show that both can be simulated.
Similar proof to the one used in Lemma 4.5.1 can be used to show the following lemma:

**Lemma 5.4.1** If \(c_1, \ldots, c_{p-1}\) belong to \(X_p\) and are the cuts of a \(p\)-partition of \(T\) and if in addition all edge-lengths are integers, one has

\[
rdc(c_k) \in X_{p!}, \quad k = 1, \ldots, p - 1.
\]

In view of Lemma 5.4.1 and of Proposition 5.3.1, we can prove the following lemma (see Lemma 4.5.2 for the detailed proof):

**Lemma 5.4.2** If all edge-lengths are integers, then the partition obtained at the end of each stage of the continuous down-shifting algorithm has the property that all its cuts belong to \(X_p\).

**Lemma 5.4.3** If \(c_1, \ldots, c_{p-1}\) belong to \(X_p\) and are the cuts of a \(p\)-partition of \(T\) and if \(c'_1 = \sigma(c_1), \ldots, c'_{p-1} = \sigma(c_{p-1})\) are the corresponding cuts in \(T'\), then one has

\[
\begin{align*}
RDC(c'_h) &= p! \ rdc(c_h), \quad c_h \text{ blocked} \\
&= p! \ rdc(c_h) - 1, \quad c_h \text{ unblocked}.
\end{align*}
\]

**Proof:** The lemma easily follows from Lemma 5.2.5 and from the definition of the function \(rdc(c)\). (Recall that for an unblocked cut \(c\) in \(T\), the resulting down-component \(rdc(c)\) is the same as its down-component; while its corresponding cut \(c'\) in \(T'\) has resulting down-component obtained by shifting its cut down 1 unit).

**Lemma 5.4.4** Under the hypotheses of Lemma 5.4.3,

(a) If there is some blocked active cut in \(T\), then \(c_h\) is a blocked active cut in \(T\) iff \(c'_h\) is a blocked active cut in \(T'\).

(b) If all active cuts in \(T\) are unblocked, then \(c_h\) is an unblocked active cut in \(T\) iff \(c'_h\) is an unblocked active cut in \(T'\).

(c) If all active cuts in \(T\) are unblocked, and there is no blocked cut \(c_k\) such that \(rdc(c_k) = \text{LARGEST} - \frac{1}{p!}\), then one has

\[
\begin{align*}
\text{ACTIVE} &= \sigma(\text{active}), \quad \text{PASSIVE} = \sigma(\text{passive}), \quad \text{NEUTRAL} = \sigma(\text{neutral}).
\end{align*}
\]
Proof:

(a) $c_h$ is blocked iff $c'_h$ is blocked. If $c_h \in \text{active}$, then $rdc(c_h) \geq rdc(c_k), \quad k = 1, \ldots, p - 1$. If $c_h$ is active and blocked, then by Lemma 5.4.3 one has

$$RDC(c'_h) = plrdc(c_k) \geq plrdc(c_h) \geq RDC(c'_k).$$

Hence $c'_h$ is active. Conversely, if $RDC(c'_h) = plrdc(c_h) < plrdc(c_j) = RDC(c'_j)$, a contradiction.

(b) Assume that all active cuts in $T$ are unblocked, and let $c_h \in \text{active}$. Let us show that $c'_h \in \text{ACTIVE}$. Let $c_k$ be an arbitrary cut in $T$, and let $c'_k = \sigma(c_h)$. If $c_k \in \text{active}$, then $c_k$ is unblocked and therefore

$$RDC(c'_h) = plrdc(c_h) - 1 = plrdc(c_k) - 1 = RDC(c'_k).$$

On the other hand, if $c_k \notin \text{active}$ then $rdc(c_h) \geq rdc(c_k)$. By Lemma 5.4.3 one must have $rdc(c_h) \geq rdc(c_k) + \frac{1}{pl}$. It follows that

$$RDC(c'_h) = plrdc(c_h) - 1 \geq plrdc(c_k) + \frac{1}{pl} - 1 = plrdc(c_k) \geq RDC(c'_k).$$

Hence $c'_h \in \text{ACTIVE}$. Moreover, $c'_h$ must be unblocked since $c_h$ is such.

(c) From (b) one has $\sigma(\text{active}) \subseteq \text{ACTIVE}.$

Under the assumptions of (c), for any $c_k \notin \text{active}$, one has $rdc(c_k) \leq \text{LARGEST} - \frac{2}{pl}$. Hence if $c_k$ is any active cut, $RDC(c'_k) \leq pl(\text{LARGEST} - \frac{2}{pl}) = plrdc(c_k) - 2 < RDC(c_h)$. It follows that $\sigma(\text{passive} \cup \text{neutral}) \subseteq \text{PASSIVE} \cup \text{NEUTRAL}$, which implies that $\sigma(\text{active}) \supseteq \text{ACTIVE}$. Thus $\sigma(\text{active}) = \text{ACTIVE}$. Since the notions of passive and neutral cut depend only on the notions of active cut and on the cut-tree, and since the latter is invariant under $\sigma$ after Lemma 5.2.5, one must also have $\sigma(\text{passive}) = \text{PASSIVE}$ and $\sigma(\text{neutral}) = \text{NEUTRAL}.$

Lemma 5.4.5 If throughout a unit cycle all cuts $c'_h, h \in A'$ are unblocked and if initially their resulting down-component has weight $L \geq 1$ then

(a) throughout the execution of the unit cycle, when a cut is chosen to be down-shifted the weight of its resulting down-component is either $L$ or $L + 1$.

(b) at the end of the unit cycle, for each $h \in A'$ one has $RDC(c'_h) = L - 1$. 

98
Lemma 5.4.5 can be proved in a similar fashion to Lemma 4.5.5: In the proof of Lemma 4.5.5 the down-component of a cut is replaced by the resulting down-component for the proof of Lemma 5.4.5, for example replace $DC$ by $RDC$.

Lemma 5.4.6 During a unit cycle, each cut $c_h$, $h \in A'$, is down-shifted a number of times equal to its speed.

This lemma can proved in an identical way to Lemma 4.5.6.

Let $P$ be the $p$-partition at the beginning of a given sliding stage, let $c_1, \ldots, c_{p-1}$ be its cuts, let LARGEST be the largest length of a resulting down-component of a cut of $P$, and let $A$ be the index-set of the active cuts.

