INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Fun With Cones

Implementation of an Approximation Algorithm for Computing Sums of Euclidean Distances

By
Jillian Faye Hockey

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

25 September 2001

© Copyright
2001, Jillian Faye Hockey

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-67063-5

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

Fun With Cones
Implementation of an Approximation Algorithm for
Computing Sums of Euclidean Distances

submitted by

Jillian Faye Hockey

Dr. Frank Dehne
(Director, School of Computer Science)

Dr. Prosenjit Bose
(Thesis Supervisor)

Carleton University
25 September 2001

Abstract

This thesis considers an approximation scheme presented by Bose *et al* [BMM01] for computing the sum of Euclidean distances $w(p)$ from a point p to all points in a set S of n points. The approximation computes $w(p)$ to within a factor $1/\cos \frac{\theta}{2}$, where θ is an arbitrarily small constant dependent on a constant k . The set S is preprocessed in $O(kn \lg n)$ time and space and, depending on the data structure used, computes the sum in $O(k \lg n)$ or $O(k \lg^2 n)$ time.

As k increases, the approximation error decreases; for example, the approximation is found to overestimate the Euclidean sum by no more than 9% for $k \geq 8$ and by no more than 3% for $k \geq 13$.

This thesis uses experimentation and statistical analysis to characterize how the implemented approximation scheme performs in practice, as compared with the expected theoretical behaviour.

The application of this preprocessing technique to facility location problems, specifically the Weber problem, is discussed.

Acknowledgements

I have somehow managed to spend (way too many) years working towards a Master's degree. Finally, after considering several thesis topics in different areas, I am finally finished. Over the years, I have been fortunate to have the support of family, friends, and others I have met on the path to achieving this goal. How can I individually thank each person who has shown their support without forgetting anyone? The list is longer than this thesis.

To my supervisor, Jit Bose, thank you for putting faith in me to finish, especially at those times when even I wasn't so convinced I would.

When I was frustrated with segmentation faults, writing, and statistics, I didn't have to look far to find the support of my colleagues. Special thanks to Steffen Christensen and to Jason Morrison for not only providing me with advice on my work, but as well for contributing to many sanity breaks. Thanks to those who turned a blind eye or joined in on the fun during those breaks. Don't deny this, Darrell!

Thanks to the many other friends I have made who have helped me, one way or another, in my struggle to complete my Master's thesis. Many thanks to Nadine, Marian, Nicki, Terry, Tracey, and those I've played field hockey with for being my estrogen-friendly support network. Thanks to all the guys too!

Importantly, thanks to my family; to my parents for being proud when I was not, to my sisters for daring to live with me, and to Paul, because you are family. Finally, thanks to those I haven't mentioned by name, for your understanding.

It need not matter how many times people ask you the dreaded question, "what next," after hearing, "when will you finish," for so long. The worst part would be if no one asked! - Mom

Contents

Acknowledgements	i
1 Introduction to Facility Location	1
2 Summary of the Thesis	8
2.1 Chapter 1	8
2.2 Chapter 3	9
2.3 Chapter 4	9
2.4 Chapter 5	10
2.5 Chapter 6	10
2.6 Chapter 7	11
2.7 Summary	11
3 Definitions and Preliminaries	13
3.1 Geometric Preliminaries	14
3.1.1 Affine Transformations	14
3.2 Abstraction and Program Development	16
3.3 Analysing Algorithms	17
3.3.1 Model of Computation	17
3.3.2 What to Analyse	19
3.3.3 <i>Units</i> of Measure: Growth of Functions	19
3.4 The Tree: An Abstract Data Type	21
3.4.1 Binary Search Tree	22

4	Approximation of $w(p)$	24
4.1	Simplicial Cones	25
4.2	The d_k Distance Function	26
4.3	Fast evaluation of $w(p)$ using d_k distances	27
4.4	Approximation Bounds	29
5	Geometric Searching	33
5.1	Range Searching	34
5.2	Data Structures for Geometric Range Searching	36
5.2.1	Range Trees	37
5.2.2	Persistent Trees	41
6	Implementation of Range Search Data Structures	46
6.1	Software Libraries and Reuse	47
6.2	The LEDA Experience	49
6.2.1	Augmenting the LEDA Range Tree	50
6.2.2	Abandoning LEDA	57
6.3	Building from Scratch: Preliminaries	58
6.3.1	k -Oriented Coordinate System	58
6.3.2	Red-Black Tree	60
6.3.3	Extendable Implementation of a Red-Black Tree	61
6.4	Persistent Search Tree	65
6.4.1	Extending the Persistent Data Structure for our Needs	70
6.5	Range Tree	73
7	Implementation and Testing of $w_k(p)$	75
7.1	Experiments and Analysis	77
7.2	Benchmark Program: Computing the Weight	80
7.3	Generating Random Point Sets	82
7.4	Automated Testing	86
7.5	Analysis of the Test Output	88
7.5.1	Overview of the Analysis of Variance (ANOVA)	89
7.5.2	Fitting a Curve	90

7.5.3	Accuracy of the Approximated Weight	93
7.5.4	Significance of the Effect of Point Distribution on Performance	99
7.5.5	Observed vs. Theoretical Creation Time	102
7.5.6	Observed vs. Theoretical Query Time	103
7.5.7	Cost-Benefit Analysis	109
7.6	Summary Remark	112
8	Summary and Conclusions	113
8.1	Summary of the Implementation	113
8.2	Summary of the Experimental Results	114
8.2.1	Future Work	116
	Bibliography	117

List of Tables

4.1	Worst case approximation error ϵ as a function of k	32
6.1	Class <code>coordinateSystem</code> public interface	60
6.2	Class <code>rbtree</code> protected interface	65
6.3	Class <code>rbtree</code> public interface	66
6.4	Class <code>perstree</code> public interface	70
6.5	class <code>mymap</code> public interface	72
6.6	Two-key persistent search tree: class <code>aperstree</code> public interface . . .	73
7.1	Variables for Weight Computation	78
7.2	Benchmark: Class <code>testSum</code> public interface	81
7.3	Class <code>pointGenerator</code> public interface	85
7.4	Description of test parameters	87
7.5	Parameters for generating source code	88
7.6	Summary table of Analysis of Variance (Example)	90
7.7	Summary of Accuracy Analysis for different nk -pairs (Range Tree) . .	97
7.8	Summary of Accuracy Analysis for different nk -pairs	99
7.9	Summary of ANOVAs on nk pairs for significance of point distribution	100
7.10	Summary of observed creation and query times over different k and n	108
7.11	Summary of cost-benefit results for $k = 5$ and $k = 7$	111

List of Figures

4.1	A set of k simplicial cones C in $\mathbb{R}^2$	25
4.2	a. k -oriented coordinates b. k -oriented distance	26
4.3	Isosimilarity contours for l_1 -norm and k -oriented distance	26
4.4	Euclidean distance $d(x, y)$ versus approximated distance $d_k(x, y)$. . .	29
4.5	Approximation constant versus number of cones	32
5.1	One- and Two-Dimensional Range Search	35
5.2	One-dimensional Range Tree Query	36
5.3	2-dimensional Range Tree	38
5.4	Recursion tree on size of $\mathcal{T}$ plus $\mathcal{T}_{assoc}(v)$ for all v	40
5.5	Path-copy in a Persistent Tree	42
5.6	Persistent Tree and Query on Small Point Set	43
6.1	k -oriented Dominance Queries	47
6.2	LEDA range_tree Hierarchy	51
6.3	Cartesian to k -oriented coordinate transformation	54
6.4	Simple internal node augmentation: size of subtrees	55
6.5	Introducing auxiliary information into LEDA binary tree nodes . . .	56
6.6	Rebalancing the Red-Black Tree: Rotation	61
6.7	RB-Tree Pointer Changes: Attach, Splice, Replace	64
6.8	Rootvec: vector of tree-node pointers to successive roots	67
6.9	Two consecutive versions of a persistent search tree	67
6.10	Persistent tree supporting dominance queries on points	72
7.1	Original Data for Example Curve Fitting	91

7.2	Least-Squares Fit of Data for Example Curve Fitting	92
7.3	(Continuous) Model of (Discrete) Data for Example Curve Fitting . .	92
7.4	Fitting a Curve for an Asymptotic Expression	93
7.5	Sample Approximation Error for 9 Cones, Range Tree	94
7.6	Normal Probability Plot ($k = 9$, $n = 512$, Range Tree)	95
7.7	Normal Probability Plot ($k = 9$, $n = 32768$, Range Tree)	96
7.8	Persistent Tree Approximation Error Outliers for $k = 5$	98
7.9	Observed Creation Times ($k = 9$)	103
7.10	Observed vs. Theoretical Creation Times ($k = 9$)	104
7.11	Observed Average Query Times ($k = 9$)	105
7.12	Observed Average Query Times ($k = 9$)	106
7.13	Observed Average Query Time vs. Theoretical ($k = 9$)	106
7.14	Plots of the cost-benefit analysis ($k = 9$)	110
7.15	Projection of the cost-benefit analysis ($k = 9$)	111

Chapter 1

Introduction to Facility Location

Facility location problems are concerned with determining the optimal placement of a facility within a spatial context. The general problem has wide applicability and is not restricted to just one or two disciplines. For example, city planning involves choosing where to locate factories, schools, hospitals, or landfills within a community; industrial design often focuses on placing departments and machines within a working environment; and, computer hardware design considers the design and planning of computer circuit-boards and various kinds of computer networks.

The facility location problem is characterized by the nature of the real-world situation being modelled. Generally, a specific problem falls into one of two categories of facility location problems:

Standard or *desirable* where closeness to the facility is favored (*e.g.*, deciding where to locate a hospital or a school).

Obnoxious or *undesirable* where closeness to the facility is avoided (*e.g.*, choosing a location for a smokestack or a landfill).

From a mathematical perspective, a facility location problem is an optimization problem, where *optimization* denotes finding the minimum or maximum of a function. We call the function that we are optimizing the *objective function* of the problem. The objective function can define one or many types of parameters to be optimized; for instance, cost, time, and density. In computational geometry, the area of focus in this thesis, the objective function is often a distance function, expressed in terms of

the other locations (called *sites*) within the environment where the facility is to be located. The optimization rule for the objective function is defined by the desirable or undesirable nature of the facility to be located. The objective function will be minimized or maximized, respectively.

Also prominent in defining the optimization problem is the rule for considering the proximity relationship between the sites and the facility, as well as how the distances are measured. Two common rules include optimizing the overall sum of distances (equivalently, the average distance) from the sites to the facility, and optimizing the minimum or maximum site-to-facility distance. These approaches lead to four common classes of facility location problems, identifiable by their objective functions as follows:

minsum used for desirable facility placement when the overall distance from all sites to the facility is to be minimized. This kind of objective function might be used when distance travelled is a consideration.

minmax used for desirable facility placement when the goal is to have the farthest site as close to the facility as possible (*i.e.*, minimizing the furthest distance). This kind of objective function is used when the furthest distance travelled is a concern, such as when choosing a location for a fire station.

maxsum used for undesirable facility placement when the overall distance from all sites to the facility is to be maximized. This kind of objective function represents a situation where the overall cost of exposure of a population to a contaminant is a concern, such as when locating a chemical processing plant.

maxmin used in undesirable situations when the goal is to have the closest site as far from the facility as possible (*i.e.*, maximizing the closest distance). This kind of objective function is suited to the *NIMBY* (Not In My Back Yard) situation, as for municipal dump site location and fission reactor siting.

Other factors that add complexity to the facility location problem include the number of facilities to be located and whether the placement of the facility is restricted to a specific area. The former identifies the problem as *single-* or *multi-* facility, while

the latter qualifies the problem as *unconstrained* or *constrained* (the facility must lie within a specified boundary).

Historically, facility location decisions have been based on numerous factors, including economic, social, political, aesthetic and physical. This large set of factors obscures the existence of quantitative tools to help the decision-makers.

The field of computational geometry emerged in the late 1970s as an area of computer science that focused on solving geometric abstractions of realistic situations using a computer [O'R94]. Provided that a problem can be expressed in terms of its geometric properties, applying appropriate algorithmic techniques and using the correct data structures can efficiently solve the problem at hand. Computational geometry techniques focus on identifying the geometric aspects of a problem and on implementing an efficient algorithm to solve that problem.

Even though a large base of efficient geometric algorithms have been developed in the short history of computational geometry, the process from problem formulation to efficient, correct implementation can still present challenges. Extracting and understanding the fundamental geometric properties of a problem is the first difficulty. Once the geometric problem is understood and a general algorithm is developed, there are still many details to consider. Such details include how special or degenerate cases should best be handled and what measures should be taken to overcome computational imprecision in the algorithm. In general, geometry deals with exact arithmetic, while computers, being finite machines, do not. Therefore, specific measures must often be taken to simulate exact arithmetic in the implementation.

In this thesis, we restrict our attention to the geometric properties of the facility location problem. Both the sites and the facilities we consider are represented as points in the 2-dimensional Euclidean plane.¹ Each point p in this plane has coordinates denoted as (x, y) -tuples, or more generally, as the 0^{th} and 1^{st} components of the point p .²

As the geometric properties of the facility location problem are identified, the

¹In order to perform spatial associations or determine spatial relationships, entities must be referenced in a common coordinate system. The most common sense of the term coordinate system assumes measurement within a grid defined by axes that meet at right angles, and unit scales that form parallel, continuous fields across a plane.

²We use the computational convention of using 0-offset array indices.

description of the problem moves toward an algebraic definition. The measure of site-to-facility distance transforms from an abstract idea to a measure of distance between two points in the plane, denoted $d(p_1, p_2)$. This idea is further refined in that $d(p_1, p_2)$ is taken as any **metric**. By this we mean d is a real-valued function defined for pairs of points in a set S , such that the following properties are met for all p, q, r in S :

- i. $d(p, p) = 0$ (reflexivity)
- ii. $d(p, q) > 0$ if $p \neq q$ (positivity)
- iii. $d(p, q) = d(q, p)$ (symmetry)
- iv. $d(p, q) \leq d(p, s) + d(s, q)$ (triangle inequality)

Definition 1.1 *The Minkowski r -metric,³ denoted d_{jk} , computes the distance between two points j and k in d -dimensional space as:*

$$d_{jk} = \left[\sum_d (|x_j - x_k|)^r \right]^{1/r}$$

Some common metrics for facility location are special cases of the Minkowski metric:

Rectangular ($r = 1$)⁴ The distance between two points, p_1 and p_2 , is the length of the shortest path from p_1 to p_2 traveling only in distances parallel to the axes of the coordinate system in which the points are defined. This metric is often used for the measure of distances within a city, as roads are generally set up in a grid.

Euclidean ($r = 2$) The distance between two points, p_1 and p_2 , is the length of the *straight-line* path from p_1 to p_2 . This metric is ideal for measuring distances that do not have physical obstacles.

³Also called the l_p -norm. Both the Minkowski r -metric and the l_p norm are equivalent for $r = p$.

⁴Also called *Manhattan* or *City block distance* [FW74]

Supremum ($r = \infty$) The distance between two points, p_1 and p_2 , is the maximum difference of their respective coordinates.

Squared Euclidean Although not a Minkowski metric, the distance between two points, in this case p_1 and p_2 , is the square of the Euclidean distance between p_1 and p_2 . This metric might be used when the site-to-facility distance carries a cost that uniformly increases quadratically.

Definition 1.2 *The weight of a point p with respect to a set $S = \{p_1, \dots, p_n\}$, denoted $w(p)$ is:*

$$w(p) = \sum_{i=1}^n d(p, p_i) \quad (1.1)$$

This thesis focuses on the unconstrained minsum single-facility location problem using a Euclidean metric. The optimization problem is formulated algebraically as follows:

Problem 1.1 *Locate or place a point or facility, p_{opt} , with respect to a surrounding set $S = \{p_1, \dots, p_n\}$ of demand points (sites) in the plane in such a way that the objective function is minimized over all points in the plane:*

$$w(p_{opt}) = \min_{p \in \mathbb{R}^2} \sum_{i=1}^n d(p, p_i)$$

where $d(x, y)$ is the Euclidean distance between points x and y .

Problem 1.1 is known as the *Weber problem*.⁵

Bajaj [Baj88] shows that the Weber problem in general is not solvable over the field of rationals $\mathcal{Q}$ for $n \geq 5$; that is, the optimization problem is a nonlinear polynomial equation that is irreducible over $\mathcal{Q}$. There exists no rational algorithm for solving (i.e., finding the roots of) non-linear equations of degree greater than 4, leaving only approximation methods as a means for obtaining the optimum solution; therefore p_{opt} may not be computed exactly.⁶ In many irrational root-finding problems, the solution

⁵Also identified with the General Fermat Problem problem [Kuh73, FW74], the generalized Weber problem [Baj88], and the Fermat-Weber location problem [CT90].