For any $t$, $0 \leq t \leq b$, let $P(t)$ be the partition obtained from $P$ after each cut $c_h$, $h \in A$, is made to slide by $t \cdot s_{c_h}$. Let $c_1(t), \ldots, c_{p-1}(t)$ be the cuts of $P(t)$.

Lemma 5.4.7 For each $0 \leq t < b$, the index-set of the active cuts of $P(t)$ remains equal to $A$, and all these active cuts remain unblocked.

Proof: For each $0 \leq t < b$ and for each $h \in A$ one has $rdc(c_h(t)) = \text{LARGEST} - t$

From the definition of the bottleneck $b$, it follows that for each $0 \leq t < b$ and for each $h \in A$, all cuts $c_h$ are unblocked, and

$$\text{LARGEST} - t > rdc(c_h(t)), \quad k \notin A.$$  

Hence the lemma follows.

Lemma 5.4.8 During any given sliding stage for each $0 \leq t < b$ such that $t \in X_p$, the collective slide from $P(t)$ to $P(t + \frac{1}{p})$ can be simulated.

Proof: If $P$ is the $p$-partition at the beginning of the sliding stage, then by Lemma 5.4.2, all its cuts $c_1, \ldots, c_{p-1}$ belong to $X_p$. Since, for each $h \in A$, $c_h(t)$ is obtained from $c_h$ through a down-shift by $t \cdot \text{speed}(c_h) \in X_p$. By Lemma 5.4.7, all active cuts of $P(t)$ are unblocked and they are precisely the cuts $c_h(t)$, $h \in A$. All these cuts have a resulting down-component with length $\text{LARGEST} - t$. Since all cuts of $P(t)$ belong to $X_p$, there is a corresponding partition $P' = P'(t)$ whose cuts are $c'_1 = \sigma(c_1(t)), \ldots, c'_{p-1} = \sigma(c_{p-1}(t))$. By Lemma 5.4.4 (b) all cuts $c'_h$, $h \in A'$, are active and unblocked. Thus, if $L =$
\[ \max \{ RDC(c_k) \mid k = 1, \ldots, p - 1 \} \], by Lemma 5.4.3, one must have \( L = p!(\text{LARGEST} - t) - 1 \).

**CLAIM:** For each \( k \notin A \), if \( c_k' = \sigma(c_k(t + \frac{1}{p!})) \), then one has \( RDC(c_k') \leq L \). We shall first prove the **CLAIM** under the following hypothesis.

\( (H) \) : In the partition \( P(t + \frac{1}{p!}) \) all cuts \( c_h(t + \frac{1}{p!}), h \in A \), remain active and unblocked.

Under this assumption, one must have, for all \( h \in A, k \notin A, \) \( rdc(c_k(t + \frac{1}{p!})) \leq rdc(c_h(t + \frac{1}{p!})) = \text{LARGEST} - t - \frac{1}{p!} \).

Hence, taking into account Lemma 5.4.3,

\[ RDC(c_k') \leq p! \cdot rdc(c_k(t + \frac{1}{p!})) = p!(\text{LARGEST} - t) - 1 = L. \]

Notice that \( H \) certainly holds for all \( 0 \leq t < b - \frac{1}{p!} \), in view of Lemma 5.4.7. Furthermore, \( H \) is also satisfied for \( t = b - \frac{1}{p!} \) when the sliding stage is not of type 1.

Hence the only remaining case is when \( t = b - \frac{1}{p!} \) and the sliding stage is of type 1.

In this case, by the definition of bottleneck one has

\[ \text{LARGEST} - b \geq rdc(c_k(b)), \quad \forall k \notin A. \]

Hence, for \( k \notin A \), one has

\[ RDC(c_k') \leq p! \cdot rdc(c_k(b)) \leq p! \cdot (\text{LARGEST} - (b - \frac{1}{p!})) - 1 = L. \]

Thus the **CLAIM** is proved.

Next, consider a unit cycle with starting partition \( P' \). Taking into account Lemma 5.4.6, one sees that the partition of \( T' \) obtained at the end of the unit cycle actually coincides with the partition \( P'' \) whose cuts are \( c_1', \ldots, c_{p-1}' \). Notice that throughout the unit cycle the \( k \)-th cut, for \( k \notin A \), is never down-shifted. Hence the weight of its resulting down-component is non-decreasing during the cycle, and since it does not exceed \( L \) at the end, it remains \( \leq L \) throughout the cycle. Now, observe that by Lemma 5.4.5 (a) every time a cut is chosen to be down-shifted the weight of its resulting down-component is either \( L \) or \( L + 1 \). Therefore, the down-shifts of the unit cycle form an admissible sequence that transforms \( \{\sigma(c_1(t)), \ldots, \sigma(c_{p-1}(t))\} \) into \( \{\sigma(c_1(t + \frac{1}{p!})), \ldots, \sigma(c_{p-1}(t + \frac{1}{p!}))\} \).
In view of Lemma 5.2.5, the next lemma can be proved in an identical way to Lemma 4.5.9.

Lemma 5.4.9 Every jump can be simulated.

Lemma 5.4.10 When the continuous down-shifting algorithm stops, the discretized shifting algorithm also does.

Proof: The continuous down-shifting algorithm stops whenever for some partition \( P \) one of the two following conditions holds:

(i) \( \text{LARGEST} < \text{SHORTEST} \)

(ii) all active cuts are unblocked and \( \text{LARGEST} = \text{SHORTEST} \). (that is, \( b_4 = 0 \))

In either case, let \( L \) be the largest weight of a resulting down-component of a cut in the corresponding partition \( P' \) of \( T' \) and let \( W_{\text{min}} \) be the smallest weight of a component \( P' \) of \( T' \). By Lemma 5.4.3 one has:

in case (i)\[
L \leq p!\text{LARGEST} < p!\text{SHORTEST} = W_{\text{min}}
\]

and in case (ii),

\[
L = p!\text{LARGEST} - 1 = p!\text{SHORTEST} - 1 < p!\text{SHORTEST} = W_{\text{min}}.
\]

In both cases the discretized down-shifting algorithm stops, since the stopping condition \( L < W_{\text{min}} \) is satisfied.

Using the lemmas obtained, Theorem 5.4.11 can be proved in an identical way to Theorem 4.5.12:

Theorem 5.4.11 The continuous down-shifting algorithm is correct.
Chapter 6

Conclusions and Computational Results

$p$-Median and $p$-centre problems have been reviewed in this thesis. It has been shown that the $p$-median problem and the $p$-centre problem are NP-hard on a general graph and are polynomially solvable on an acyclic graph (a tree). Furthermore polynomial algorithms for solving the continuous Max-Min tree partition problem and the continuous Min-Max tree partition problem have been presented.