⁶Even if the root of the algebraic equation could be computed exactly, numbers on a computer can only be stored and output in finite precision. Radicals in their exact form (i.e., $\sqrt[n]{m}$), are not normally understood by the computer.

proceeds by iterations, starting from some approximate solution and iterating until some predetermined convergence criterion is satisfied [PTVF92].

Since we will be considering point sets containing more than five demand points, an iterative approach is needed to solve the Weber problem. This requires that the Euclidean weight function (see Definition 1.2) be computed many times. (1.1) can be evaluated with a *brute force* approach in $O(n)$ time and space by simply computing $d(p, p_i)$ individually for all $p_i \in S$ and summing these values. The evaluation of $w(p)$ is the bottleneck in solving Problem 1.1. The objective function is not separable (see Definition 1.3) under the Euclidean metric; therefore, each candidate for p_{opt} requires that n evaluations of the distance metric be computed (in the naïve approach).

Definition 1.3 *A function in n variables, $f = f(x_1, x_2, \dots, x_n)$ is separable iff it can be formed as the sum of n univariate functions $f_i(x_i)$ (i.e., $f = \sum_{i=1}^n f_i(x_i)$).*

The only known way to evaluate the weight function (1.1) in sub-linear time is to use an approximation scheme. Bose *et al* [BMM01] propose an approximation scheme for evaluating the sum of Euclidean distances and apply it to the Weber problem. They cite Bajaj in supporting the application of their approximation scheme, emphasizing that the value for p_{opt} may not be computable exactly. Bose *et al*'s scheme approximates (1.1) to within a factor of $1/\cos \frac{\theta}{2}$ in $O(k \log n)$ time, where θ is an arbitrarily small constant and k is a constant dependent on θ . This is done by preprocessing the points in S in $O(kn \log n)$ time and space.

The goal of this thesis is to implement the approximation scheme for summing Euclidean distances using different data structures. The different implementations are compared against each other and against the brute-force approach with respect to the preprocessing time, the computation time of (1.1), and the storage required in memory to store the data structure. Also included in the discussion are observations on the implementation process. Much of the theory and background is presented for the general case; however, the implementation and examples focus on the two-dimensional case.

In Chapter 2, we present a chapter by chapter summary of the thesis. Chapter 3 gives background knowledge relevant to the Euclidean sum approximation scheme. In Chapter 4, the Euclidean distance approximation algorithm is presented. Chapter 6

focuses on the implementations of the data structures required for the approximation scheme. Chapter 7 presents the implementation of the Euclidean sum approximation scheme, compares the range search data structures experimentally and statistically, and attempts to characterize their behaviour in computing the Euclidean sum. Finally, a summary of the experimental results and conclusions are found in Chapter 8.

Chapter 2

Summary of the Thesis

This chapter presents a chapter by chapter summary of the thesis.

2.1 Chapter 1

In Chapter 1 we introduce general facility location problems as optimization problems. Two categories of facility location problems are defined, desirable and undesirable, which depend on the type of facility that is being located. The desirable or undesirable nature of a problem is identified by the optimization rule on the proximity relationship being modelled. Minsum and minmax are desirable optimization rules, while maxsum and maxmin criteria characterize undesirable optimizations. A problem may also consider locating more than one facility and we distinguish between single-facility and multi-facility location problems.

Facility location is studied in different fields, including operations research, urban planning, and computational geometry. In this thesis, we approach the problem from a computational geometry perspective and focus on the single-facility minsum problem, or Weber problem. The paper by Bose *et al* [BMM01] describes an approximation scheme for evaluating the sum of Euclidean distances (or the weight of a point) with application to the Weber problem. Their research is motivated by Bajaj's paper [Baj88], which shows that the Weber problem in general is not solvable over the field of rationals $\mathcal{Q}$ for $n \geq 5$.

2.2 Chapter 3

Chapter 3 presents general background information for the reader. The information includes a description of the basic geometric concepts used in this thesis, including points, coordinate systems, and transformations on geometric objects. This is followed by a review of general concepts relating to algorithm design and analysis, including data structures terminology, model of computation, and growth of functions. The chapter concludes with an overview of the tree abstract data type, and specifically, the binary search tree. The reader who is familiar with these notions is encouraged to skip Chapter 3.

2.3 Chapter 4

Bose *et al*'s approximation scheme, introduced in Chapter 1, is discussed in more detail in this chapter. The chapter begins by describing concepts that are used in approximating the weight of a point. These include: simplicial cones, k -oriented distances, and dominance queries.

The k -oriented distance between two points is similar to the rectangular metric, but sheared in the x -direction by $2\pi/k$ radians. It is not itself a metric since symmetry cannot be ensured. The notable aspect of the k -oriented distance measure is its separability, which allows points in a set to be preprocessed in order to approximate the weight of a point with respect to the point set in sublinear time.

Bose *et al*'s approximation scheme is presented in detail, and outlines how the points in a set of points are preprocessed in order to then answer the weight of a point. The weight of a point is computed by performing k dominance queries on the data structure holding the preprocessed information.

Since an approximation is not good unless it computes a value to within a certain threshold of the true value, the chapter concludes with a proof of the theoretical bounds on the approximation constant.

2.4 Chapter 5

Bose *et al*'s approximation scheme requires a data structure that supports insertion, deletion and dominance counting queries of two-dimensional point data. Chapter 5 provides an overview of geometric searching; in particular, range searching. The range tree and the persistent search tree are two data structures that are used for geometric range searching and are both described in detail.

The range tree supports queries in $O(\log^2 n)$ time and requires $O(n \log n)$ space, while the persistent tree supports queries in $O(\log n)$ time and $O(n \log n)$ space. These data structures allow Bose *et al*'s approximation for the weight of a point with respect to a set to be supported in $O(k \log^2 n)$ time and preprocessing in $O(kn \log n)$ time and space.

Again, the reader who is familiar with geometric searching is encouraged to skip the chapter.

2.5 Chapter 6

Chapter 6 presents an account of implementing the range search data structures (*i.e.*, the range tree and the persistent search tree). Two alternate approaches are explored for developing the implementations: use existing implementations of the foundation data structures and adapt them to our needs; or, build the data structures from scratch.

The geometric primitives of the LEDA library are used, along with the LEDA range tree data type implementation, in an attempt to reuse existing software to efficiently and easily develop the range tree data structure for the approximation scheme. The implementation is abandoned after some disappointing results.

The persistent search tree and the range tree are implemented from scratch, but not without borrowing from an existing implementation of a red-black binary search tree [Mor], the foundation data type for both range search data structures. The red-black tree is presented in detail, highlighting differences between this implementation and the standard implementation. The description is followed with a detailed description of developing the persistent search tree using the philosophy and code from the

basic red-black tree implementation. The chapter concludes with a brief description of the range tree implementation, built from scratch, which has many similarities to the persistent tree implementation.

2.6 Chapter 7

Chapter 7 concentrates on comparing the implementations of the range search data structures, focusing on their effectiveness and efficiency in approximating $w(p)$. Specifically, we attempt to characterize the implementations by answering questions such as:

1. What effect does k have on the accuracy of the approximated weight function?
2. Is the point distribution of S a significant factor contributing to the measured response time for a query?
3. Are the observed creation times and average query times proportional to the theoretical creation and query times?

The analysis necessitates the use of careful experimentation and statistical techniques, so the chapter begins with an overview of the statistics background for the analysis. A description of the implementation of the approximation scheme follows, along with details of how the experiments are created and run. The chapter ends with detailed analyses of the data.

2.7 Summary

Chapter 8 reviews the process of implementing the range search data structures for the Euclidean sum approximation algorithm and presents the results of the experiments performed on the data structures for characterizing their performance in practice. The experimental results are presented in the form of answers to the questions posed in Chapter 7.

In short, we conclude that a larger number of cones used in the approximation data structure results in a more accurate approximation of the Euclidean distance

(compared to the computed *true* value). The approximation error in practice is less than the bound on the theoretical worst-case, which is itself a true epsilon approximation. As k increases, the approximation error decreases; for example, in theory the approximation is found to overestimate the Euclidean sum by no more than 9% for $k \geq 8$ and by no more than 3% for $k \geq 13$. When looking at uniformly or normally distributed points, we find the overestimate to be roughly two-thirds that of the predicted maximal error.

The observed preprocessing (or creation) and query times of the approximated Euclidean sum are proportional to their theoretical counterparts. The cost of preprocessing can be regained after a number of queries are performed on the same point set since both range search data structures exhibit significantly smaller query times (in computing the weight of a point) than the naïve approach to computing the sum.

The persistent search tree appears to exhibit faster query times than the range tree data structure; however, the creation time of the persistent search tree appears much greater than the creation time of the range tree. Consequently, the number of queries to regain the cost of using the persistent tree is greater than the number of queries needed to perform in order to regain the cost of using the range tree.

A number of summary tables (Tables 7.7, 7.8, 7.10, and 7.11) presented in Chapter 7 might help the user to decide on the number of cones to use if the approximation scheme is used in practice. A number of factors must be considered in order to apply the approximation scheme in practice, including:

1. how much approximation error will be tolerated?
2. what are the limitations on the available memory for storing the approximation data structure?
3. how many queries will be performed on the same point set?

Chapter 3

Definitions and Preliminaries

An investment in knowledge pays the best interest. - Benjamin Franklin

For completeness, we present a description of the basic geometric concepts used in this thesis followed by a review of the basic concepts relating to algorithm design and analysis. We refer the reader to [CLR90, AU92, PS85, Wei92] for a more in-depth treatment of the topics covered in this chapter. The reader who is familiar with these basic notions should skip this chapter.

Computer science is a science of abstraction. We use the tools of this science to represent and manipulate models of real-world problems inside a computer, often allowing us to answer a question or find a solution to a given problem. A large part of this thesis focuses on evaluating, implementing, and analysing the approximation scheme for computing the weight $w(p)$ of a point (*i.e.*, the sum of all distances from the point p to all points in a given set) proposed by Bose *et al*; that is, we treat the approximation to (1.1) as a programming problem. In this chapter, we review the notation, design concepts and analysis techniques applied to implementing the proposed solution.

3.1 Geometric Preliminaries

This section reviews the basic geometric primitives used in this thesis with the objective of introducing the adopted notation. It is not included as an exhaustive overview of geometric concepts and we refer the reader who requires more background knowledge in geometry to any introductory textbook on geometry.

The predominant geometric objects considered in this thesis are points in Euclidean space. Formally, a **point** p is a d -tuple of real numbers and is generally represented as a vector of d coordinates, $p = \{x_0, x_1, \dots, x_{d-1}\}$, in the reference coordinate system. It is the convention in computer science to index the coordinates starting at 0, considering the index as an element offset into the vector. Individual coordinates may be referenced by their offset into the vector as follows: $p[0]$ for the first coordinate of point p , $p[1]$ for the second coordinate, and so on

Points in Euclidean space are referenced within a **Cartesian coordinate system**; that is, a system that assumes measurement within a grid defined by axes that meet at right angles and by unit scales that form parallel, continuous fields across the plane in each dimension. For points in 2-dimensional Euclidean space, x and y are often used to denote $p[0]$ and $p[1]$, respectively, and distances are computed with the Euclidean metric (described in Chapter 1, page 4).

Not all coordinate systems have orthogonal reference axes, as we will see in Chapter 4, but translation from the Cartesian system to another is easily achieved through algebraic operations.

3.1.1 Affine Transformations

Affine transformations are algebraic operations performed on points and objects defined by sets of points. Affine transformations are line preserving, which means straight lines are carried into straight lines and parallel lines into parallel lines, but the distance between points and angles between lines may be altered. There are four basic affine transformation operations:

Translation shifts all points on an object by an equal amount.

Rotation rotates a point about the origin in such a way that the point remains a constant distance from the origin.

Scaling stretches or shrinks an object in the direction of one or both of the coordinate axes, not necessarily by the same factor. Scaling by a negative factor results in a reflection about the corresponding axis.

Shear a uniform stretching applied to all points on an object, causing any two points to slide relatively to each other in a direction parallel to the direction of the shear.

Algebraically, these transformations are generally expressed as matrix operations operating on vectors of Cartesian coordinates of the appropriate dimension. The four basic transformations are used alone or in combination to describe all possible affine transformations. For example, other affine transformations formed by combining the aforementioned basic operations includes rotating an object about a point other than the origin or reflecting an object about an arbitrary line. We term the result of transforming a point p the *image of p* and denote it by p' .

We will see that the two-dimensional rotation and shear transformations are sufficient for this thesis. The general two-dimensional rotation matrix for rotating a point p by an angle of θ about the origin, denoted $R(\theta)$, is $\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$. The two-dimensional shear matrix for a shear of angle ϕ in the x -direction, denoted $H_x(\phi)$, is $\begin{bmatrix} 1 & -\cot \phi \\ 0 & 1 \end{bmatrix}$. Similarly, $H_y(\phi)$ is $\begin{bmatrix} 1 & 0 \\ \cot \phi & 1 \end{bmatrix}$.

Translating a set of points from one coordinate system or frame of reference to another is achieved by identifying the affine transformations for transforming the coordinate axes, then applying those transformations to all points defining geometric objects in the plane. For example, the translations between coordinate frames used in this thesis are achieved by shearing the plane by θ degrees and rotating the result by a multiple k of θ . For a given point p , we find its image under transformation as follows:

$$\begin{aligned}
 p' &= R(k\theta) \cdot H_x(\theta) \cdot p \\
 &= \begin{bmatrix} \cos k\theta & -\sin k\theta \\ \sin k\theta & \cos k\theta \end{bmatrix} \begin{bmatrix} 1 & -\cot \theta \\ 0 & 1 \end{bmatrix} p \\
 &= \begin{bmatrix} \cos k\theta & \cos k\theta \cdot \cot \theta - \sin k\theta \\ \sin k\theta & \sin k\theta \cdot \cot \theta + \cos k\theta \end{bmatrix} p
 \end{aligned} \tag{3.1}$$

In the two-dimensional case, once the points are translated, they are operated on as if the new reference frame has origin situated at $(0,0)$ and axes in the x - and y -directions.

3.2 Abstraction and Program Development

The process of program development involves making an abstraction or model of a real-world situation and implementing that abstraction. Often, identifying a good abstraction can be difficult because only details that can be accurately specified for or represented by a computer can be modelled.

An **abstraction**, or **data model**, normally has two aspects [AU92]:

State The values that data objects can assume. These are also called the static aspects of the data model.

Behaviour The operations on the data; that is, the ways in which values of objects can be changed (existing objects) or created (new objects). These are also called the dynamic aspects of the data model.

A **programming language**, such as C++, is an abstraction of mathematical concepts. It provides elementary data objects, like integers, real (floating-point) numbers, characters, arrays, and pointers, and provides appropriate operations (*i.e.*, arithmetic and access) for manipulating these objects. In the context of programming, we refer to models of data as **abstract data types (ADTs)**. ADTs, such as lists, trees, and graphs, are not represented by the programming language. When a programming language lacks a built-in representation for an abstract data type, it is up to the programmer to develop an appropriate construct, or **data structure**, to represent those data models internally to the programming language.¹

When “abstracting away” the details of the real-world situation being modelled, the programmer must identify which details of the problem are important to include in the program model (*i.e.*, the static aspects of the problem). The programmer must also identify the dynamic aspects of the problem; that is, the operations that

¹The terms *abstract data type* and *data structure* are often used interchangeably in much of software engineering.

will be used to obtain a solution by manipulating the data models or associated data structures. An **algorithm** is a precise description of the sequence of steps or operations that can be carried out on the data in order to obtain a solution to the problem.

From problem formulation to working implementation, the development of software typically spans many phases. In the most general sense, solving a programming problem involves designing an algorithm to solve the problem, implementing the algorithm in some programming language, compiling and running the program, then analysing the program and its output. For large projects, these steps may be repeated a number of times on different components of the solution, or on variants of the solution.

3.3 Analysing Algorithms

A number of different candidate algorithms might be presented as possible solutions to a given programming problem. Ideally, we would like to consider the performance of these algorithms before starting the implementation in order to identify potentially superior solutions, or to discard those that are potentially inferior. These candidate algorithms are analysed and subsequently compared against each other under the same model of computation (§3.3.1).