It has been shown that in the $p$-median problem, it is justifiable to use the discrete formulation instead of the continuous formulation and hence simplify the computation of the algorithm.

We used the problem of locating a fixed number of fire hydrants along the street network to illustrate the need for the continuous formulation of the $p$-centre problem. In the $p$-centre problem, we have shown that the continuous formulation and the discrete formulation are not the same.

We reviewed an efficient algorithm of complexity $O(n\log^3 n)$ for the continuous $p$-centre problem on a tree as implemented by Megiddo and Tamir [46]. This algorithm used a parallel searching scheme, which improved the time complexity of the algorithm.

In some practical contexts, for example the problem of allotting the maintenance of a tree-like highway network among number of service units with equal work-capacities, it is not feasible to require that all the points on the same edge be served by the same service unit. This is because forbidding edge-splitting among different units may result in poor workload balance. This motivates the investigation of the continuous Min-Max
tree partition problem and the continuous Max-Min tree partition problem.

We examined the discrete algorithm for solving the Min-Max tree partition problem (the Max-Min tree partition problem respectively), then a polynomial algorithm was derived for solving the continuous Min-Max tree partition problem (the continuous Max-Min tree partition problem respectively). We were able to show the correctness for both algorithms for trees with rational edge-lengths. However, both algorithms can be applied on a tree with real valued edge-lengths.

The bounds for the time complexity of the continuous shifting algorithm and the continuous down-shifting algorithm are derived from approximations. It might be possible to have a better bound using some efficient data structures, since our estimate is based on a somewhat crude implementation of the algorithm.

The running time of the continuous shifting algorithm is proved to be \( O((n^2 p + np^2)(p + d)) \) and the running time of the continuous down-shifting algorithm is proved to be \( O(n^2 p^2 + np^3) \). In the next section we will investigate some computational results on the continuous down-shifting algorithm.

### 6.1 Computational Results

The purpose of this section is to test the time complexity of the continuous shifting algorithm for solving the Max-Min tree partition problem. The running time of the continuous down-shifting algorithm is shown to be \( O(n^2 p^2 + np^3) \). Now we will test this bound.

The algorithm was coded in Turbo Pascal 6 and run on an IBM 486 DX66. The running time of the algorithm is measured in seconds. The running time of the algorithm is denoted by \( t \), the number of edges of \( T \) is denoted by \( n \) and the number of cuts in \( T \) is denoted by \( p \).

Before we start the investigation of the computational results, some terminology must be clarified: In the computational results, the terms binary tree and \( m \)-ary tree will be used to denote strict binary trees and strict \( m \)-ary trees. If the height of an \( m \)-ary tree \( (m \geq 2) \) is a maximum among the heights of \( m \)-ary trees (we are assuming that those \( m \)-ary trees have same number of edges), then this tree is called the \( m \)-ary tree with maximum height. Note: To construct an \( m \)-ary tree \( T \) with maximum height, \( T \) must be linear in structure; for each level in \( T \), at most one edge which has \( m \) son-
edges. In this section, the random trees are generated in such a way, that the number of edges incident from a fork is restricted to be a number between 2 and 10. The average running time is averaged over a set of running times obtained from the trees with same number of edges and same number of cuts.

Here is a description of the experimental plan. At the start of the experiment, a number of test trees were generated. The test trees were constructed for two main purposes:

1. The test trees are used to verify the correctness of the program, where the test results indicate that the program is correct.

2. The test trees are used to determinate the effects of (i) the magnitudes of the edge-lengths of a tree or (ii) the number of forks of a tree, on the complexity of the algorithm. The results indicate that these two factors on their own, do not effect the order of the complexity of the algorithm. Hence, it was decided that it is justified to record only the number of edges of the sample tree, the number of the cuts in the sample tree and the running time of the algorithm.

The basic idea behind our experimental plan is to use an investigative technique. A large number of trees with various tree structures were generated, where we tested the complexity of the algorithm. From the running times obtained, we locate the tree structures where the algorithm yields a high order of complexity. We focused on those tree structures, and we generated more trees with many variations of those tree structures.

The first sample set consists of 750 randomly generated trees. To ensure that we do not have a biased sample (the underlying probability distribution of the trees is assumed to be uniform, so we tried to cover as much of the distribution space as possible), the following steps are taken:

1. The 750 trees are divided into 25 sets, where each subset consists of 30 trees with the same number of edges. So, a tree in this sample set has \( n \) edges, where \( n \in \{10, 20, 30, \ldots, 250\} \).

2. The maximum growth rate \( r \) of a randomly generated tree \( T \), means that the maximum number of son-edges for each edge in \( T \) is not greater than \( r \). Now, in each subset of 30 trees (with \( n \) edges), we subdivide the set into 6 sets of 5 trees, where all 5 trees have the same maximum growth rate of \( r \). In this sample set, \( r \in \{2, 4, 6, \ldots, 12\} \), where each edge of \( T \) is randomly assigned \( n_e \) son-edges. The value of \( n_e \) is determined by some probability function and the range of \( n_e \) is \( \{0, 2, 3, 4, \ldots, r\} \). For example, if a tree has a maximum growth rate of 4,
then each edge in $T$ can branch into 0 son-edge, 2 son-edges, 3 son-edges or 4 son-edges. This step makes sure that the sample trees are not biased towards any particular tree structure.

(3) All the edge-lengths of the sample set are uniformly distributed over the set \{1,2,3,...,100\}.

A further 500 trees (approximately) were generated, using similar schemes to the above. However, we varied the number of edges, the maximum growth rate, the weights of the discrete probability function, and the edge-lengths. Approximately another 100 trees were randomly generated with special structures, namely the strictly $n$-ary trees and the $n$-ary trees with maximum height.

In Chapter 5, the time complexity of the algorithm is proved to be a polynomial in $n$ and $p$ ($O(n^2p^2 + np^3)$), hence the computational results will be tested in two major sections:

- In the first section we examine the order of $n$ in the time complexity to the continuous shifting algorithm: We plotted the graph of $\log_2 t$ versus $\log_2 n$. Now, the slopes of this type of graph indicate the order of $n$.