3.3.1 Model of Computation

A model of computation² is a formal framework for analysing algorithms in terms of their cost (with respect to computer resources). For a given problem, the cost is generally expressed as a function in terms of the size of an instance being solved. Preparata and Shamos [PS85] define a model of computation as an “(abstraction of a computer that) specifies the primitive operations that may be executed and their respective costs.” The primitive operations of a model of computation are those for which a fixed cost is charged. It assumes, for example, that it takes one unit of time to perform any primitive operation. This is in contrast to an actual computer, for

²Also called a *machine model*

which the processor time taken is generally operation-dependent.

Many models of computation exist, differing in allowed operations and in the costs of those operations. In analysing an algorithm, an appropriate model is chosen to reflect the essential features of the computational devices being used. Similarly, the computational devices being used must be able to reasonably represent as the fundamental objects being operated on. For example, in computational geometry, points are considered fundamental objects [PS85]. Therefore, for a set S of n points, we require that performing an operation once on each point in S has a cost that is proportional to n .

Geometric calculations deal with real numbers. Even if points have integer coordinates, the distance between two points might be an irrational number, or the image of a point under some transformation might be calculated using trigonometric operations. We therefore require a machine model that supports real numbers, or reasonable approximations thereof. Preparata and Shamos [PS85] describe such a model, the *random access machine (RAM)* model, in which each storage location is capable of holding a single real number and any location in memory can be accessed in equal time (*i.e.*, indirect addressing). The primitive operations that are provided at unit cost are:

1. arithmetic operations ($+$, $-$, $\times$, $\div$);
2. comparison between two real numbers ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$);
3. accessing a memory location (integer addresses); and,
4. common analytic functions, such as k^{th} root, trigonometric functions, exponent, and logarithm.

As these operations happen in constant time for hardware floating-point operands, this model is asymptotically accurate; therefore, this model is suited to analysing algorithms that are implemented in high-level programming languages, such as C and C++.

This thesis assumes a RAM model of computation.

3.3.2 What to Analyse

The *execution* or *running time* is a measurement most commonly used to describe the performance of an algorithm. In §3.3.1, we explained that for a given problem, an algorithm's cost is generally expressed as a function in terms of the size of an instance being solved. When analysing the running time of an algorithm, the function reflects the number of primitive operations on the input. The input size of an instance is dependent on the nature of the problem; for example, sorting a list of points may be expressed in terms of the number of points in the input while the multiplication of two integers may be discussed in terms of the number of bits required in the representation of the integers.

While the running time is generally the most important resource to measure, another important resource to consider is *space* or memory used to store the data structures used in the algorithm.

3.3.3 Units of Measure: Growth of Functions

The previous section discussed the measure of resource-use of an algorithm. This theoretical measure of the execution time both helps to characterize an algorithm's efficiency and also provides a mechanism whereby different algorithms' performance can be compared. As for any measurement, we require some sort of *unit* of measure for understanding the value and for making comparisons. In this section, we discuss the concept of growth of functions in order to interpret the analysis of an algorithm.

In the previous section, we saw that an algorithm's running time cost is expressed as a function of the size of the instance being solved. Suppose we have such a function, describing the running time of an algorithm in terms of the size of the input. The function is typically an increasing function: as the size of the input gets larger, so too will the value of the function. This increasing behaviour of the function is called the growth of the function. In algorithm analysis, an algorithm's performance is characterized for very large inputs; that is, as the size approaches *infinity*.

The notation used to describe the growth of a function as the size of the input approaches the limit is called *asymptotic* notation. The different types of asymptotic notation represent upper, lower, and tight bounds on the running time. We review

this notation below.

Given a function $f(n)$, we say that $f(n) \in O(g(n))$ if $g(n)$ is an upper bound on $f(n)$. Formally, $O(g(n))$ describes the set of functions:

$$O(g(n)) \in \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \\ \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$$

In other words, for all values $n \geq n_0$, the function $g(n)$ is larger than or equal to $f(n)$ to within a constant factor c .

Defining a lower bound is analogous to the upper bound. The notation for representing a lower bound on the function $f(n)$ is $f(n) \in \Omega(g(n))$. This describes a function $g(n)$ that is smaller or equal to $f(n)$ to within a constant factor c for all $n \geq n_0$.

An asymptotically tight bound is more strict than the others because it combines the conditions for both upper and lower bounds into the requirements for identifying an asymptotically tight bound. The asymptotically tight bound of a given function $g(n)$ is denoted by $\Theta(g(n))$ and defines the set of functions:

$$\Theta(g(n)) \in \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \\ \text{such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}.$$

In other words, for $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor.

The Θ -notation bounds a function from above and below. In other words, a function $f(n)$ that is $\Theta(g(n))$ automatically satisfies $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

In the analysis of algorithms, we generally consider the worst-case behaviour³ and restrict ourselves to O -notation. When an upper bound is defined on the worst-case running time of an algorithm, it is by implication bounding the running time on arbitrary inputs as well. Another advantage of using O -notation is that it is often easy to describe the running time of an algorithm by inspecting the algorithm's overall structure [CLR90, p.27]. For example, a doubly-nested loop structure on the entire

³The worst-case behaviour is a scenario in which the input is a special case that results in the maximum use of resources.

input of size n immediately yields an $O(n^2)$ upper bound on the worst-case running time: for each of n elements, perform n operations.

3.4 The Tree: An Abstract Data Type

The most general type of tree is a **free tree**, which is a connected, acyclic, undirected graph [CLR90]. A free tree on a set of vertices contains one path from any vertex in the tree to any other vertex in the tree (connected) and only one path between those two vertices (acyclic). There is no specified direction of travel within the tree (undirected).

A **rooted tree** is a free tree in which one of the vertices is distinguished from the others. This special vertex is called the root of the tree. The vertices in a rooted tree are often called **nodes**. Introducing the idea of a root places an order (and imposes a terminology) on the nodes in the tree. For example, the root r is the *ancestor* of all nodes in the tree. All nodes in the tree are *descendants* of the root. More specifically, if the last edge on a path from the root r to a node x has endpoints x and y , then y is a *parent* of x and x is a *child* of y . Similarly, the parent of y is the *grandparent* of x . A node with no children is called an *external* or *leaf* node, while all other nodes are referred to as *internal*.

A tree rooted at a child of a node x is called a *subtree* of x .

An **ordered tree** is a tree for which the children of a node are explicitly ordered; that is, there is a distinction between the first child, second child, and so on.

A **binary (rooted) tree** is an ordered rooted tree in which all nodes have at most two children. The children of a node in a binary tree are distinguished as the *left child* or *right child* of their parent node, depending on their relative order with respect to the parent node. This enables the definition of a total order on the nodes according to their location in the in-order traversal of the tree's nodes. The left child comes before the parent, while the right node comes after the parent in the overall ordering of nodes.

3.4.1 Binary Search Tree

An important application of binary trees is in their use in searching [Wei92]. A **binary search tree** is a binary tree for which each node v of the tree is assigned a *key* value. Keys are elements from a linearly-ordered set (*i.e.*, an order is defined for all pairs of items in the data set). For instance, $\leq$ (less than or equal to) is a total order⁴ on integers: for any two integers, one of them is less than or equal to the other. A binary search tree results from inserting these ordered keys into a binary tree such that for every node v in the tree, the left subtree contains nodes with keys smaller than (or equal to) v 's key, and the right subtree contains all nodes with keys strictly larger than the key for v . The key-nodes make up the *internal nodes* of the binary search tree.

Keys may be associated with certain information; that is, keys provide an index into the information stored at the leaves of a binary search tree. Generally, the information pertaining to a certain key k is located in the *external* or *leaf node* of the tree where the search on k ends. The value *null* is used to represent empty keys or information. A null node is denoted **nil**.

The general algorithm for searching a tree x on key k is:

BINARYTREESearch(x , k)

1. while x is internal node
2. if $k < x.key$
3. then $x \leftarrow x.left$
4. else $x \leftarrow x.right$
5. if $k = x.key$
6. then return x
7. else return *nil*

An alternative popular approach for representing information in a binary search tree is to store the information pertaining to a certain key k in the same node as the key. External nodes are all assigned *nil*. This approach simplifies the search terminating condition since the distinction between internal and external nodes does not need to be made. Using this approach, the above algorithm changes so that line 1

⁴Formally, a total order is a relation that is reflexive, transitive, and antisymmetric.

becomes:

while $x \neq \text{nil}$ and $k \neq \text{key}[x]$

and lines 5-7 are replaced by the simple return statement: `return x`.

Search trees support many dynamic-set operations⁵ [CLR90, p.244] and can implement any of these operations in time proportional the height of the tree.

A **balanced binary search tree** structure on n nodes is a binary search tree in which any path from the root to a leaf node is not longer than $\lg n$.⁶ Balanced binary search trees are used to ensure that basic dynamic set operations take $O(\lg n)$ time in the worst case [CLR90, p.263].

⁵The basic dynamic-set operations are: insert, search, delete, successor, predecessor, minimum, and maximum.

⁶ $\lg(n) \equiv \log_2 n$.

Chapter 4

Approximation of $w(p)$

The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them. - Sir William Lawrence Bragg

Chapter 1 introduced the Weber problem (see Problem 1.1). It was shown by Bajaj [Baj88] that the solution to the Weber problem can be an irreducible polynomial when the number of demand points is larger than four, leaving only approximation methods as a means of computing the optimum solution. Bose *et al* [BMM01] provide an approximation scheme that computes the sum of Euclidean distances, $w(p)$, from a point p to all points p_i in a given set S , in sub-linear time.

Bose *et al*'s approximation scheme uses simplicial cones, k -oriented distances, and dominance queries to approximate $w(p)$. These concepts are outlined in Sections 4.1 and 4.2. Section 4.3 describes the approximation algorithm and Section 4.4 proves the theoretical bounds on the approximation constant.

4.1 Simplicial Cones

The notion of simplicial cones is needed to understand the approximated distance scheme. We focus on the two-dimensional case. The intersection of any 2 lines forms a simplicial cone in $\mathbb{R}^2$, with an apex at the point where the 2 bounding planes meet.

We denote by $C = \{c_0, \dots, c_{k-1}\}$ a collection of k simplicial cones of fixed angle θ , $0 < \theta \leq \pi$, such that:

1. each cone $c \in C$ has its apex at the origin; and,
2. $\cup_{c \in C} c = \mathbb{R}^2$.

The angle θ is defined by k , its value is computed by dividing the circle into k equal slices: $\theta = 2\pi/k$. The construction of C can be imagined as the successive rays that result from rotating the positive x -axis over angles $i \cdot \theta$, $0 \leq i < k$.

Figure 4.1: A set of k simplicial cones C in $\mathbb{R}^2$

Each cone's bounding planes define the axes of a coordinate system.¹ The axes that define c_i are denoted t_i and t_{i+1} , where t_i has direction $i\theta$, $0 \leq i \leq k$ and $t_0 \equiv t_k$, as depicted in Figure 4.1.

A point p represented in the coordinate system of cone c_i is denoted $t_i(p)$ and has coordinates $t_i(p)[0], \dots, t_i(p)[d-1]$. We call these the k -oriented coordinates of p in c_i and term c_i defined by t_i and t_{i+1} the k -oriented plane. Figure 4.2a depicts the k -oriented coordinates of the point p in c_i .

¹Contrary to the general coordinate system described in (Chapter 1 Footnote 1), the coordinate system defined by one of the cones is not orthogonal for values of k other than $k = 4$. The k -oriented coordinate system assumes measurement within a network of uniformly spaced lines that are parallel to the axes defining the coordinate system.

4.2 The d_k Distance Function

The k -oriented distance in c_i between two points x and y , denoted $d_k(x, y)$, is the length of shortest path from x to y traveling only in distances parallel to the axes of c_i . This is effectively computed by translating x to the origin (see Figure 4.2b):

$$d_k(x, y) \equiv d_k(x - x, y - x) \equiv d_k(O, y - x)$$

Figure 4.2: a. k -oriented coordinates b. k -oriented distance

The $d_k(x, y)$ distance measure in the k -oriented plane is similar to the normal rectangular metric in the Euclidean plane, but is not itself a metric because it may not be symmetric. Figure 4.3 shows the isosimilarity contours² of the Rectangular metric and the k -oriented distance. The d_k metric appears visually as, and is effectively, a sheared or skewed version of the Rectangular metric.

Figure 4.3: Isosimilarity contours for l_1 -norm and k -oriented distance

The idea of calculating the k -oriented distance between two points as a translation in the k -oriented plane is a key concept of the $w(p)$ approximation algorithm.

²The full set of points at distances d from the origin.

4.3 Fast evaluation of $w(p)$ using d_k distances

This section reviews the approximation scheme for summing Euclidean distances proposed by Bose *et al* [BMM01]. The section ends with a sketch of the algorithms for preprocessing and querying the set of points S in Problem 1.1 in order to compute the approximation to $w(p)$.

Recall the argument from Chapter 1 that in order to solve Problem 1.1 efficiently, we can approximate the sum of Euclidean distances from a single point, say q , to a set of points $S = \{p_1, \dots, p_n\}$:

$$w(q) = \sum_{i=1}^n d(q, p_i) \quad (4.1)$$

where $d(x, y)$ is the Euclidean distance between x and y .

The approximation of $w(q)$ using k -oriented distances is denoted as:

$$w_k(q) = \sum_{i=1}^n d_k(q, p_i) \quad (4.2)$$

We will show that it is easy to compute (4.2) quickly and efficiently by preprocessing the points in S . The computation is not exact, however, and we show in Section 4.4 that it in fact has bounds such that $w(q) \leq w_k(q) \leq \frac{1}{\cos \frac{\theta}{2}} w(q)$. Since $\frac{1}{\cos \frac{\theta}{2}}$ approaches 1 as θ approaches 0, the approximation is expected to get better as the number of cones k is increased.

Definition 4.1 *Point Dominance*

A point p dominates a point r in $\mathbb{R}^d$ if $p[j] \geq r[j]$ for all $0 \leq j < d$. This relationship is denoted $p \geq r$. Similarly, in the k^{th} -orientation, p dominates r in c_i if $t_i(p)[j] \geq t_i(r)[j]$ for all $0 \leq j < d$.

The key step in the approximation of the sum of distances is to translate the simplicial cone structure C so that its apex lies at q and determine the contribution of the points in each cone individually through a series of dominance queries as follows:

$$w_k(q) = \sum_{j=1}^k \sum_{i=1}^n d_k(q, p_i) \cdot [p_i \in c'_j] \quad (4.3)$$

where c'_j denotes c_j translated such that its apex is at q , and $[X]$ is Knuth's predicate evaluation function [GKP94].³

At this point, two observations allow us to evaluate $w_k(q)$ as a pair of dominance queries:

1. $p_i \in c'_j$ if and only if p_i dominates q in c'_j ; and,
2. $d_k(q, p_i) = d_k(O, p_i - q)$ (by translation)
 $= \|t_j(p_i - q)\|$ (by definition)
 $= \|t_j(p_i)\| - \|t_j(q)\|$

Applying these two observations to (4.3) results in the following expression containing two dominance queries:

$$\begin{aligned} w_k(q) &= \sum_{j=1}^k \sum_{i=1}^n \|t_j(p_i)\| - \|t_j(q)\| \cdot [t_j(q) \leq t_j(p_i)] \\ &= \sum_{j=1}^k (\sum_{i=1}^n \|t_j(p_i)\| \cdot [t_j(q) \leq t_j(p_i)] \\ &\quad - \|t_j(q)\| \cdot |\{p_i : t_j(q) \leq t_j(p_i)\}|) \end{aligned} \quad (4.4)$$

Where the individual dominance queries are:

$$\sum_{i=1}^n \|t_j(p_i)\| \cdot [t_j(q) \leq t_j(p_i)] \quad (4.5)$$

and

$$\|t_j(q)\| \cdot |\{p_i : t_j(q) \leq t_j(p_i)\}| \quad (4.6)$$

Effectively, $w_k(q)$ is determined by summing the k -oriented distance from the origin to each point that dominates q (4.5) and subtracting once for each of those points the k -oriented distance from the origin to q (4.6). This is repeated for each $c_i \in C$. These are a dominance query, and a dominance counting query, respectively.

Breaking (4.4) down into smaller parts demonstrates how the precomputation of $d_k(O, p_i)$ for all $p_i \in S$ can be useful. Given a data structure $\mathcal{T}$ that supports the insertion of and dominance queries on (ordered) point data, we can describe the preprocessing and query algorithms as follows:

³ $[X] = 1$ if the expression X is true; otherwise $[X] = 0$.