- In the second section we examine the order of $p$ in the time complexity to the continuous shifting algorithm: We plotted the graph of $\log_2 t$ versus $\log_2 p$. Now, the slopes of this type of graph indicate the order of $p$.

In this section, the method of least squares is used to calculate the slope for the best fitting line that passes through the observations.

### 6.1.1 Time Complexity of the Algorithm as a Function of $n$

In the first section, the relationship between the number of edges and the running time is examined, by varying the number of edges of the tree with a fixed number of cuts. The algorithm is applied to series of trees and the running time is recorded.

The algorithm was applied to a set of randomly generated trees. Some of the results are given in Table 6.1. We also plotted the graph of $\log_2(t)$ vs. $\log_2(n)$. The slopes obtained from the plots of the average running time versus the number of edges are less than 1. This indicates that the time complexity of the algorithm obtained from most of random trees from this set is less than $O(n)$. In Figure 6.1, a graph of some of those plots is presented. From the graph it is not clear if we have reached a high enough $n$ to
indicate the effect of $n$ on the time complexity of the algorithm.

The algorithm was then applied to strict binary trees with maximum height, strict 3–ary trees with maximum height, and strict 4–ary trees with maximum height (see Table 6.3 and Figure 6.3). By comparing the slopes in Figure 6.2 and the slopes in Figure 6.3, the slopes are around 1 which implies that, under this type of construction, the complexity of the algorithm: Let $T_1$ be trees with

<table>
<thead>
<tr>
<th>Slope of</th>
<th>C20 to 250</th>
<th>C10 to 250</th>
<th>C100 to 250</th>
<th>Slope of</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 70$</td>
<td>$0.80631839$</td>
<td>$0.48139168$</td>
<td>$0.30421788$</td>
<td>$n = 80$</td>
</tr>
<tr>
<td>$n = 90$</td>
<td>$0.80631839$</td>
<td>$0.48139168$</td>
<td>$0.30421788$</td>
<td>$n = 100$</td>
</tr>
</tbody>
</table>

Table 6.1: Computational results for the graph in Figure 6.1.

However, the slopes seem to indicate that the bigger the ratio between the number of edges and the number of cuts, $\frac{C_2}{C_1}$, the larger the slope. Hence we used another set of randomly generated trees to test this; see Table 6.2 and the graph in Figure 6.2. The slopes are around 1 which implies that, under this type of construction, the complexity of the algorithm is $O(n)$.

The algorithm was then applied to strict binary trees with maximum height, strict 3–ary trees with maximum height, and strict 4–ary trees with maximum height (see Table 6.3 and Figure 6.3). By comparing the slopes in Figure 6.2 and the slopes in Figure 6.3, the slopes of the edges in this graph are random assigned, the major difference between the binary tree, the 3–ary tree and the 4–ary tree is that, of the trees. For a given $n$, the height of the binary tree with maximum height is greater than the height of the 3–ary tree with maximum height. Similarly, the height of the 3–ary tree with maximum height is greater than the height of the 4–ary tree with maximum height. Hence the graph indicates that increasing the height of a tree also increases the coefficient of $n$ in the time complexity of the algorithm: Let $T_1$ and $T_2$ be trees with $n$ edges. Suppose the running time of the algorithm on $T_1$ (respectively $T_2$) with $p$ cuts be $c_1p^2n^2$ (respectively $c_2p^2n^2$); $c_1$, $c_2$ are the constant coefficients. In the graph, we plot $\log$(running time) versus $\log(n)$, hence if
Figure 6.1: Graph of the average running time on some random trees.

Figure 6.2: Graph of the average running time on some randomly generated trees.
<table>
<thead>
<tr>
<th>no.of edges</th>
<th>Ave. Time</th>
<th>Sample Set1</th>
<th>Sample Set2</th>
<th>Sample Set3</th>
<th>STDev.</th>
<th>log2(n)</th>
<th>log2(AveTime)</th>
<th>Fitted Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>2300</td>
<td>18.53771083</td>
<td>1.78E+01</td>
<td>2.21E+01</td>
<td>1.98E+01</td>
<td>5.692254457</td>
<td>11.1671415</td>
<td>4.07287826</td>
<td>4.14227533</td>
</tr>
<tr>
<td>2000</td>
<td>15.58297874</td>
<td>1.58E+01</td>
<td>1.34E+01</td>
<td>1.75E+01</td>
<td>2.044046315</td>
<td>10.9927828</td>
<td>3.98189131</td>
<td>3.868646952</td>
</tr>
<tr>
<td>1727</td>
<td>12.19842550</td>
<td>7.14E+00</td>
<td>9.93E+00</td>
<td>1.86E+01</td>
<td>4.890322303</td>
<td>10.7549237</td>
<td>3.48781625</td>
<td>3.579873243</td>
</tr>
<tr>
<td>1287</td>
<td>8.234262211</td>
<td>4.27E+00</td>
<td>7.66E+00</td>
<td>1.28E+01</td>
<td>4.269667162</td>
<td>10.3406876</td>
<td>3.04305764</td>
<td>3.015465171</td>
</tr>
<tr>
<td>974</td>
<td>7.326716865</td>
<td>1.02E+01</td>
<td>2.63E+00</td>
<td>9.31E+00</td>
<td>4.568635456</td>
<td>9.29777786</td>
<td>2.67305560</td>
<td>2.541166885</td>
</tr>
<tr>
<td>732</td>
<td>3.159186427</td>
<td>4.56E+00</td>
<td>3.18E+00</td>
<td>1.74E+00</td>
<td>1.402322519</td>
<td>9.51968836</td>
<td>1.68563068</td>
<td>1.8308328296</td>
</tr>
<tr>
<td>549</td>
<td>2.454612524</td>
<td>2.14E+00</td>
<td>3.14E+00</td>
<td>2.08E+00</td>
<td>0.565661484</td>
<td>8.10492339</td>
<td>1.75469800</td>
<td>1.321656542</td>
</tr>
<tr>
<td>412</td>
<td>1.244650517</td>
<td>3.98E+00</td>
<td>1.17E+00</td>
<td>1.18E+00</td>
<td>0.121256836</td>
<td>8.66800027</td>
<td>0.31785232</td>
<td>0.759945652</td>
</tr>
<tr>
<td>310</td>
<td>1.234405817</td>
<td>1.07E+00</td>
<td>8.96E-01</td>
<td>1.78E+00</td>
<td>0.481900619</td>
<td>8.27612446</td>
<td>0.30841693</td>
<td>0.1954433184</td>
</tr>
<tr>
<td>232</td>
<td>0.869495403</td>
<td>1.59E+00</td>
<td>5.72E+00</td>
<td>5.04E+00</td>
<td>0.894117562</td>
<td>7.85768995</td>
<td>0.18692050</td>
<td>0.3759063174</td>
</tr>
<tr>
<td>174</td>
<td>0.749927209</td>
<td>7.13E+00</td>
<td>5.56E+00</td>
<td>9.74E+00</td>
<td>0.997356871</td>
<td>7.42943494</td>
<td>0.41794212</td>
<td>0.842747862</td>
</tr>
<tr>
<td>141</td>
<td>0.394667683</td>
<td>3.10E+00</td>
<td>4.62E+00</td>
<td>3.62E+00</td>
<td>0.992747607</td>
<td>7.13951323</td>
<td>0.34132221</td>
<td>0.135831432</td>
</tr>
<tr>
<td>86</td>
<td>0.43197043</td>
<td>5.10E+00</td>
<td>2.72E+00</td>
<td>5.13E+00</td>
<td>0.186719852</td>
<td>6.81707644</td>
<td>0.21186971</td>
<td>0.2795730706</td>
</tr>
<tr>
<td>37</td>
<td>0.10375695</td>
<td>1.20E+00</td>
<td>6.91E-01</td>
<td>1.27E+00</td>
<td>0.011756557</td>
<td>5.29455368</td>
<td>0.32487446</td>
<td>0.3629079206</td>
</tr>
<tr>
<td>10</td>
<td>0.04504276</td>
<td>3.05E-00</td>
<td>6.20E-02</td>
<td>4.50E-00</td>
<td>0.015759641</td>
<td>3.32102939</td>
<td>4.47198227</td>
<td>4.5707502026</td>
</tr>
</tbody>
</table>

Table 6.2: Computational results for the graph in Figure 6.2.
Figure 6.3: Graph of the running time vs. number of edges, where the algorithm approaches theoretic worst bound (proved in Chapter 5) on these trees.

Figure 6.4: Graph of $O(n^2)$ slopes.

109
Table 6.3: Computational results for the graph in Figure 6.3.
the intercept of the plot of $T_1$ is greater than the intercept of the plot of $T_2$ then $c_1 > c_2$.

Furthermore, we found trees for which the time complexity of the algorithm is $O(n^2)$, see Table 6.4 and Figure 6.4.

### 6.1.2 Time Complexity of the Algorithm as a Function of $p$

In the second section, the relationship between the number of cuts and the running time is examined, by varying the number of cuts of the tree with fixed number of edges. The algorithm is applied to series of trees and the running time is recorded.

#### Table 6.5: Computational results for the graphs in Figure 6.6.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>0.396460927</td>
<td>0.198628</td>
<td>0.0001</td>
<td>0.001</td>
<td>30</td>
<td>0.35362-2</td>
<td>9.125-03</td>
<td>0.1</td>
</tr>
<tr>
<td>20</td>
<td>0.808361267</td>
<td>0.465215</td>
<td>0.0101</td>
<td>0.0101</td>
<td>20</td>
<td>1.365-01</td>
<td>4.75-02</td>
<td>0.8</td>
</tr>
<tr>
<td>10</td>
<td>1.469703633</td>
<td>0.778321</td>
<td>0.0202</td>
<td>0.0202</td>
<td>10</td>
<td>3.266-01</td>
<td>1.256-01</td>
<td>2.7</td>
</tr>
<tr>
<td>5</td>
<td>2.160794473</td>
<td>1.103864</td>
<td>0.0303</td>
<td>0.0303</td>
<td>5</td>
<td>6.265-01</td>
<td>2.716-01</td>
<td>6.4</td>
</tr>
<tr>
<td>3</td>
<td>3.337422625</td>
<td>1.478737</td>
<td>0.0404</td>
<td>0.0404</td>
<td>3</td>
<td>1.076-01</td>
<td>5.068-01</td>
<td>12.5</td>
</tr>
<tr>
<td>2</td>
<td>4.038672073</td>
<td>1.839708</td>
<td>0.0505</td>
<td>0.0505</td>
<td>2</td>
<td>1.636-01</td>
<td>7.786-01</td>
<td>21.6</td>
</tr>
<tr>
<td>1</td>
<td>5.212204747</td>
<td>2.351688</td>
<td>0.0606</td>
<td>0.0606</td>
<td>1</td>
<td>2.416-01</td>
<td>1.266-01</td>
<td>34.3</td>
</tr>
<tr>
<td>80</td>
<td>6.469705642</td>
<td>2.563622</td>
<td>0.1111</td>
<td>0.1111</td>
<td>80</td>
<td>3.376-01</td>
<td>1.906-01</td>
<td>91.2</td>
</tr>
<tr>
<td>9</td>
<td>7.972776303</td>
<td>3.158448</td>
<td>0.1212</td>
<td>0.1212</td>
<td>9</td>
<td>4.596-01</td>
<td>2.480-01</td>
<td>72.9</td>
</tr>
<tr>
<td>100</td>
<td>9.036265335</td>
<td>3.685498</td>
<td>0.1313</td>
<td>0.1313</td>
<td>100</td>
<td>6.246-01</td>
<td>4.040-01</td>
<td>100</td>
</tr>
<tr>
<td>110</td>
<td>11.38733588</td>
<td>4.284711</td>
<td>0.2020</td>
<td>0.2020</td>
<td>110</td>
<td>8.276-01</td>
<td>5.796-01</td>
<td>133.1</td>
</tr>
<tr>
<td>120</td>
<td>13.28019727</td>
<td>4.921021</td>
<td>0.2121</td>
<td>0.2121</td>
<td>120</td>
<td>1.006-01</td>
<td>7.426-01</td>
<td>172.8</td>
</tr>
<tr>
<td>130</td>
<td>15.32044137</td>
<td>5.707944</td>
<td>0.2222</td>
<td>0.2222</td>
<td>130</td>
<td>1.260-01</td>
<td>8.205-01</td>
<td>219.7</td>
</tr>
<tr>
<td>140</td>
<td>17.26530603</td>
<td>6.516107</td>
<td>0.2323</td>
<td>0.2323</td>
<td>140</td>
<td>1.560-01</td>
<td>9.036-01</td>
<td>274.4</td>
</tr>
<tr>
<td>150</td>
<td>19.53562565</td>
<td>7.700651</td>
<td>0.2424</td>
<td>0.2424</td>
<td>150</td>
<td>1.826-01</td>
<td>1.330-01</td>
<td>337.5</td>
</tr>
<tr>
<td>160</td>
<td>21.85254251</td>
<td>8.465324</td>
<td>0.2525</td>
<td>0.2525</td>
<td>160</td>
<td>2.186-01</td>
<td>1.576-01</td>
<td>409.8</td>
</tr>
<tr>
<td>170</td>
<td>24.13448791</td>
<td>9.833737</td>
<td>0.2626</td>
<td>0.2626</td>
<td>170</td>
<td>2.726-01</td>
<td>1.876-01</td>
<td>491.3</td>
</tr>
<tr>
<td>180</td>
<td>26.75098638</td>
<td>11.245649</td>
<td>0.2727</td>
<td>0.2727</td>
<td>180</td>
<td>3.286-01</td>
<td>2.569-01</td>
<td>583.2</td>
</tr>
<tr>
<td>190</td>
<td>29.48485251</td>
<td>12.606628</td>
<td>0.2828</td>
<td>0.2828</td>
<td>190</td>
<td>3.776-01</td>
<td>2.966-01</td>
<td>685.9</td>
</tr>
<tr>
<td>200</td>
<td>32.36106735</td>
<td>14.4525</td>
<td>0.3030</td>
<td>0.3030</td>
<td>200</td>
<td>4.406-01</td>
<td>3.606-01</td>
<td>800</td>
</tr>
</tbody>
</table>