Preprocessing:

1. for $j = 1$ to k
2. Construct $\mathcal{T}_j$, containing $\{t_j(p_1), \dots, t_j(p_1)\}$

Querying point q (evaluating $w(q)$):

1. $sum = 0$
2. for $j = 1$ to k
3. Perform dominance query on q in $\mathcal{T}_j$: $sum = sum + (4.5) - (4.6)$

For each cone c_j , the dominance query returns the sum of distances $d_k(O, p_d)$ for all points dominating q in c_j and the number (count) n_j of those points. The only new computation required is $d_k(O, q) \cdot n_j$ for each c_j .

In the next chapter, we will describe some data structures that support insertion, deletion and dominance counting queries of two-dimensional point data in $O(\log^2 n)$ time and require $O(n \log n)$ space. The existence of such data structures implies that we can support querying in $O(k \log^2 n)$ time and preprocessing in $O(kn \log n)$ time and space.

4.4 Approximation Bounds

In order for the approximation scheme to be reasonable, the estimated value $d_k(x, y)$ must be a good approximation of the Euclidean distance $d(x, y)$. In this section, we show that the k -oriented distance is not more than a constant factor ϵ (dependant only on the number of cones, k) larger than the Euclidean distance.

Figure 4.4: Euclidean distance $d(x, y)$ versus approximated distance $d_k(x, y)$

Figure 4.4 depicts a simplicial cone c_i where θ is the angle between the axes t_i and t_{i+1} , $\theta < \pi$. The segment $\overline{ps}$ represents $d(p, s)$, the Euclidean (straight-line) distance,

while the path $\overline{prs}$ represents $d_k(p, s)$, the k -oriented distance, between the points p and s . Recall that the k -oriented distance is the length of the shortest path travelling only in distances parallel to the axes of c_i .

$$d_k(x, y) = |pr| + |rs|$$

The triangle inequality states $|ps| \leq |pr| + |rs|$, which immediately implies that $d(p, s) \leq d_k(p, s)$. Alternatively,

$$\epsilon \cdot |ps| = |pr| + |rs| \tag{4.7}$$

where ϵ is some constant. We will now proceed to show an upper bound on $d_k(p, s)$.

Lemma 4.4.1 $d(p, s) = \epsilon_\alpha d_k(p, s)$, where ϵ_α is a function of α and is equal to $\frac{\sin \alpha + \sin(\theta - \alpha)}{\sin \theta}$.

Proof: Referring to Figure 4.4, and using the law of sines, we get:

$$\frac{1}{\sin(\pi - \theta)} = \frac{|rs|}{\sin \alpha} = \frac{|pr|}{\sin(\theta - \alpha)}$$

Simplifying and using the property $\sin(\pi - x) = \sin(x)$ yields:

$$|pr| = \frac{\sin(\theta - \alpha)}{\sin \theta} \quad |rs| = \frac{\sin \alpha}{\sin \theta} \quad \text{and} \tag{4.8}$$

Substituting (4.8) into (4.7) results in an expression for ϵ in terms of α :

$$\epsilon_\alpha = \frac{\sin \alpha + \sin(\theta - \alpha)}{\sin \theta} \tag{4.9}$$

□

Lemma 4.4.2 $\max_{0 \leq \alpha \leq \frac{\theta}{2}} \epsilon_\alpha = \epsilon_{\frac{\theta}{2}}$

Proof: To find the value of α that maximizes ϵ_α , we take the partial derivative of (4.9) with respect to α and set it to 0.

$$\begin{aligned} \frac{\delta \epsilon}{\delta \alpha} &= \frac{\cos \alpha}{\sin \theta} + \frac{-\cos(\theta - \alpha)}{\sin \theta} = 0 \\ \frac{1}{\sin \theta} (\cos \alpha - \cos(\theta - \alpha)) &= 0 \\ \cos \alpha &= \cos(\theta - \alpha) \\ \alpha &= \theta - \alpha \\ \alpha &= \frac{\theta}{2} \end{aligned} \tag{4.10}$$

□

Theorem 4.4.1 *The k -oriented distance approximation of the Euclidean metric gives a result that is not more than a constant factor $\epsilon \leq \frac{1}{\cos \frac{\theta}{2}}$, $\theta < \pi$, larger than the Euclidean distance; that is,*

$$d(x, y) \leq d_k(x, y) \leq \epsilon d(x, y)$$

Proof: The lower bound follows from the triangle inequality:

$$d(p, s) \leq d(p, r) + d(r, s) = d_k(p, s)$$

The upper bound follows directly from Lemma 4.4.2:

$$d_k(x, y) \leq \epsilon d(x, y)$$

Substituting (4.10) into (4.9) yields the worst-case approximation constant:

$$\epsilon_{max} = \frac{2 \sin \frac{\theta}{2}}{\sin \theta} = \frac{2 \sin \frac{\theta}{2}}{2 \cos \frac{\theta}{2} \sin \frac{\theta}{2}} = \frac{1}{\cos \frac{\theta}{2}}$$

□

Table 4.1 shows, for different values of k , the corresponding angles θ , calculated worst-case approximation constant ϵ_{max} , and maximum percentage error.

k	θ (degrees)	ϵ_{max}	% error	k	θ (degrees)	ϵ_{max}	% error
3	120	2.000	100	12	30	1.035	3.5
4	90	1.414	41	13	27.7	1.030	3.0
5	72	1.236	24	14	25.7	1.026	2.6
6	60	1.155	16	15	24	1.022	2.2
7	51.4	1.110	11	16	22.5	1.020	2.0
8	45	1.082	8.2	17	21.2	1.017	1.7
9	40	1.064	6.4	18	20	1.015	1.5
10	36	1.051	5.1	19	18.9	1.014	1.4
11	32.7	1.042	4.2	20	18	1.013	1.3

Table 4.1: Worst case approximation error ϵ as a function of k

Charting these values (see Figure 4.5) demonstrates that as k increases, the improvement in the approximation constant becomes less significant.

In the testing and analysis section of this thesis (Chapter 7), we characterize the theoretical approximation constant with respect to the experimental results. We also attempt to draw conclusions on good choices for k ; that is, we make a statement concerning values for k that provide a reasonable approximation for $w(p)$ while minimizing the required storage.

Figure 4.5: Approximation constant versus number of cones

Chapter 5

Geometric Searching

Geometry is the only science that it hath pleased God hitherto to bestow on mankind. - Thomas Hobbes

This chapter gives an overview of geometric searching, in particular, range searching. Data structures used for geometric range searching; the range tree and the persistent search tree, are also introduced. The reader who is familiar with these topics might skip this chapter.

In a general sense, searching means to look thoroughly into or over a context in order to locate or discover something. The context in which we search may involve, for example, determining whether a particular word exists in a file or whether a set of points possesses a certain property. The formulation of the search; that is, describing what we are looking for in the search context, is called a **query**.

Searching arises frequently in geometric applications: computer graphics (which objects are overlapping?), geographic information systems (returning information related to a given location), and computer aided design systems (where do objects intersect? which objects are within the display area?). Preparata and Shamos [PS85] identify two major types of geometric searching problems:

Location problems Given a point and a space (*e.g.*, plane) divided into regions, identify the region that the query point lies in.

Range-search problems Given a set of points in space and some standard geometric shape arbitrarily translatable in space (query domain), repeat or count all the points contained within the query domain.

In this thesis, we focus on range-search problems. The rest of this chapter introduces range searching, followed by a description of the data structures used for range searching (§ 5.2): range trees and persistent search trees.

5.1 Range Searching

Range searching arises in many geometric and database [dBvKOS97] applications and is a central problem in computational geometry. In general, a search on a given set of geometric data involves determining whether the set possesses a certain property (*e.g.*, whether an N -gon contains a specified point, whether a set is convex). In some cases, only one query is required to find the answer; in others, many queries might be performed repeatedly on the same set of data. The former is called a *single-shot query*, while the latter is referred to as *repetitive-mode queries*.

Many single-shot queries can be performed in $O(n)$ time, since each of the elements in the geometric set might need to be examined only once or a constant number of times. For example, querying the sum of distance from a point q to all points p_i in a given set S of n points, requires just one distance computation for each point in S , hence this single-shot query takes time linear in the size of the input set S .

When the same point set is queried more than once (*e.g.*, suppose we query the sum for a number of different points q_j on the same set S) it may be useful to preprocess S into a data structure [Aga96] in order to answer the query faster. This is accomplished at an expense, so the analysis of such a data structure focuses on four separate cost measures [PS85]:

Query time Time required to respond to a single query.

Storage Memory required to store the data structure.

Preprocessing Time Time needed to arrange the data into the search data structure.

Update Time Time to add an item to or delete an item from the data structure.

Two common instances of range-searching include *range counting* and *range reporting*; that is, counting the number of points within a given query range and reporting all the points in the query range, respectively. A typical range query follows:

Problem 5.1 *Given a set S of n points, and a box (or simplex) with sides parallel to the coordinate axes, count or report the points that are contained inside the box (respectively, simplex).*

The two-dimensional case in the Euclidean plane can be translated to counting or reporting all the $p_i = (x_i, y_i)$ that satisfy $x_{\min} \leq x_i \leq x_{\max}$ and $y_{\min} \leq y_i \leq y_{\max}$ for the rectangle defined by $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$. Analogously, the query range for the three-dimensional case is a rectangular prism or box.

The one-dimensional case translates to a range-search on a set of real numbers. The *points* translate to numbers x_i on the real axis and the *query box* can be interpreted as an interval $[x_L, x_R]$.

Figure 5.1: One- and Two-Dimensional Range Search

In the next section, we review some data structures that are commonly used for range queries.

5.2 Data Structures for Geometric Range Searching

Orthogonal range searching involves finding all points inside a d -dimensional axis-parallel box. (See Problem 5.1).

The one-dimensional range searching problem, which consists of finding all values within a given range, is efficiently solved using a well-known data structure: the balanced binary search tree (§3.4.1), denoted $\mathcal{T}$. The internal nodes store coordinate values as keys to guide the searches while the leaves contain the actual *points*. Reporting the points in a query range $[x_L, x_R]$ proceeds as follows:

1. Search the nodes v of $\mathcal{T}$ for both values x_L and x_R until x_L is smaller than *and* x_R is larger than $v.key$. Denote by v_{split} this first node v that splits the search range such that $v.key$ no longer lies strictly above or below the search range.
2. Continue the search separately on the left and right subtrees of v_{split} :
 - left subtree** Search with x_L in the left subtree of v_{split} , reporting all points in the right subtree of every node v where the search path goes left.
 - right subtree** Search with x_R in the right subtree of v_{split} , reporting all points in the left subtree of every node v where the search path goes right.
3. Check the leaves where the search path ends against the query range to determine if they lie inside or outside of the query range.

Figure 5.2: One-dimensional Range Tree Query

Lemma 5.2.1 *The cost of querying a one-dimensional range tree $\mathcal{T}$ on n points is $O(\lg n)$ to count the number of points in the query range and $O(\lg n + \delta)$ to report the actual points, where δ is the number of reported points.*

Proof: In the simplest case, the query range contains one node. The search path in $\mathcal{T}$ ends at the node v_{split} and has length $O(\lg n)$. One point is counted or reported, resulting in a query time of $O(\lg(n) + 1)$.

In the worst case, v_{split} is the root. Both search paths for x_L and x_R have length $O(\lg n)$ in $\mathcal{T}$, resulting in a query time proportional to $2\lg n$ and the number of reported points, δ , if the points themselves are being reported. For a counting query, the subtrees do not need to be traversed in order to report the leaves and the $O(\delta)$ term is not included.

It follows that the cost for querying a one-dimensional range tree is $O(\lg n + \delta)$.

□

The one-dimensional binary tree range-search generalizes to higher dimensions. Given the point set, S , of d -dimensional point data, the d -dimensional range query is considered as d successive one-dimensional range queries; that is, a one-dimensional query is performed for each of the coordinates of the points in S . The following sections review some data structures commonly used for multi-dimensional range searching: range trees, fractional cascading, and persistent trees.

5.2.1 Range Trees

A range tree is a multi-dimensional search tree for performing range queries on a set S of n points. In one dimension, the range tree is simply a balanced binary search tree on one-dimensional point data. In d -dimensions, the range tree is recursively defined. The definition of the range tree closely follows [dBvKOS97].

First, we introduce the notion of the *canonical subset* of a node in a binary tree. The **canonical subset** of a node v , denoted $S(v)$, is the subset of points stored in the leaves of the subtree rooted at the node, v . For the root, the canonical subset is the entire set of points S while the canonical subset of a leaf is simply the point stored there.

Definition 5.1 *A range tree for points in d -dimensional space is defined in two parts as follows:*

1. *The first-level, or main, tree, T is a balanced binary search tree on the first dimension of the point data, S . Each node v has a second-level range-tree structure associated with it, denoted $T_{\text{assoc}}(v)$, operating on the points in $S(v)$.*
2. *The i^{th} -level range tree, $T_{\text{assoc}}(v)$ is a $((d - i) + 1)$ -dimensional range tree on the i^{th} -dimension of the points in $S(v)$. Each node v' is associated with an $(i + 1)^{\text{st}}$ -level range tree on the points in the canonical subset of v' .*

We turn our focus to the two-dimensional case.

A two-dimensional range tree on a set S of n points has a first-level tree sorted on the x -coordinates of the points in S . Each node v of this first-level tree contains a (pointer to a) simple binary tree sorted on the y -coordinates of the canonical subset of v . The structure is shown in Figure 5.3.

Figure 5.3: 2-dimensional Range Tree

A two-dimensional query on the set S of n points reports on the points in the query range $[x_L, x_U] \times [y_L, y_U]$ (recall Figure 5.1). The query proceeds on the main tree with the range $[x_L, x_U]$ exactly as if performing a one-dimensional query, with one exception. Instead of reporting the points in the subtrees along the search path, as described above, another one-dimensional search with the range $[y_L, y_U]$ is carried out on the associated trees, $T_{\text{assoc}}(v.\text{child})$, of the nodes v along the search path. Specifically, the search proceeds as follows from the node v_{split} in the main tree:

1. **left subtree** Search with x_L in the left subtree of v_{split} . Perform a one-dimensional range query on the canonical subset $S(v.right)$ of every node v where the search path goes left.
- right subtree** Search with x_R in the right subtree of v_{split} . Perform a one-dimensional range query on the canonical subset $S(v.left)$ of every node v where the search path goes right.
2. Check the leaves where the search path ends against the query range to determine if they lie inside or outside of the query range.

Lemma 5.2.2 *The cost of querying an axis-parallel box on the two-dimensional range tree $\mathcal{T}$ on n points is $O(\lg^2 n)$ to count the number of points in the query range and $O(\lg^2 n + \delta)$ to report the actual points, where δ is the number of reported points.*

Proof:

Consider the search algorithm. For each node v in the search path of $\mathcal{T}$, the one-dimensional range query is performed on $\mathcal{T}_{assoc}(v)$, which takes $O(\lg n)$ time to count the points and $O(\lg n + \delta_v)$ time to report the points. The total time to query the two-dimensional range tree is therefore the sum of the time to perform the one-dimensional query on $\mathcal{T}_{assoc}(v)$ over all visited nodes v in the search path:

$$\sum_{v \in \text{searchpath}} O(\lg n + \delta_v)$$

The result of $\sum_v O(\delta_v)$ is $O(\delta)$, time proportional to the total number of reported points. The length of the visited paths for x_L and x_R in the main tree $\mathcal{T}$ are each $O(\lg n)$. It follows that the total query time $\sum_v O(\lg n + \delta_v)$ is $O(\lg^2 n + \delta)$, where $O(\delta)$ is replaced by a constant term if the query is a counting query.

□

Lemma 5.2.3 *The storage cost for the two-dimensional range tree is $\Theta(n \lg n)$.*

Proof: The proof is computed by a recursion-tree on the size of the associated structures, $\mathcal{T}_{assoc}(v)$, for all v in $\mathcal{T}$ (see Figure 5.4).

Figure 5.4: Recursion tree on size of $\mathcal{T}$ plus $\mathcal{T}_{assoc}(v)$ for all v

The first-level of the tree contains n nodes. The top-most node is called *root*. The associated structure, $\mathcal{T}_{assoc}(\text{root})$, of *root* contains all the points in S ; therefore requiring $O(n)$ storage. The canonical subsets of each child of *root* are of size $|S|/2$; therefore each $\mathcal{T}_{assoc}(\text{root.child})$ requires $O(n/2)$ storage. The recursion tree depicts, without loss of generality, a balanced scenario. At each level of $\mathcal{T}$, we have a total of n nodes in all the $\mathcal{T}_{assoc}$. There are $\lg n$ levels in the tree for a total of $n \lg n$ nodes in all trees. We have, in all cases, $n + n \lg n$ nodes. It follows that the tree requires $\Theta(n \lg n)$ space.

□

A technique, called *Fractional Cascading*, offers an improvement over range trees in query time by a $\lg n$ factor. It achieves the better query time by reconsidering the algorithm for performing queries on the associated structures $\mathcal{T}_{assoc}(v)$. The proposed secondary-level searches are modified to take only $O(1 + \delta_v)$ time, resulting in a total query time of $O(\lg n + \delta)$. The improvement uses the fact that many one-dimensional searches are being performed with the same range. For more detail, we refer the reader to [dBvKOS97, CG86].

5.2.2 Persistent Trees

A persistent search tree differs from an ordinary search tree in that after a series of updates, old versions of the tree can be accessed. A data structure that is not persistent is called *ephemeral*, which literally means lasting for a very short time. A common and direct approach to implementing persistence is to start with an ephemeral data structure for sorted sets, such as a binary search tree, and make it persistent [ST86].

The basic dynamic set operations can be organized into two categories: access operations, and update operations. Access operations constitute those operations that get information from the tree without changing the structure of the tree: search, successor, predecessor, minimum, and maximum. Update operations, on the other hand, constitute those operations that change the structure of the tree: insert and delete. The treatment of these operations by a particular persistent tree characterizes the data structure as one of two types identified in the literature [ST86, DSST89]: *fully-persistent* or *partially-persistent*.

A **fully persistent** data structure allows accesses and updates to all versions of the tree, while a **partially persistent** data structure allows access to all versions and updates only on the most current tree. Partially persistent data structures are much easier to implement than those that are fully persistent since we require handling updates only on the most current tree. We don't need full persistence for the purpose of this thesis, therefore, we focus on the partially persistent case. As we will see, updates are only required on the most current version of the tree.

There is some evidence in the literature supporting the persistent search tree as a data structure for range searching [Cha85], but only for the specific case of range-querying called *dominance queries*. The rest of this section explains how a persistent search tree can be used for the special case of range searching where the upper corner of the query box is located at (∞, ∞) .

Persistence is achieved through path-copying on updates [ST86]. Path-copying provides a method of retaining the old version of the tree when a new version is created by an update. The entire tree could be copied, but at a high cost of $O(i)$ time and space per update, where i represents the size of the tree at the time of the update. Instead, copying only the nodes that are modified by an update operation is sufficient in maintaining different versions of the tree, provided there is some way to

distinguish between the different versions. Path-copying is summarized as follows:

1. Any node that contains a pointer to a node that is copied must itself be copied.
2. Assuming that every node contains pointers only to its children, then copying one node v requires copying the entire path from the root to v .

The cost for copying a path is $O(\lg i)$ time and space per update, where i represents the size of the tree at the time of the update. Figure 5.5 demonstrates two consecutive versions of a simple binary tree resulting from the insertion of node D into the persistent structure.

Figure 5.5: Path-copy in a Persistent Tree

Many persistent trees are defined in terms of time, explaining how to maintain a set of items that changes over time. The key for an item in the tree is generally taken as a timestamp indicating when the item was inserted into (or deleted from) the tree. The item's **value** is stored in the tree and indexed by the item's key. In some contexts, the notion of time is only indirectly relevant [Cha85]. In geometry, for example, it is convenient to consider the x -axis as a time axis and the y -axis as an axis for defining item values, therefore inducing a one-to-one correspondence between time and point position in the plane. We explain in detail how range queries translate to querying the state of the data structure “attached” to the x -axis at a particular discrete event.

The persistent data structure $\mathcal{T}$ for performing range searching on a set S of n points with a two-dimensional axis-parallel query box is based on the binary search tree. Building the persistent tree is achieved by visiting each p_i in S as follows:

1. Generate a list $S_{x\text{-sorted}}$ by sorting the points in S by their x -coordinates, from largest to smallest.
2. Traverse $S_{x\text{-sorted}}$, updating the persistent search tree at each point as follows:
for each $p_i \in S_{x\text{-sorted}}$, $0 \leq i < |S_{x\text{-sorted}}|$, insert p_i into $\mathcal{T}$. We call the x -coordinate of p_i the key of $\mathcal{T}_i$.

We denote by $\mathcal{T}_i$ the version of the tree resulting from the i^{th} update operation. Figure 5.6 shows an example of a small point set and the persistent tree representing that point set.

Figure 5.6: Persistent Tree and Query on Small Point Set

Recall that persistent search trees are suitable for a special class of range searches, called dominance queries (see Definition 4.1); that is, a search of a persistent tree can easily and efficiently answer the question, “which points in S dominate the query point q ?” Querying the persistent tree with point q proceeds as follows:

1. Find the tree $\mathcal{T}_i$ with smallest key larger than $q[0]$; that is, find $\mathcal{T}_i$ representing all points in S with x -coordinate larger than $q[0]$.
2. Search with $q[1]$ in $\mathcal{T}_i$, reporting all points in the right subtree of every node $v \in \mathcal{T}_i$ where the search path goes left. This is similar to performing a one-dimensional range query with $q[1]$ and ∞ in $\mathcal{T}_i$ (§5.2).

Lemma 5.2.4 *The cost of a two-dimensional dominance query on the persistent search tree $\mathcal{T}$ on n points is $O(\lg n)$ to count the number of points in the query range and $O(\lg n + \delta)$ to report the actual points, where δ is the number of reported points.*

Proof: Consider the search algorithm. Step 1 can be accomplished with a binary search on the keys of $\mathcal{T}$. This is done in $O(\lg n)$ time.

The result of the first-level search points to a tree $\mathcal{T}_i$ on which a one-dimensional range-query is performed at a cost $O(\lg n)$ for counting and $O(\lg n + \delta)$ for reporting (see Lemma 5.2.1).

The total cost to query the persistent tree is the sum of the costs of performing the first-level and second-level queries:

$$O(\lg n) + O(\lg n + \delta) = O(\lg n + \delta)$$

It follows that the total query time is $O(\lg n + \delta)$. □

Lemma 5.2.5 *The storage cost for the two-dimensional range tree is $\Theta(n \lg n)$.*

Proof: The storage cost is computed using a simple summation by noticing that the construction of the persistent search tree for dominance queries only inserts points and never deletes them.

Suppose the keys are stored in a binary tree. This requires $O(n)$ storage for the first-level data structure on n points. Each key maps to a balanced tree in the persistent structure, which is built from n updates by insertion; each update copies a path proportional to $\lg i$, where i is the number of nodes in the current tree. The total number of nodes is therefore computed from:

$$n + \rho \sum_{i=1}^n \lg i$$

where ρ is some constant.

All binary trees have height at least $\lg i$, where i is the number of nodes in the tree. This results in a lower bound for the size of the persistent data structure:

$$n + \sum_{i=1}^n \lg i = n + \lg \prod_{i=1}^n i = n + \lg n! = \Omega(n \lg n)$$

Suppose a red-black tree is used for the second-level data structure. The red-black tree has height at most $3 \lg i$, where i is the number of nodes in the tree. It follows that the upper bound on the size of the second-level data structure is:

$$n + 3 \sum_{i=1}^n \lg i = n + 3 \lg \prod_{i=1}^n i = n + 3 \lg n! = O(n \lg n)$$

It follows that the storage for the persistent search tree (on two-dimensional point data) is $\Theta(n \lg n)$. $\square$

Chapter 6

Implementation of Range Search Data Structures

I really hate this damned machine

I wish that they would sell it

It never does quite what I want

But only what I tell it. - Dennie L. Van Tassel

The calculation of $w_k(p)$ occurs as a sequence of k dominance queries with a given point q with respect to a point set S (see Definition 1.2). Section 5.2 presented some data structures that are commonly used for performing range queries, a specific type of dominance query. In this chapter, we explore the implementation and querying of those data structures.

When we translate the simplicial cone structure C and determine the contribution of the points dominating q in each cone individually, we are effectively performing k dominance queries. For a cone, c_i , and query point q , the axis-parallel query box has its lower-corner at q , its upper-corner, opposite to q , at infinity; and, is bounded by rays parallel to the axes of c_i and emanating from q . The dominance query is equivalent to a range query with upper limit infinity. Figure 6.1 illustrates the k -oriented dominance query for two differently-oriented cones.

Two alternate implementations are considered as candidates for implementing a data structure that enables the calculation of $w_k(p)$. These are to use existing

Figure 6.1: k -oriented Dominance Queries

implementations of the foundation data structures and adapting them to our needs; or to build the data structures from scratch. The rest of this chapter explains both approaches and comments on some of the challenges each approach presented.

This chapter assumes the reader has basic knowledge of object-oriented programming.

6.1 Software Libraries and Reuse

In general, a software library is a group of software artifacts (*i.e.*, programming modules, classes or methods) that may be called by any program, provided the programming language is the same for both the library and the implementation. The purpose of a library is to provide a means of performing common tasks in some domain of application that the programming language does not directly address [Mey82]. Libraries offer a functional means of enriching the existing programming language with new instructions, implemented as procedure calls. They are, in effect, building materials for constructing more sophisticated software.

Within the last two decades, libraries of mathematical software and data structures have become widely accessible. In 1985, for example, Netlib began services to fill a need for distributing freely available mathematical software to the research community [BDGR95]. As a result, software produced from research in numerical analysis was made available to others who could benefit from the work, allowing them to avoid *reinventing the wheel* and thus dedicate more time to their own research. Since then, many data structures and algorithms libraries have become available, especially with

the emergence of object-oriented programming languages. Most object-oriented programming languages, such as Java and C++, include basic data structures libraries. These libraries shipped with programming languages provide sufficient functionality for many applications; however their usefulness is sometimes limited by their generality. Alternatively, many third-party data structures and algorithms libraries exist to fill specialized needs.

The predominant third-party software libraries available to the Computational Geometry community are the Computational Geometry Algorithms Library (**CGAL**) and the Library of Efficient Data Types and Algorithms (**LEDA**). Both libraries are implemented in the C++ programming language and provide basic geometric primitives such as basic objects (*e.g.*, points, vectors, and lines), predicates for testing properties or relationships among those objects, and common operations to apply to the objects, such as computing intersections and performing distance calculations. In addition, each library provides standard Computational Geometry data structures and algorithms, including convex hull, triangulation, smallest enclosing circle, and multidimensional query structures. LEDA provides an interface for visualization of geometric structures and algorithms, while CGAL interfaces with LEDA's visualization modules.

Using existing software artifacts in the development of programming projects is called **software reuse**. The availability of software libraries like LEDA can lead to quality software that is quickly produced and is more robust than software developed from scratch. Broadly, the reuse process involves three steps [BBM96]:

1. selecting an appropriate existing artifact;
2. adapting it to the purpose of the application; and,
3. integrating it into the software product under development.

Some common techniques for reusing functionality in object-oriented systems are class inheritance, object composition and parameterized types. **Class inheritance** reuses software artifacts by subclassing. It is a mechanism for extending functionality by adopting the functionality of an existing class (termed the **base** or **parent class**) and adding to it, resulting in an **augmented** or **derived (sub)class**. The internal

details of the parent class are often visible to the subclass and new functionality can be introduced by overriding existing functions and adding new ones. **Object composition** obtains more complex functionality through reuse by aggregating existing objects. Objects being composed need well-defined interfaces and must respect the interfaces of the objects they use since reference to other objects happens dynamically at run-time. No internal details of the existing objects are visible to the programmer. **Parameterized types** allow the programmer to define a type in generic terms of the other types it uses and supply specific types as parameters in place of the generic types at the point of use. Each of these methods is used at least once in the implementation stages of this thesis.

In the next section, we discuss our experience of using and adapting LEDA to the programming needs of this thesis. In particular, we focus on working with LEDA's range tree data structure.

6.2 The LEDA Experience

LEDA is a geometric library that is introduced as being easy to use, extendable, correct, and efficient [MN99, p.11]. The emphasis on ease of use is that the library does not require of the programmer a detailed knowledge of the computational geometry research or of the implementation of the data types and algorithms offered by LEDA; a basic course in algorithms and data structures should suffice for most users to be able to use LEDA. From the point of view of an average programmer, however, LEDA does not seem to be built with extensibility in mind. This section includes a discussion of our experience augmenting one of LEDA's data structure classes for this thesis. Warning: the results are disappointing.

The LEDA library is a single repository that puts to practice many major findings of the algorithms community and makes the resulting implementations directly accessible to anyone who needs them. Reusing software is generally preferred over building components from scratch. LEDA provides implementations for both range trees and persistent trees, which seems like an elegant fit to the needs of this thesis because the approximation to $w(p)$ employs these data structures for dominance queries. Recall that $w(p)$ is the *weight* of a point and is computed as the sum of distances from the

point p to all points p_i in a set S of n points. The approximation is denoted $w_k(p)$.

The range tree version of the simplicial cone structure used in the weight approximation algorithm is created by inheriting from the available LEDA range tree implementation. We are unable to use the existing implementation “as-is” because the nodes of the range tree need to be augmented with additional information and, consequently, the additional information needs to be updated during tree restructuring operations. The LEDA range tree is designed in such a way that reuse by inheritance seems to be encouraged: many members of the base class are declared as virtual for dynamic binding purposes¹ and are defined in a protected interface.² LEDA’s `d2_dictionary` class is a good example of reusing the range tree through inheritance.

The attractiveness of LEDA extends beyond the available implementations of the data structures we require. We find the support for graphical visualization of geometric objects useful in observing how our algorithm operates on a point set. Also, LEDA’s geometric primitives are useful in performing the transformations required to determine the k -oriented coordinates of a point. LEDA’s documentation provides that the primitive operations are implemented in such a way that numerical degeneracies resulting from computations are avoided where possible. Additionally, if the decision is made that computation with exact numbers is required, occurrences of the point data type can be replaced by the `rat_point` data type, which uses algebraic manipulation to provide an “exact” number type.

6.2.1 Augmenting the LEDA Range Tree

A simplicial cone c_i is essentially a range tree that stores point information from a set S of n points, the original set of points for which we want to approximate the weight of a query point. Recall that the keys for a range tree data structure, or more generally, a binary search tree based data structure, are the indexing elements for accessing nodes in the tree. The keys to guide insertion and searches in the simplicial

¹A parent class can declare certain functions as `virtual`, which indicates that dynamic binding should be used. Given a pointer of type `base*`, the compiler and loader will guarantee the correct correspondence between objects and the (virtual) functions applied to them at run time.

²A protected member of a class can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class.

cones are the k -oriented coordinates $(t_i(p_j)[0], t_i(p_j)[1])$ of the points $p_j \in S$. For a cone c_i , the first-level tree is built on the 0^{th} k -coordinates and the second-level trees on the 1^{st} k -coordinates of these points. For each point, both the original point in Cartesian space and the k -oriented distance between the point and the origin are stored as information in the external nodes of the range tree. The original point is stored for display purposes.

A notable characteristic of the LEDA range tree data structure is that it keeps internal and external nodes separate, instead of representing the two types as one physical entity. The implication of this design decision is that the restructuring operations can be quite complex as the integrity of the represented information must be maintained in two disjoint locations for individual data. This raises issues of data coherency and consistency.

The LEDA range tree implementation does not allow duplicate keys in the tree, forcing us to consider how duplicate data is treated. For example, two points that are identical can be handled by introducing a reference counter in the node to indicate the number of copies of said point. Special consideration is also required for points that have either identical first coordinates or identical second coordinates. These special cases require extensive modification of the source code of the parent data structures, which can be challenging since the range tree is already the third class deep in the inheritance hierarchy, as shown in Figure 6.2. As Robert Binder observes [Bin00, p.68], “Dynamic binding and complex inheritance structures create many opportunities for faults due to unanticipated bindings or misinterpretation of correct usage”.

Figure 6.2: LEDA range_tree Hierarchy

The *workaround* solution adopted for augmenting the LEDA range tree for this thesis is to include actual points from S as keys for both levels of the range tree, and

to assume that no two $p_j \in S$ are the same. As a consequence, the transformation from Cartesian coordinates to k -oriented coordinates occurs up to $3 \lg i$ times³ for each operation in a tree of size i , since the Cartesian points, instead of the k -oriented points, are stored at the internal nodes. In the following section, we discuss how the linear order is defined to handle the k^{th} orientation transformations on the point data.