Table 6.5: Computational results for the graphs in Figure 6.6.
Figure 6.5: The graph of some maximum running time from some set of randomly generated trees with same number of edges.

Figure 6.6: The graphs of some average running time from some set of random trees with same number of edges.
Again, the algorithm was applied on a set of randomly generated trees and graphs of $\log_2(t)$ vs $\log_2(p)$ were plotted. Most of the slopes of the average running time are strictly below 3, which indicates that the time complexity of the algorithm on most of the trees is less than $O(p^3)$. We presented in Figure 6.5, a graph of the maximum running time on the randomly generated trees with same number of edges. The standard deviations of the average running time of the algorithm, executed on the random trees, are very high, which indicate that the coefficients of $p$ depended on the structure of the trees. See Table 6.5 and Figure 6.6, where we plot the average running time against the number of cuts with the error bars to indicate the variations.

Consideration of the slope of the plotted lines seems to indicate that the bigger the ratio between the number of cuts and the number of edges, $\frac{c}{e}$, the steeper the slope. The top two graphs in Figure 6.7 are of the running time versus number of cuts for the tree, selected from a set of random trees with same number of edges, with the largest running time. In Figure 6.7, the graph of the tree with 10 edges has a slope of 2.98. This implies we have found a tree for which the time complexity of the algorithm is $O(p^3)$.

![Graph of the worst time tree from 30 sample trees with 10 edges](image)

![Graph of the worst time tree from 30 sample trees with 170 edges](image)

**Figure 6.7: The graphs of some worst running time.**

We applied the algorithm on series of strict binary trees and strict 3-ary trees. We are interested in those tree structures, because for a given $n$ those tree structures have a large number of forks, which increases the number of bottleneck points during the execution of the algorithm.
Figure 6.8: Graph of the computational results of random binary and 3-ary trees.

Figure 6.9: The graph of some maximum running time from some set of randomly generated trees with same number of edges.
Table 6.6: Computational results for the graphs in Figure 6.10 and Figure 6.11.
Figure 6.10: Graph of the computational results in Table 6.6.

Figure 6.11: Graph of the computational results in Table 6.6.
The algorithm is applied on sets of randomly generated binary trees and 3-ary trees. The average running time for each set is then calculated and the results are plotted in a graph (see Figure 6.8). The slopes of these plots are less than 2.4. Since we have a tree for which the time complexity of the algorithm is $O(p^3)$ (see Figure 6.7), we will compare the computational results with respect to $p^3$. Hence the slopes in Figure 6.8 indicate that it is not sufficient just to increase the number of forks in a tree, to increase the magnitude of the order of $p$.

We then tested the algorithm on series of trees, where the height of the tree and the edge-lengths are varied (see graph in Figure 6.9 is one of such plots). The edge-length of a tree does not have significant effect on the slope of the graph, but the height of a tree does effect the slope (see Figure 6.11 and Table 6.6).

### 6.1.3 Conclusions

The computational results indicate that the $n^2$ part of the time complexity ($O(n^2p^2 + np^3)$) is correct for the implementation used. The $p^3$ part of the time complexity is correct also. However, the computation results seem to indicate that our algorithm might have a lower time complexity on most trees. We assert that the regularity hypothesis (see page 94 for more details) contributed to the lower time complexity on most trees.

Here is a summary of the computational results:

1. For a given $T$, the running time of the algorithm increases when $p$ increases.
2. Given two trees $T_1$, $T_2$ and for a fixed $p$, if $T_1$ contains more forks than $T_2$, then the running time on $T_1$ will generally be greater than the running time on $T_2$.
3. The computational results indicate that the edge-lengths of $T$ do not affect the order of $n$ nor the order of $p$.
4. The order of $n$ in the time complexity increased with respect to the height of $T$.
5. For trees with $n$ edges, the coefficients of $n$ increased with respect to the height of $T$.
6. From the computational results, the maximum order of $n$ in the complexity of the algorithm is approximately 2. Hence the computational results indicate that the $n^2$ part of the time complexity ($O(n^2p^2 + np^3)$) is correct.
7. From the computational results, the coefficients of $p$ depended on the structure
of the trees.

(8) The order of $p$ in the time complexity increased with respect to $T$ with large number of forks. The computational results indicate that trees with higher value of height increase the order of $p$ in the time complexity.

(9) The maximum order of $p$ in the complexity of the algorithm is approximately 3. The computational results indicate that the $p^3$ part of the time complexity ($O(n^2p^2 + np^3)$) is correct.
Appendix A

A Detailed Proof of The One-to-One Correspondence of $\sigma$

Here we will give the formal proof for Lemma 4.3.4 (hence Lemma 5.2.5, since they are the same).