Linearly Ordered Data

The base-level data structure for the range and persistent trees is a binary search tree (§3.4.1), for which the data must come from a totally ordered set.⁴ LEDA requires that the linear order on data types be clearly defined so that data may be correctly inserted into the range tree. LEDA uses the global `compare(type V1, type V2)` function as a mechanism for defining a full order on data. If a suitable comparison function doesn't exist for a given user-defined type, then one must be defined for LEDA by the programmer. Formally, the compare function describes a binary relation $\mathcal{R}$ on a set of data $\mathcal{D}$ for all x, y in $\mathcal{D}$ as follows:

$$\text{compare}(x, y) \begin{cases} < 0, & \text{if } x \mathcal{R} y \text{ and } x \neq y \\ = 0, & \text{if } x = y \\ > 0, & \text{if } y \mathcal{R} x \text{ and } x \neq y \end{cases}$$

Different approaches are considered for defining the linear order on k -oriented coordinates. In the spirit of software reuse, LEDA's geometric primitives, such as line intersection and rotation about a fixed point, are used in the implementation. The motivations for reusing LEDA components include ease of implementation, correctness of the available primitives, and avoiding having to rewrite existing code.

Given an arbitrary axis t , and two distinct points b and c on t , the methods considered for comparing k -coordinates using LEDA primitives are:

Signed area with a third arbitrary point Choosing a third point d , to the left of, and not on, the directed line t , we compute the signed area of $\triangle bcd$. This

³The LEDA balanced binary search tree is based on the red-black tree, a balanced binary tree for which the longest path from the root to any leaf node is at most $3 \lg m$, where m is the number of nodes in the tree.

⁴Binary search-tree property [CLR90, p.245]: Let x be a node in a binary search tree. If y is a node in the left subtree of x , then $\text{key}[y] \leq \text{key}[x]$. If y is a node in the right subtree of x , then $\text{key}[x] \leq \text{key}[y]$.

indicates whether $\angle bcd$ is a left-hand or a right-hand turn, from which we can deduce the relative order of b and c on the line. Implications: d must always be on the same side of t and must be chosen in such a way that the computation of the signed-area is conclusive (not degenerate). A good candidate for d may vary between cases.

Compare directions $\vec{bc}$ and $\vec{t}$ Determine the direction of the vector defined by b and c . If the direction is the same (or close to) that of t , then $b < c$; otherwise, $b > c$. This approach allows us to take into account small differences resulting from computation and we can compare the directions to within a certain tolerance.

Distance to predetermined point on line Given a point p on t that is bigger or smaller than all comparison data, the Euclidean distances $d(b, p)$ and $d(c, p)$ are computed. The relative order can be deduced from the magnitudes of those distances; for example, if p is the biggest point on t , then $b < c$ only if $d(b, p) > d(c, p)$; otherwise, $b > c$. Implications: p is an arbitrarily chosen point and requires knowledge of all comparison data prior to the first comparison. Additionally, it is necessary to know whether p is the biggest or smallest point on t .

Note that these comparison functions apply to two points on a reference axis. For comparing two points in an arbitrary cone with respect to a certain reference axis, the points must first be projected onto the axis. The approaches seem complicated and require the choice of arbitrary third-party objects for the comparison.

After considering the above-mentioned methods, a similarity is noted between the k -oriented distance approximation scheme and the construction of another cone-based approximation algorithm. The theta-graph, an approximation to the complete graph on n points [Smi97], uses a fairly straightforward approach to determining the relative order of k -oriented points. In addition, it does not require the programmer to choose any arbitrary reference objects for use in the comparison of two points, which is a common challenge in the methods considered, above. Figure 6.3 demonstrates the steps for computing the coordinates of a point in the k^{th} orientation. The translation

from one coordinate system to the other is equivalent to performing the shear and rotate affine transformations outlined in Section 3.1.1 (Equation 3.1).

Figure 6.3: Cartesian to k -oriented coordinate transformation

For the coordinates $t_i(p)$, the point p is projected in directions parallel to axes t_i and t_{i+1} respectively onto the axes t_{i+1} and t_i . The magnitude of $t_i(p)[0]$, for example, is determined by rotating the image of p on t_i about the origin until it coincides with the x -axis, calling the result p' . The magnitude of $t_i(p)[0]$ is then the x -coordinate of p' . Once the transformations on the point are completed and the k -coordinates known, the points can be compared as if they are points in the Cartesian plane.

The minimum information required by the compare function for performing a comparison of two Cartesian points with respect to a k -orientation are the points' Cartesian coordinates and the axes t_i and t_{i+1} , which define the reference coordinate system (cone c_i). The k -oriented operations require knowledge of the dimension of the tree in which the operations are taking place so that the appropriate reference axes are used in the operations.

Querying the Simplicial Cone Structure

The query method supplied by the LEDA `range_tree` class performs a query with an axis-parallel box. To reuse the available query method as a dominance query requires supplying a parameter to the query method that represents the upper limit of the query box. For a dominance query, this upper limit is analogous to the point (∞, ∞) in the simplicial cone being queried. The question of how to choose an appropriate arbitrary point arises again and instead of pursuing an answer, a new method is

introduced in the LEDA `range_tree` class:

```
void rt_GTquery(rt_item& Left, list<rt_item> result );
```

This method specifically handles dominance reporting queries (for the query point `Left`). The points dominating the given query point are returned in `result`, a list of range tree node items.

While `rt_GTquery` performs reporting queries, the weight approximation algorithm employs counting queries. Counting queries require storing and maintaining additional information in the internal nodes of the binary tree structure. For a subtree rooted at a node v , the size of the subtree and the sum of distances from the origin to those points dominating v needs to be stored in the node v and, when querying, this information needs to be reported for the right subtree when the search path goes left (§5.2). We now explore some design alternatives available for the implementation of the central data structure for the weight approximation. Figure 6.4 shows, conceptually, the simple binary tree and the corresponding internal nodes augmented with the sizes of the subtrees that we require for the implementation. Note that for clarity of presentation, the sums of distances are omitted from the diagram.

Figure 6.4: Simple internal node augmentation: size of subtrees

In LEDA's binary tree implementation, internal and external nodes are kept physically separate: internal nodes contain data that is used to guide searches in the tree while external nodes hold the data for retrieval. It is difficult to introduce auxiliary information into the internal nodes in LEDA binary trees and even more difficult to update this information. The design alternatives for augmenting LEDA nodes are to augment the external nodes, or to augment the internal nodes (see Figure 6.5). Not

all external nodes have a corresponding internal node for either alternative; specifically, the external node with the largest key does not have a corresponding internal node.

External node augmentation forces the question of how the auxiliary information is maintained without traversing entire subtrees to count nodes. Consider adding the node with key F into the tree. Counts must be updated for nodes B , D , E , and F , the nodes along the access path to node F . The access path to F from the root is at most $3 \lg i$, where i is the number of nodes in the tree, which implies that up to $3 \lg i$ additional accesses are required to update the external nodes corresponding to those nodes on the access path.

For internal node augmentation, we need only sum the sizes of a node v 's children to update the size of v ; however, the resulting sizes of the subtrees for the same keys vary from the external node and the simple augmentation (Figure 6.4) methods, which implies that the query algorithm may depend on the chosen method of augmentation.

Figure 6.5: Introducing auxiliary information into LEDA binary tree nodes

The LEDA range tree seems ill-suited for the extensions required for performing efficient counting queries.

6.2.2 Abandoning LEDA

Every great improvement has come after repeated failures. Virtually nothing comes out right the first time. Failures, repeated failures, are finger-posts on the road to achievement. - Charles F. Kettering

The decision to abandon the LEDA implementation of the distance approximation data structures is motivated by the difficulty in maintaining the internal node information in the range tree data structure. LEDA defines a `propagate_modifications` method that is called on tree restructuring, but documentation on using this method is scarce. The programmer needs to understand the internal operations of the inherited classes in order to provide a reasonable `propagate_modifications` method for the derived class. We feel that the effort required to achieve a working solution of the simplicial cone through derivation of LEDA classes is not reasonable for this project.

The advantages of reusing software through inheritance are numerous (*e.g.*, the approach is straightforward, it is simple to modify an existing implementation by overriding functions) but the advantages are shadowed by some disadvantages. One of the disadvantages is that the parent class imposes its physical representation on the derived subclass, resulting in a logical interdependence of the classes. Changes made to one of the classes might force or require changes in the other. Attempting to augment LEDA's range tree class, for example, requires modifying the implementation of the base class, which proves to be an involved task for the average programmer. The changes required to get a working implementation of the augmented data structure quickly become unmanageable. The failed attempt to implement the cone range search data structure through inheritance serves as a good example for which inheritance is probably not the best choice of reuse techniques. On the other hand, there does not appear to be a reasonable reuse solution of the LEDA range tree class through object composition or parameterization either.

In what proves to be an ambitious attempt to reuse existing software artifacts, a reasonable solution for implementing the data structures for the approximation algorithm is clouded by the intricacies of the LEDA library. The resulting solution is awkward, inelegant, and can not achieve the theoretical query time without either modifying the inherited subclasses or replacing those classes with something more

flexible and appropriate. The lessons learned by attempting to adapt LEDA's range tree are not lost, however, in the solution built from scratch, which is described in the next section. Specifically, the *built-from-scratch* approach focuses on:

1. Building from a more flexible, adaptable base class;
2. Using reuse techniques (class inheritance, object composition, and parameterization) appropriate to individual situations; and,
3. Building more specialized classes so that different concepts are separated, such as general coordinate systems and individual simplicial cones.

6.3 Building from Scratch: Preliminaries

The foundation data structure of range and persistent trees is the balanced binary search tree. In order to implement range and persistent trees, a complete and thorough understanding of the binary search tree data structure is required. The goal in developing the range search data structures is to start with a complete, comprehensive and stable version of the foundation data structure; in our case, the red-black tree.

The binary search tree assumes an ordering on the keys that are inserted into the tree. Recall that a simplicial cone is essentially a coordinate system of reference that induces an ordering on the points. We require a method for transforming the original points' coordinates from the Cartesian plane to a non-orthogonal coordinate system, which we refer to as the k -oriented coordinate system. Section 6.3.1 discusses how the concept of an arbitrary coordinate system is encapsulated in a C++ class. The following section explains the red-black tree data structure in detail and then provides a description of an existing implementation that is easily extendable.

6.3.1 k -Oriented Coordinate System

The Euclidean sum approximation algorithm imposes a non-orthogonal coordinate system on the point data. In this section, we describe how the behaviour of the coordinate system is implemented as a C++ class.

The two predominant operations carried out on the set of demand points, S , are:

1. Given a point $p \in S$ and a cone c_i , calculate the k -oriented coordinates $(t_i(p)[0], t_i(p)[1])$ of p .
2. Given two points, p and r , return the result of comparing them in the k^{th} orientation.

We have two options for implementing these operations under a coordinate system interface:

1. Use geometric transformations offered by the LEDA library; or,
2. Apply affine transformations (3.1) directly.

Since we continue to use LEDA's primitive geometric objects (*i.e.*, points and lines) and the logic for transformation from Cartesian coordinates to k -oriented coordinates is already implemented (from the previous section), we choose LEDA's geometric transformations for implementing operations in the `coordinateSystem` class. The `coordinateSystem` interface is implemented in such a way that, should a change be required, algebraic implementations of the affine transformations can easily replace the LEDA-based operations without affecting the `coordinateSystem` interface. The interface is outlined in Table 6.1.

A benefit of programming with LEDA's geometric transformations is that it leads to self-documenting code; that is, the LEDA functions have meaningful names, which lead to a straightforward understanding of the algorithms being implemented. Should the coordinate system implementation later be reused, the programmer does not need special knowledge of algebraic transformations, but instead of just the geometric concepts named as LEDA methods.

A `kCompare` function object is defined in the same file as the `coordinateSystem` class definition. It is initialized with a coordinate system object and a dimension in which comparisons should occur, and returns the result of comparing two points in the reference coordinate frame defined by the coordinate system object. The function object simplifies the process of sorting a list of points according to their k -oriented coordinates, such as when points are inserted into a persistent search tree data structure.

<i>function</i>	<i>description</i>
creation	<code>coordinateSystem(line axis1, line axis2);</code> Creates a new coordinate system defined by the axes <code>axis1</code> and <code>axis2</code> , which must be distinct from each other.
modify	<code>void set_axes(line axis1, line axis2);</code> Changes the axes of the coordinate system to <code>axis1</code> and <code>axis2</code> , which must be distinct from each other.
coordinate transformation	<code>point kCoords(const point &p);</code> Returns a the image of the Cartesian point <code>p</code> translated to in the reference frame defined by this coordinate system's axes.
coordinate comparison	<code>int kCompare(const point &a, const point &b, int dim=0);</code> Returns the result of comparing the coordinates of the Cartesian points <code>a</code> and <code>b</code> against the dim^{th} axis of this coordinate system.

Table 6.1: Class `coordinateSystem` public interface

6.3.2 Red-Black Tree

Red-black trees are one of the many balanced search tree schemes commonly presented in introductory data structures textbooks (e.g., [CLR90]). Balanced search trees guarantee that the basic dynamic-set operations take $O(\lg n)$ time in the worst case. This is achieved by encoding *balancing information* in the nodes of the search tree, $\mathcal{T}$. The tree is *rebalanced* using this auxiliary node information after inserting or deleting a node v into $\mathcal{T}$ by restructuring the tree along the access path.⁵ From an implementation point of view, one of the key attractions of red-black trees is that only one extra bit of information per node is needed to store the balance information and at most $O(1)$ rotations are ever needed for rebalancing the tree.

The red-black balancing scheme assigns each node in $\mathcal{T}$ a colour, either red or black, subject to the following constraints:

1. Every leaf node is black;
2. **Red Constraint:** if a node is red, both its children are black; and,

⁵This is the path in the tree $\mathcal{T}$ from the root to the inserted or deleted item v .

3. **Black Constraint:** every path from the root to an arbitrary leaf node contains the same number of black nodes.

Rebalancing is a bottom-up process in a red-black tree. On insertion, a new node v , containing the new item, is attached at the end of the search path in $\mathcal{T}$ and coloured red. This preserves the black constraint on $\mathcal{T}$; however, this may violate the red constraint. Deletion of a node, on the other hand, does not affect the red-constraint but may violate the black constraint.

When the red-black conditions are violated, they must be restored by changing the colours of some of the nodes in $\mathcal{T}$ and also by restructuring the tree. Restructuring the tree happens by changing the (child- and parent-) pointer structure through **rotation**. Rotation is a tree operation that preserves the in-tree ordering of nodes (see Figure 6.6).

Figure 6.6: Rebalancing the Red-Black Tree: Rotation

Performing a right-rotation, for example, on subtree rooted at a node x assumes the left-child y of x is not *NULL*. The rotation pivots around the link between x and y , making y the new root of the subtree, and reassigning y 's right-child and x 's left-child appropriately.

6.3.3 Extendable Implementation of a Red-Black Tree

It is possible to build a red-black tree by deriving from an existing binary tree implementation. An actual implementation can be complicated, however, by the many possible rotations, by the possibility that some subtrees may be empty, and by the root node, which has no parent. For these reasons, it is common practice to simplify red-black trees' implementation and operations by employing the following two sentinels [Wei00, p.676]:

1. A `nil` node is used in place of the `NULL` pointer. This node is always coloured black.
2. A root pointer is defined and points to the node that is the actual root of the tree. The parent of the node in the root position of the tree is set to the special `nil` node.

Since the range-tree and the persistent search tree are both based on a red-black tree, it is convenient to find an existing implementation of a red-black tree that is easily modified and extended. Morin [Mor] makes available an extensible implementation of a red-black tree in a design that makes it simple to derive subclasses that maintain hierarchical information at the nodes of the tree.

A common extension of a binary tree $\mathcal{T}$ is to introduce some sort of order statistic (*i.e.*, information based on the in-tree order of a node) into the nodes of the tree. For example, by keeping track of the size⁶ of each node v in $\mathcal{T}$, the k^{th} smallest element in the tree can be accessed without traversing the first k elements of the tree. If k is larger than the size of a node, then the k^{th} ordered node will be in the right-subtree. The size example can be generalized for introducing other order statistics.

Consider extending an existing implementation of a basic red-black tree that performs only the basic dynamic-set and rebalancing operations, class `basic_rbtree`. The augmented tree (class `aug_rbtree`) results from inheriting the implementation of the basic red-black tree. The extension requires that each node v in the augmented tree maintain auxiliary information, a `size` member, which stores the size of the subtree rooted at v . When the structure of the tree changes through insertions, deletions and rotations, so too will the sizes of some, but not necessarily all, internal nodes. It is important then, on insertion, deletion and rotation of nodes, to update the sizes of all affected nodes. In contrast to LEDA's implementation of binary search trees, where a physical distinction is made between external and internal tree nodes, our implementation makes the distinction only conceptually. This also will be more efficient, as updated information will already be in the processor cache for maintenance purposes.

⁶the size of a node v is defined as the number of descendants of v , including v itself

In principle, the maintenance of the introduced size variable seems simple enough. During an insert operation, each node on the path from the root to the insertion point gains one node in its subtree and, therefore, has its size increased by 1. Similarly, for the removal of a node, each node on the path from the root to the node that is removed loses one node from its subtree and has its size decremented by 1. The augmented tree can easily handle this maintenance by following the insertion path back to the root from the inserted node, as in the following example:

```
aug_rbtree::insert(k)
  1. v = base_rbtree::insert(k)
  2. while (v != root)
  3.   v.size ← v.size + 1
  4.   v ← v.parent
```

Restructuring operations, on the other hand, which are implemented in the base class, have no knowledge of the size variable and are therefore not capable of maintaining the sizes of subtrees. It is inefficient on updates to allow the parent class to perform the restructuring and then have the derived class try to fix the size values later.

[Mor] made some key observations about the behaviour of a balanced binary search tree subject to updates that result in more efficient updates of the tree. Morin's observations introduce a slight modification to the standard implementation of binary search trees (see [CLR90]). However subtle, these modifications make the balanced binary search tree (*e.g.*, the red-black tree) easier to extend than those offered by most data structures libraries.

The basic implementation of a binary tree defines the structure-modifying operations insert, delete, left-rotate, and right-rotate. The rotations only change the size values of two nodes. Insert and delete are slightly different in that they modify the tree and then restructure the tree through rotations, so the size values need to be corrected before the rotations occur in order for the sizes to update correctly on rotation. The general approach aggregates a number of pointer changes into one logical group for each operation. An augmented red-black tree design that uses inheritance as a reuse technique has the parent class perform the rotations, then the derived class update the auxiliary information of the two affected nodes. For example, if the basic red-black tree implementation has a left rotate method, `base_rbtree::leftRotate(y)`,

which performs the rotation depicted in Figure 6.6, then an augmented version of the same method might resemble:

```
aug_rbtree::leftRotate(y)
  1. x ← y.leftchild()
  2. base_rbtree::leftRotate(y)
  3. update(y)
  4. update(x)
```

where `update(v)` resets the size of node v by summing the sizes of v 's children. Note that the order in which the updates occur matters.

Morin identifies three different groups of pointer changes for inserting and deleting nodes that can be aggregated into similar logical structure-modifying operations: *attach*, *splice*, and *replace* (see Figure 6.7). Attach is tantamount to insert, while splice and replace make up the delete operation.

Figure 6.7: RB-Tree Pointer Changes: Attach, Splice, Replace

An augmented version of the red-black tree, which uses *attach*, *splice*, and *rotate* methods to implement the insertion and deletion of nodes, performs updates in a manner analogous to `leftRotate`, above. That is, for a given update operation, the affected nodes are saved, the corresponding base class update method is called, then the affected nodes are updated accordingly.

The augmented version of the red-black tree gains access to the update methods through a protected interface. Dynamic binding is used for update methods (*i.e.*, the methods are declared `virtual`). The methods in the protected interface are redefined as necessary in the derived class. Update methods in the red-black tree's protected interface are outlined in Table 6.2.

The public interface for the red-black tree (and its derivatives) is outlined in Table 6.3. Augmented versions of the tree might include other methods to reflect the additional functionality of the extended implementation.

<i>rbtree protected methods and description</i>
<code>left_rotate(rbtree_node *x)</code> <code>right_rotate(rbtree_node *x)</code> Performs a (left or right) rotation of the node <code>x</code> .
<code>attach(rbtree_node *x, rbtree_node *y)</code> Makes the single node <code>x</code> the left or right child of node <code>y</code> .
<code>splice(rbtree_node *y, rbtree_node *x)</code> Makes the parent of node <code>y</code> jump over it to the node <code>x</code> .
<code>replace(rbtree_node *z, rbtree_node *y)</code> Replaces node <code>z</code> with node <code>y</code> .

Table 6.2: Class `rbtree` protected interface

6.4 Persistent Search Tree

Enough differences exist between the (red-black) persistent search tree and the basic red-black tree that developing the persistent tree through inheritance of the `rbtree` class is not reasonable. The persistent tree is, however, modelled after the `rbtree` class and is developed by modifying the class directly. The classes `rbtree` and `rbtree_node` become `perstree` and `perstree_node`, respectively. To make the data structure persistent, we introduce the array data structure, a mechanism whereby consecutive roots of trees may be stored so that different versions of trees may be accessed.

The array is the basic data structure for storing a collection of identically-typed objects. An array can be declared and used in one of two basic ways: as a built-in array or as a vector. The syntax for both is similar; however, because the vector is a built-in *object*, it is easier and safer to use than the primitive array, which is a built-in *type*. The vector class, for example, is designed to maintain the integrity of the data structure subject to access and update operations, whereas the array built-in type requires the programmer to handle this maintenance. A data member `rootvec` of type `vector<perstree_node*>` is added to the `perstree` class and stores pointers to the root nodes of all versions of the persistent tree structure, as depicted in Figure 6.8. The data member `root`, carried over from the `rbtree` class, is used as a pointer to the root node of the most current tree only in the persistent structure.

<i>function</i>	<i>description</i>
creation	<code>rbtree();</code> Creates a new red-black tree for scalar data.
insertion	<code>rbtree_node * insert(scalar d);</code> Inserts the data item d into the tree. Insertion of duplicate items creates duplicate nodes.
search	<code>rbtree_node * search(scalar d, unsigned t);</code> Returns a reference to the smallest element in the (t^{th}) tree greater than or equal to d , or <i>nil</i> if no such element exists
node value	<code>scalar value(rbtree_node *x);</code> Returns the value of node x .
clear	<code>void clear();</code> Clears (deletes) all nodes from the persistent structure.

Table 6.3: Class `rbtree` public interface

The red-black tree is made persistent by using path copying. A review of the `rbtree` class reveals that the `basic_insert()` function, which handles the low-level insertion of nodes into the tree, is the only `rbtree` method that needs to be modified to render the red-black tree persistent on insertion. In addition, the code modifications to do this are quite minor: the basic insertion algorithm remains the same, with the exception that each node in the current tree on the path from the root to the insertion point is copied. The branches of the current tree that are not followed by the insertion algorithm are linked to by the nodes along the newly-created path. Figure 6.9 shows two consecutive versions of a persistent search tree and their equivalent simple binary search trees. The shaded nodes depict the newly-copied path after the insertion of the node with key E . After a path-copy, some nodes are linked to by more than one parent node. Double-headed arrows depict reliable child-parent relationships, while single-headed arrows depict only reliable child-node information. Only the most current tree's parent pointers can be trusted. Additionally, the balance information in the nodes can only be trusted for the most current tree.

Maintaining both child and parent pointers is not possible if the theoretical creation bounds of the persistent search tree are respected. If pointers to parent nodes

Figure 6.8: Rootvec: vector of tree-node pointers to successive roots

Figure 6.9: Two consecutive versions of a persistent search tree

are maintained for all versions of the tree, additional node copying would result from persistent tree update operations. Consider the insert operation. Each insert would require $\Omega(m)$ time and space as opposed to $\Omega(\lg(m))$ time and space per insert, where m is the number of nodes in the current tree. The rationale is as follows: if one of the child pointers of a node v changes, as for all nodes along the copied access path, v itself must be copied. All nodes for which child information does not change instead have parent information change, and so those nodes are (recursively) copied, resulting in the copying of every node in the tree.

Copying a node is complicated by the special case of the `nil` node. A `copy_node()` function is introduced into the `perstree`. Its purpose is to copy nodes, in much the same way that the `new_node()` function creates nodes. Since the `perstree_node` class has no knowledge of the `nil` node, the `copy_node` function is implemented in the `perstree` class to check against the special `nil` case before calling the copy constructor on the node to be copied.

Other than the `basic_insert` method, methods implementing update operations do not need to be modified, as updates happen in the most current tree only. As

long as the root variable points to the most current tree and the parent and child pointers for each node are correctly assigned in the current tree, no code modifications between the `rbtree` and the `perstree` versions are required.

Since we only maintain balance and parent-link information in the most current tree, we need to observe which persistent tree functions are access methods and which are update methods. The access operations must not attempt to follow the parent link of any node in an access operation nor call any other method that does the same. Nodes that access their parent link include the following member functions:

1. `update patterns`: `left_rotate`, `right_rotate`, `splice`, `attach`, `replace`. These update patterns are called only on update operations, which apply to the most current version of the tree only.
2. `insert_node`: update operation on most current version of tree.
3. `successor`: called by `remove()`, an update operation.
4. `remove_fixup`: update operation on most current version of tree.
5. `clear`: works on all versions of tree. Must be rewritten.

Since the approximation algorithm only adds and does not remove data from the persistent structure, we disable the remove operation. It is a little more complicated than the insert with respect to path copying using update patterns. Because the remove function is disabled, the `remove_fixup()` and `successor()` functions do not need to be modified (nor enabled) at this time.

With the insertion of data implemented and the removal of nodes disabled, the focus turns to a method for searching the persistent tree structure. [ST86] suggest that persistent data structure access operations take both a key x and a time t value, so that a search involves finding and returning an item in the persistent set at time t with greatest key less than or equal to x . If no such item exists, a null item, the `nil` node, is returned. A difference between this general description in the literature and the implementation for this thesis is that `search(x, t)` returns the item at position t with *smallest key greater than or equal to x* .

The final operation implemented for the persistent search tree data structure is the `clear()` operation. This operation is absolutely necessary for avoiding memory

leaks; that is, the memory that is dynamically allocated for the creation of nodes in the trees must be returned to the heap when the persistent tree structure is cleared or deleted. Objects that are created with the C++ `new` operator, such as nodes in the persistent search tree, must be explicitly deleted using the `delete` operator.

A challenge of clearing a persistent data structure is that some links point to the same node; that is, each node in the persistent structure potentially has more than one parent node, however, each physical node has only one parent pointer. Parent pointers can therefore not be trusted for accessing the nodes in an arbitrary tree in the persistent structure. The entire structure needs to be cleared without making a `delete` call to the same address twice, so it is necessary to decide on some mechanism to keep track of which nodes have been deleted and which nodes remain successive trees in the persistent structure are traversed. There are two options that present themselves as prominent candidate solutions to this problem:

Nodes contain an extra data member The member `incount` of a node maintains the number of nodes that are linked to it (*i.e.*, the parent count of the node). Each call to `delete` a node v decrements v 's `incount` by one. When the count reaches 0, the `delete` operator is called to remove the node from the heap. The LEDA point object uses a similar idea. A node's `incount` data member must be adjusted on insert and remove operations in the persistent tree.

A list of nodes for deletion is created The `clear()` method maintains a list of addresses to be deleted. A set container type is used to keep track of objects that need to be returned to the heap, since sets do not allow duplicate items. All trees of the persistent structure are traversed, with their nodes being added to the set. Since the set does not store duplicate entries, a subsequent call on each item in the set ensures that the `delete` operator is invoked only once for each node to be deleted. No additional overhead is required in the `perstree_node` class or in other `perstree` class methods.

`Clear` is implemented using the second solution. Recall that parent pointers are unavailable for use in tree traversal in any version of the persistent tree but the most current; therefore, a mechanism whereby path history is maintained is required. This

can be achieved by using recursive calls to the clear method, but recursion renders the set difficult to maintain as a local variable; or, by using *pseudo*-recursion, where a stack container type is employed for storing the visited path. The implementation adopts the stack approach for saving the traversal history.

The public interface of the basic persistent search tree class is outlined in table 6.4.

<i>function</i>	<i>description</i>
creation	<code>perstree();</code> Creates a new persistent tree for scalar data.
insertion	<code>perstree_node * insert(scalar d);</code> Inserts the data item d into the tree. Insertion of duplicate items creates duplicate nodes.
search	<code>perstree_node * search(scalar d, unsigned t);</code> Returns a reference to the smallest element in the (t^{th}) tree greater than or equal to d , or <i>nil</i> if no such element exists
node value	<code>scalar value(perstree_node *x);</code> Returns the value of node x .
clear	<code>void clear();</code> Clears (deletes) all nodes from the persistent structure.

Table 6.4: Class `perstree` public interface

6.4.1 Extending the Persistent Data Structure for our Needs

In the spirit of writing robust, re-usable code, the basic persistent tree is implemented with extensibility in mind. In the last section, an implementation of a basic red-black tree was transformed into a persistent red-black tree, resulting in a simple persistent data structure that uses update patterns and is easily extended for our needs.

The basic persistent tree is augmented to include the sizes of subtrees as information in its nodes. The augmented persistent tree is implemented by the classes `aperstree` and `aperstree_node`. In a similar way that the basic persistent search tree is modelled after the basic red-black tree, the extended functionality in the augmented persistent tree class bears resemblance to that of the augmented red-black tree class. The inheritance structure and extensions are alike for the augmented persistent

tree and augmented red-black tree and result in the augmented persistent tree having the capability for storing both leaf and inner-node information and for performing inner-node calculations automatically on updates.

Recall from Section 5.2.2 that persistent trees are suitable for implementing dominance queries on a set S of points. For a persistent tree $\mathcal{T}$, $\mathcal{T}_j$ denotes the version of the tree resulting from the j^{th} update operation. For a coordinate frame defined by axes t_i and t_{i+1} and points $p_m \in S$ sorted according to their t_i -coordinates, the points p_m are inserted into $\mathcal{T}$ such that $\mathcal{T}_j$ represents all points in S with t_i -coordinate larger than p_j , the j^{th} -ordered point in S . The first or t_i -coordinate of p_j is the key of $\mathcal{T}_j$. The $p_m \in S$ are stored in $\mathcal{T}$ according to their t_{i+1} -coordinate.

The persistent tree is extended to accommodate dominance queries of points. The general persistent tree distinguishes between versions $\mathcal{T}_b$ and $\mathcal{T}_c$ only by their indices b and c , which are integers. A specific version of the persistent tree $\mathcal{T}_b$ is accessed by indicating its position b in the `rootvec` data member, the vector of roots. The augmented tree needs to associate the index b of $\mathcal{T}_b$ with $\mathcal{T}_b$'s key. This association is implemented with a map data structure, which stores key-index pairs and provides for the fast retrieval of an index based on the key.

The changes required in the `aperstree` class to accommodate the association between keys and versions of the persistent tree include introducing a map data member, `indexMap`, and modifying the insertion and search methods to use this new data member.

Consider the insertion of a new point into the augmented persistent tree. The insertion keys of the point are a key1-key2 pair representing the t_i and t_{i+1} coordinates of the point, respectively. The second key is first inserted into the persistent tree structure $\mathcal{T}$, which results in the creation of a new version of the persistent tree, rooted at the last entry in `rootvec`. The index of this newest entry in `rootvec` is obtained and inserted into the `indexMap` data member as the key2-index pair. The structure of the map and root vector persistent data structure is depicted by Figure 6.10.

The main challenge faced in using a map for the implementation of the augmented persistent tree is in performing the first-level search for the query function. The map data structure that is part of the C++ Standard Template Library (STL) does not

Figure 6.10: Persistent tree supporting dominance queries on points

have a useful `find()` operation suited to our search needs since the STL map returns a null value if the key is not found, whereas we require a map that returns the node with the closest value (*i.e.*, largest value smaller than the search key). The `rbtree` class is easily augmented as a map with this functionality. The map is implemented with similar functionality to the STL map, with the exception of the `find` function. The interface is described in Table 6.5.

<i>function</i>	<i>description</i>
creation	<code>myMap();</code> Creates a new map for (key:info) = (scalar:int) pairs.
insertion, update	<code>int& operator[] (const scalar& k);</code> Search for k. Return value at k if found; otherwise, make a new map entry and return the default value 0.
search	<code>const scalar find(const scalar& k);</code> Search for k; return the smallest key that is not less than k.
clear	<code>void clear()</code> Clears all entries from the map.