Under the assumption that all edge-lengths are integral, the reduction of $T$ to a tree $T'$ can be performed using the reduction algorithm defined on page 57. The resulting tree $T'$ has $O(p!nl_{\text{max}})$ nodes, where $l_{\text{max}}$ is the maximum length of an edge of $T$.

Let $X_k$ be the set of those points $x$ of $T$ such that $\text{dist}(x) \in \mathbb{N}/k$, $(k = 1, 2, \ldots)$ Let $x, y$ be two points of $T$. The point $y$ is a descendant of the point $x$ if either $\text{edge}(x) = \text{edge}(y)$ and $\text{dist}(x) \geq \text{dist}(y)$ or $\text{edge}(x) \neq \text{edge}(y)$ and $\text{edge}(y)$ is a descendant of $\text{edge}(x)$.

Lemma A.0.1 There exists a one-to-one correspondence $\sigma$ between $X_{p_1}$ and the edge-set of $T'$, such that, for all $x, y \in X_{p_1}$, $y$ is a descendant of $x$ iff $\sigma(y)$ is a descendant of $\sigma(x)$ in $T'$. Furthermore, if $c_1, \ldots, c_{p-1}$ belong to $X_{p_1}$ and are the cuts of a $p$-partition of $T$, then the $p$-partition of $T'$ whose cuts are $\sigma(c_1), \ldots, \sigma(c_{p-1})$ has the following property: for $i = 1, 2, \ldots, p - 1$ the weight of the down-component of $\sigma(c_i)$ is equal to the length of the down-component of $c_i$ multiplied by $p!$. A similar property holds for the root component.

Proof: In the discrete shifting algorithm we may assume, without loss of generality, that the edge with a cut belongs to the down-component of the cut. Hence, a cut is placed at the tail of an edge.
We will define $\sigma$ recursively. Let $\sigma$ maps the root of $T$ to the root of $T'$ and the fork of $T$ to the zero weighted node of $T'$. Furthermore, if $v$ is a terminal vertex of $T$, then $\sigma(v) = \phi$.

Let $x \in X_p \cap (u, v)$, where $(u, v)$ be an edge in $T$.

If $\text{dist}(x) = \frac{1}{p!}$, then one of the following cases is true:

• If $v$ is a terminal vertex, then $\sigma(x)$ is the tail of the edge which incident to the terminal vertex of $T'$.

• If $v$ is a fork, then $\sigma(x)$ is the tail of the edge which incident to the father vertex of $\sigma(v)$.

• If $v$ is a vertex of degree 2, then $\sigma(x)$ is the tail of the father edge of $\sigma(v)$.

Now, if $\text{dist}(x) > \frac{1}{p!}$, then $\sigma(x)$ is the tail of the father edge of $\sigma(y)$, where $\text{dist}(y) = \text{dist}(x) - \frac{1}{p!}$ and $y \in (u, v)$.

It is clear that $\sigma$ satisfied the conditions stated in the lemma.
Appendix B

Some Results For Chapter 5

B.1 The Formal Description of the Algorithm

Here is the detailed formal description of the procedure. In what follows,

- \( \text{TOTAL-LENGTH} \) is the total length of the tree \( T \);
- \( DC(k) \) is the length of the down-component of the \( k \)-th cut;
- \( RDC(k) \) is the length of the resulting down-component of the \( k \)-th cut;
- \( A \) is the index-set of all active cuts;
- \( \text{speed}(h) \) is the speed of the \( h \)-th cut, \( h \in A \);
- \( \text{father}(h) \) is the index of the father of the \( h \)-th cut.

**CONTINUOUS DOWN-SHIFTING ALGORITHM**

\[
\begin{align*}
\text{begin} & \\
& k := 1; \quad c_k := r_0; \\
& DC(k) := RDC(k) := \text{LARGEST} := \text{TOPCOMP} := \text{TOTAL-LENGTH} \quad \text{SHORTEST} := 0; \\
& A := \{c_k\}; \quad \text{do repeat} \quad \text{JUMP:}
\end{align*}
\]
if LARGEST < SHORTEST then stop;
if there is some blocked cut in $A$ then
  choose some blocked cut $c_h \in A$;
  $c_h := \text{jump}(c_h)$; \{if $c_p$ is the dummy cut, then by convention $\text{jump}(c_p) = c_p$\}

ADDCUT:
  if there is no cut at $r_0$ then
    if $k \leq p - 1$ then
      $k := k + 1$;
      $c_k := r_0$;
      $DC(k) := RDC(k) := \text{TOPCOMP}$;
    endif
  endif

FAST UPDATE:
  $m := \text{father}(c_h)$;
  $DC(m) := DC(m) + DC(c_h) - RDC(c_h)$;
  updateRDC(c_h);
  if $RDC(m) = \text{LARGEST}$ then $A := A \cup \{m\}$;
  if $RDC(m) > \text{LARGEST}$ then
    $A := \{m\}$;
    $\text{LARGEST} := RDC(m)$;
  endif
  if $k \geq p$ then $\text{SHORTEST} := \min\{DC(1), \ldots, DC(k)\}$;
  \{\text{SHORTEST is 0 as long as some component is empty}\}
else \{all active cuts are unblocked\}

SLIDE:
  if LARGEST = SHORTEST then stop; \{that is $b_4 = 0$\}
else
  Find the index set $P$ of all passive cuts;
  Find the index set $N$ of all neutral cuts;
  for each $h \in A$ compute $\text{speed}(h)$ \{see algorithm for speed below\};
  compute $b_1$ using (1);
  $\text{MAXNEUTR} := \max\{RDC(h) : h \in N\}$;
  $b_2 := \text{LARGEST} - \text{MAXNEUTR}$;
  compute $b_3$ using (3),(4),(5);
  $b := \min\{b_1, b_2, b_3, b_4\}$;
  for each $h \in A$ make $c_h$ slide by $b \cdot \text{speed}(h)$;
endif

UPDATE:
for each $i = 1$ to $k$ update $DC(i)$ and $RDC(i)$;
LARGEST := $\max\{RDC(1), \ldots, RDC(k)\}$;
TOPCOMP := TOTAL LENGTH$\ldots$ $DC(1) - \ldots - DC(k)$;
if $k \geq p$ then SHORTEST := $\min\{DC(1), \ldots, DC(k)\}$;
{SHORTEST is 0 as long as some component is empty}
$A := \{h : 1 \leq h \leq k, RDC(h) = \text{LARGEST}\}$;
end if
enddo repeat
end

Algorithm for calculating the speed of a cut

for $i = 1..k$ do {\textit{k is the number of cuts that have been introduced}}
begin
speed($i$) := 1;
if $\text{son}(i) \neq \emptyset$ then
for each $c_k \in \text{son}(i)$ do
if $c_k \in A$ then speed($i$) := speed($c_i$) + speed($c_k$);
endif
endfor
end if
end
end for

Note that we introduced the cuts from 1 to $p-1$ in that order during the continuous shifting algorithm, and hence the $\textbf{for}$ loop does a bottom-up search on the cut-tree.

Notice that each iteration of the $\textbf{repeat}$ loop corresponds to a single stage. The rô\-le of the dummy cut $c_p$ is to ensure that the optimality test ($\text{SHORTEST} \geq \text{LARGEST}$) is performed when $c_p$ becomes active, that is, when the length of the top-component becomes equal to LARGEST. Consider, for example, the problem of finding an optimal $p$-partition for the trivial tree consisting of a single edge $e$. When all the $p - 1$ cuts become unblocked nothing would prevent them from being down-shifted all the way to head($e$), were it not for the presence of the dummy cut $c_p$ above the root. As soon as the lengths of the $p$ components become equal, $c_p$ becomes active and the while loop halts since the condition LARGEST = SHORTEST holds.
B.2 Complexity of the Continuous Down-Shifting Algorithm

We shall prove that the time-complexity of the continuous down-shifting algorithm is \( O(n^2p^2 + np^3) \).

A sliding stage is said to be of type \( i \) if it ends with an event of type \( i \), that is, if \( b = b_i (i = 1, \ldots, 4) \). A sliding stage is hybrid if two or more among the four bottlenecks coincide, that is, if the stage is of more than one type.

**Theorem B.2.1** The continuous down-shifting algorithm runs in \( O(n^2p^2 + np^3) \) time.

**Proof:** We shall first establish a polynomial upper bound on the number of stages.

**CLAIM 1:** The total number of jumping stages is at most \((n-1)(p-1)\).

**Proof:** Each jumping stage consists of a single jump. Since all cuts move down, no cut ever jumps over a fork more than once. Furthermore, no cut jumps over any leaf other than root. It follows that the total number of jumps (or equivalently, of jumping stages) never exceeds \((n-1)(p-1)\).

**CLAIM 2:** The total number of sliding stages is at most \((n-1)(p-1)(2p-1) + 1\).

**Proof:** Let us assume for the moment that there is no hybrid stage.

When an active cut becomes blocked, before getting blocked again it must jump over some fork. Then, in view of CLAIM 1, the number of stages of type 1 is at most \((n-1)(p-1)\). Next, observe that a stage of type 2 or 3 always results in a unit increase of the number of active cuts (all the cuts that were active at the beginning of any such stage remain active at the end of the stage). As a consequence, any stage of type 2 or 3 is followed by at most \( p - 2 \) stages of these two types: afterwards, either a jump or a stage of type 1 or 4 must necessarily take place. Therefore, the number of stages of type 2 or 3 is bounded above by \( 2(n-1)(p-1)^2 \). Finally, there is at most one stage of type 4, since right after it the algorithm halts.

The above bounds hold a fortiori if some sliding stage is hybrid. Thus the total number of sliding stages is at most \((n-1)(p-1) + 2(n-1)(p-1)^2 + 1 = (n-1)(p-1)(2p-1) + 1\).

Next, we analyze the complexity of any single stage. Our estimate is based on a
(n = no. of nodes, p = no. of components = no. of cuts + 1)

Complexity of single stage

<table>
<thead>
<tr>
<th>Computation</th>
<th>Order of complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Active</td>
<td>(O(p))</td>
</tr>
<tr>
<td>Fleets</td>
<td>(O(p))</td>
</tr>
<tr>
<td>Passive, Neutral</td>
<td>(O(p))</td>
</tr>
<tr>
<td>jump(c), (\forall) blocked active c</td>
<td>Overall (O(n))</td>
</tr>
<tr>
<td>speed (c), (\forall c \in) Active</td>
<td>Overall (O(p))</td>
</tr>
<tr>
<td>(b_1)</td>
<td>(O(p))</td>
</tr>
<tr>
<td>MAXNEUTR</td>
<td>(O(p))</td>
</tr>
<tr>
<td>(b_2, b_4)</td>
<td>(O(1))</td>
</tr>
<tr>
<td>AS(c), (\forall) unblocked passive c</td>
<td>Overall (O(p))</td>
</tr>
<tr>
<td>AS(c)(g), (\forall) blocked passive c, (\forall g \in) Son(edge(c))</td>
<td>Overall (O(n))</td>
</tr>
<tr>
<td>(\beta_c, \forall c \in) Passive</td>
<td>Overall (O(p))</td>
</tr>
<tr>
<td>(b_3)</td>
<td>(O(p))</td>
</tr>
<tr>
<td>(b)</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Sliding</td>
<td>(O(p))</td>
</tr>
<tr>
<td>Cut-tree</td>
<td>(O(n))</td>
</tr>
<tr>
<td>(h(e), e \in E)</td>
<td>overall (O(n))</td>
</tr>
<tr>
<td>DC(i), (\forall i)</td>
<td>overall (O(p))</td>
</tr>
<tr>
<td>RDC(i), (\forall i)</td>
<td>overall (O(n))</td>
</tr>
<tr>
<td>LARGEST, TOPCOMP, SHORTEST</td>
<td>(O(p))</td>
</tr>
</tbody>
</table>

Overall complexity of a single stage \(O(n + p)\)

No. of stages \(O(np^2)\)

Overall complexity of the algorithm \(O(n^2p^2 + np^3)\)

Table B.1: COMPLEXITY OF THE CONTINUOUS DOWN-SHIFTING ALGORITHM

somewhat crude implementation of the algorithm (in particular, we assume that all relevant values are computed from scratch in each stage, instead of being updated from the previous stage). Thus, it is not unlikely that a better bound can be achieved. On the other hand, our primary purpose here is to establish the polynomial complexity of the algorithm rather than looking after the details of an efficient implementation.

Table B.1 summarizes the order of complexity of the different computations required by the algorithm. It turns out that the running time of the continuous shifting algorithm is \(O(n^2p^2 + np^3)\). Hence the algorithm is strongly polynomial.

---

125
Bibliography


127