Table 6.5: class `mymap` public interface

After the map data member `indexMap` is introduced, searching the two-key persistent search tree is easily implemented. The search is called with two parameters (keys). The first parameter is used as a key into `indexMap` for retrieving the index j into `rootvec` where the tree $\mathcal{T}_j$ corresponding to the first parameter is rooted. A

<i>function</i>	<i>description</i>
creation	<code>aperstree();</code> Creates a new persistent tree for (scalar,scalar) data pairs.
insertion	<code>aperstree_node * insert(scalar k1, scalar k2, scalar i);</code> Inserts the data item with keys $(k1, k2)$ and information i into the tree. Insertion of duplicate items creates duplicate nodes.
search	<code>aperstree_node * search(scalar k1, scalar k2);</code> Search the persistent structure for key-pair $(k1, k2)$. The first key finds the index of which tree to search, the second key finds the matching node within that tree.
query	<code>aperstree_node* GTquery(scalar k1, scalar k2, int &n, scalar &infosum);</code> Query the persistent structure for key-pair $(k1, k2)$. Return in n the number of nodes greater than or equal to $(k1, k2)$ and in $infosum$ the sum of the information for all nodes greater than or equal to $(k1, k2)$.
node value	<code>scalar value(perstree_node *x);</code> Returns the value of node x .
clear	<code>void clear();</code> Clears (deletes) all nodes from the persistent structure.

Table 6.6: Two-key persistent search tree: class `aperstree` public interface

search is carried out on $\mathcal{T}_j$ using the second parameter as a search key k . The search ends at the smallest element in $\mathcal{T}_j$ tree with data greater than or equal to k is returned, or if no such element exists, `nil` is returned.

The augmented persistent tree public interface is described in Table 6.6.

6.5 Range Tree

The range tree implementation (classes `rangetree` and `rangetree_node`) bears many similarities to the persistent search tree implementation. Both have almost identical user interfaces, with the exception of the range tree having additional `insert` and `GTquery` methods that take an array of keys instead of individual keys as parameters.

The range tree is implemented with the general d -dimensional case in mind; that is, instead of having two separate data members in the node for storing the keys individually, the range tree stores an array of keys. The array is allocated dynamically (from the heap) when new keys (or nodes) are inserted into the tree and then released to the heap when the node is deleted.

One of the design goals in implementing the range search data structures from scratch is to provide both with identical interfaces. This approach makes it simple to perform comparative testing of the data structures. A program can be written that uses only those methods that are common to both the range tree and the persistent search tree, then each data structure can be tested in turn by *plugging it into* the program.

Originally, the augmented range tree is implemented to mirror the behaviour of the extensible red-black tree and the persistent search tree in that a base class encoding the general behaviour of the tree is implemented. An augmented version of that tree is then created through inheritance and dynamic binding. The implementation of the range tree using this approach remains incomplete due to complications resulting from casting difficulties with dynamic binding. The source of the difficulties is the pointer member in the `rangetree_node` class that stores the secondary (associated tree) structure. Treating the pointer as the appropriate class type (*i.e.*, `arangetree_node`) proves challenging to the average programmer. Introducing a void pointer instead of a tree pointer might be a suitable *workaround* solution for this problem.

No additional constructors, which result in overloading the constructor, need to be introduced for the `rangetree_node` class; however, a dimension parameter is added to the constructor for `rangetree_node` to allow for the proper allocation of memory for the keys as dynamic (heap) memory is used to store the keys.

Chapter 7

Implementation and Testing of $w_k(p)$

The purpose of computation is insight, not numbers. - Richard Hamming

One of the goals for this thesis is to compare an approximation scheme against the naïve approach for computing the weight $w(p)$ of a point p with respect to a set S of n points in the plane (1.1). The approximation scheme is based on performing dominance queries of the point set S , as described in Chapter 4, and is denoted $w_k(p)$.

Two range search data structures are implemented for performing the dominance queries on S and computing an approximation to $w(p)$: the range tree and the persistent search tree. The approximation computes (1.1) to within a factor of $1/\cos \frac{\theta}{2}$ in $O(k \log n)$ time, where θ is an arbitrarily small constant dependent on k . This is done by preprocessing the points in S in $O(kn \log n)$ time and space. The implementation of the range search data structures is discussed in Chapter 6.

This chapter concentrates on comparing the range search data structures, and focusing on their effectiveness and efficiency in approximating $w(p)$. The analysis and comparison of the performance of the range search data structures versus the naïve approach for computing the weight necessitates the use of careful experimentation and statistical techniques. Specifically, we use statistics to draw conclusions relating to the following questions:

1. What effect does k have on the accuracy of the approximated weight function (compared to the value computed by the naïve approach)?
2. Is the point distribution of S a significant factor contributing to the measured response time for a query?
3. Is the observed average query time proportional to the theoretical query time? (i.e., do range tree queries exhibit time proportional to $\lg^2 n$? persistent trees in $\lg n$? does the naïve approach exhibit query time linear in the size of the input?)
4. Is the observed creation time proportional to the theoretical creation time?
5. Is the cost of preprocessing recovered through multiple queries on the same point set? If yes, for what conditions is the cost recovered?
6. Which data structure, range tree or persistent search tree, exhibited the best query time?
7. At what point does the space used in preprocessing overwhelm the performance benefits?

The rest of the chapter is organized as follows: Section 7.1 gives an overview of designing experiments for statistical analysis, followed by a section describing the implementations of the $w_k(p)$ approximation scheme and the *benchmark*¹ program used for collecting data. Section 7.3 discusses the generation of random input samples and is followed by a section describing the approach taken for collecting data. The section ends with an interpretation of the experimental data, focusing on drawing conclusions to answer the questions stated just above.

¹A benchmark is a standardized (problem or) test that serves as a basis for evaluation or for comparison.

7.1 Experiments and Analysis

Performance is a key criterion in designing just about anything. Every-day decisions are often based on the perceived *best* alternative or solution to a problem. When the best choice is not clear, experiments are performed to discover something about the system and statistical methods are used to interpret the experimental results. Literally, an **experiment** is a test, or a series of tests, carried out under controlled conditions in order to study the performance of a system. The *system* we are testing in this thesis is the distance approximation algorithm.

An experimental problem has two aspects: the design of the experiment and the statistical analysis of the data. The two aspects are related, since the method of analysis depends on the design employed [Mon97]. A number of steps are involved in designing experiments so that the generated or collected data can be analysed by statistical methods. The use of statistics allows us to be confident that the results will be replicable and are not due to random chance.

Many steps are involved in the design of an experiment, starting with stating the goals of the experiment and defining the system on which the experiments will be conducted. Possible outcomes of the experiment might also be identified. Once the system is defined, the criteria for comparing the performance is selected; for example, the distance approximation scheme will be compared against the naïve approach with respect to the speed of evaluation of the weight of a point and the accuracy of the approximation. These comparison criteria are termed the **response variables** of the experiment; they are a measured response of the system.

The objective of conducting an experiment is to determine the influence that factors chosen by the experimenter have on the response of the system. It is necessary to identify parameters that might affect the behaviour or performance of the system being studied. The **factors** of the experiment are the parameters that are varied in the study, and hopefully are those which might be considered to have a significant impact on the performance of the system. It is also valuable to show that a given parameter have no effect on the system. Two kinds of factors are identified:

Quantitative These variable have levels that are expressed numerically, either discrete or continuous.

Qualitative or Categorical These variables have states, levels, or categories that are defined by a set of mutually exclusive and exhaustive subclasses.

We can identify both quantitative and qualitative factors for the comparison of the sum-of-distance schemes. For example, the number of cones (represented by k) and the number of points in the data set S are both quantitative factors that can be varied in the experiment. The distribution of the points in the set S and the range search data structure used in the distance approximation scheme, either range tree or persistent search tree, are treated as qualitative factors in the experiment. The different values that a factor can assume are called levels. Each factor level constitutes one alternative for that factor. In addition, the upper and lower bounds of a level for a particular factor is the **range** of that factor. Table 7.1 summarizes the variables for testing the sum-of-distance schemes.

Response Variables	
Creation time	
Query Time	
Computed Distance	

Factors (Predictor Variables)	
Quantitative	Qualitative
k : number of cones	Point distribution in S
n : number of points in S	Data structure (levels: range, persistent, naïve)

Table 7.1: Variables for Weight Computation

After the response variables and factors are identified, the goal is to determine the relative effect of the various factors by designing and defining experiments that offer maximum information with minimal effort. The number of experiments required to collect an appropriate amount of data for drawing conclusions is a function of the number of factors and their levels. It is unreasonable in many cases to test all combinations of all factors at all their levels (*i.e.*, a **full-factorial** experimental

design). Suppose, for example, that for the factors identified above, each qualitative factor has 3 levels and each quantitative factor has 10 levels, then $3 \times 3 \times 10 \times 10 = 900$ experiments are required to test all combinations of factors.

Most performance measurements result in a random quantification of the performance; that is, the outcome might be different each time the experiment is repeated. Therefore, to conduct a meaningful statistical analysis of the output, each experiment is run more than once, or **replicated**; a *minimum* of 30 replications is recommended for many statistical tests. Continuing with our example, this results in 27000 individual observations to consider in the analysis. For an integral-valued variable such as n (the number of points in the set), there may be many more than 10 levels! Often the testing can be decomposed into several phases using a smaller or simpler design, such as a **fractional-factorial** design, rather than using one combinatorial design with too many factors at too many levels. More detail can be found in the many textbooks devoted to the design and analysis of experiments (*e.g.*, [Jai91, Mon97]).

Once the data is collected, it needs to be summarized to provide meaningful information about the system being tested. There are several ways to summarize data. Most commonly, the performance is characterized by at least two traits: the central tendency of the observations and the dispersion or variability of those observations. Characterizing the central tendency of a set of data is achieved by specifying the mean, median or mode of the data, while variability can be characterized through the range of values observed or the variance or standard deviation of the observations.

The purpose of summarizing the data with statistics is to make a statement about the range in which the responses or behaviour of the majority of systems would lie. A definite statement cannot normally be made about all systems (*i.e.*, **population**) but a probabilistic statement can be made in terms of confidence intervals on the collected (*i.e.*, **sampled**) data and then conclusions can be drawn. The goal of the analysis is to estimate the population characteristics from a sample of observations rather than to compute exact population parameters; for example, to *determine* the population mean from a set of samples would require replicating the experiment infinitely many times, which is not possible.

7.2 Benchmark Program: Computing the Weight

The statistical analysis of the distance approximation scheme for a set S of n points attempts to characterize its behaviour in terms of:

1. the creation times of the data structure for computing the weight $w_k(p)$ of a point p with respect to the set S ;
2. the query times for computing the weight; and
3. the computed value of the weight.

This section describes the implementation of the computation of the weight. Special care is taken in the implementation to accommodate recording any performance measures required in the analysis.

One of the implementation goals of the $w_k(p)$ approximation scheme is uniformity among the different approaches. In other words, the solution is coded in such a way that the range search data structure could be passed as a parameter to a class without requiring knowledge of the internal structure of that class. We therefore attribute significant differences in run-time performance to the range search data structures and not to the benchmark program.

The benchmark we use for testing is computing the weight of a point p with respect to a set S of n points. The class that implements this is `testSum`. It is a template class, taking as a parameter a class that computes the weight function. The public interface (see Table 7.2) for this `testSum` class includes modules for:

1. Creating the query structure, given a set of points and number of cones.
2. Querying the structure. There are three different options.

The classes that implement the computation of the weight of a (query) point with respect to a set of points are: `euclideanSum`, `rangeDistApprox` and `persistentDistApprox`. The `euclideanSum` class computes the weight of a point directly, using the following *naïve* approach, which takes linear time in the size of the point set:

<i>Function</i>	<i>Description</i>
Creation	<pre>template<class DS> testSum(list<point>& pList); template<class DS> testSum(list<point>& pList, unsigned nCones);</pre> <p>Creates an instance of the benchmark. <i>pList</i> is a set S of n points, which will be queried and <i>nCones</i> is an optional argument, required when the template parameter <i>DS</i> is one of the range search structures (<i>euclideanSum</i>, <i>rangeDistApprox</i> or <i>persistentDistApprox</i>), and specifies the value for k (number of cones) in the approximation scheme.</p>
Querying	<pre>scalar query(const point& q;)</pre> <p>Queries a point q. Returns the <i>weight</i> of q: the sum of q to p_i for all $p_i \in pList$.</p> <pre>scalar query(const point& q, s64& time);</pre> <p>As for previous query, except that the query is timed and the number of clock ticks required to perform the query is returned in <i>time</i>.</p> <pre>void query(const list<point>& qList, s64& time);</pre> <p>One query is performed for each point in <i>qList</i> and the total number of clock ticks required to perform those queries is returned in <i>time</i>.</p>

Table 7.2: Benchmark: Class *testSum* public interface

1. $sum = 0$
2. for all p_i in $pList$
3. $sum = sum + \sqrt{p_i.x^2 + p_i.y^2}$
4. return sum

This sum could easily be computed as a stand-alone function, and called whenever the sum needs to be computed. The computation is, however, wrapped within a class interface to conform to the same model as the sum approximation data structures. This allows queries of the *naïve* approach to be processed and timed in the same way as the approximation approaches.

Class *sumDistApprox* is not intended as a public interface for the user, but instead as an interface for the programmer to aid in the creation of specialized distance approximation classes. It is a template class and takes one of the range search data structures, *rangetree* or *aperstree* (see Chapter 6), as a parameter to create the

corresponding distance approximation structure. The following classes are provided for the user: `rangeDistApprox` and `persistentDistApprox`. These are *wrapper*² classes that derive from the `sumDistApprox` template class and instantiate a range tree instance and persistent tree instance of `sumDistApprox`, respectively.

The `sumDistApprox` class requires that the range search data structure implement the following dominance query method:

```
scalar GTquery(scalar k1, scalar k2, int &n);
```

where k_1 and k_2 are the coordinates of the query point, the size of the query is reported in n and the return value is the sum of distances from the query point to all points greater than that point in the given coordinate system; that is, the result is a dominance query on the tuple $p_i = (k_1, k_2)$.

Recall from Section 5.2 that the rules for inserting points into a range tree and into a persistent tree differ. As a result, a *function object*³ describing the rules for inserting points into the range search data structure is required as a parameter for the `rangeDistApprox` and `persistentDistApprox` classes on instantiation. The `rangeCone` and `persistentCone` function objects, respectively, insert points from a given list of points in the 2-dimensional plane into the *simplicial cone* (range search data structure). The construction of each cone is based on a given coordinate system, which induces a sort-order on the points.

7.3 Generating Random Point Sets

To test the distance approximation algorithm, we could define a set of points on which the algorithm should operate. This could be a long process if we choose the points arbitrarily and want to create large point sets. In addition, we might question how to choose the points in such a way that they demonstrate the robustness of the algorithm. Ideally, we could define a set of rules, specifying a given distribution

²A *wrapper class* is an interface class that controls access to another class or adjusts its interface [Str97].

³Simply speaking, a *function object* is an object that acts like a function, using the `()`-operator. A function object is more robust than a free-standing function because its object can hold data. In addition, the class can provide operations for initializing and extracting such data [Str97].

of points, and have the computer generate the set of points *automatically* using a random number generator.

Historically, random number generation was carried out by hand (*e.g.*, throwing dice, drawing numbered balls from an urn). Mechanized devices emerged in the 1930s, followed by electric circuit-based methods [LK82]. Only with the growing popularity of computers in the 1940s and 1950s did arithmetic (computational) methods of random number generation come into consideration.⁴

Generating a random number implies generating a numerical value in accordance with a specified probability distribution. A **random variable** is a variable that takes on such a value. Random-number generators used in computing are actually completely deterministic: given a starting value, called a *seed*, and a generating function, f , the i^{th} random number, r_i , in a random sequence is generally computed as a function of one or more of its predecessors in the sequence (*e.g.*, $r_i = f(r_{i-1})$, where $r_0 = \text{seed}$). Two sequences of random numbers that are initialised with the same seed will produce exactly the same stream of random numbers.

The numbers produced by a random number generator should appear to be distributed uniformly on $[0, 1]$ and should not exhibit any correlation with each other. It is not easy to design a random number generation scheme that produces numbers which appear to be independent draws from the uniform probability distribution. There has been much research devoted to this problem and it is recommended in practice to use an existing generation scheme that has undergone statistical testing to support its “randomness”.

In some situations, it might be desirable to reproduce the same random stream exactly. For example, suppose we want to repeat an experiment using the same (generated) point set, then we would like our experiment to be **repeatable**. By allowing a seed to be specified, a program implementing an experiment can be written in such a way that the results may be reproduced exactly.

LEDA offers generators for points and sets of points, including the ability to generate points within a geometric shaped area of a specified size, such as a square or a circle. When points are randomly generated inside a specified region, they have

⁴Random numbers generated through arithmetic methods are often called *pseudorandom* numbers in computing literature because of their deterministic nature; however, in the context of computing, *random* and *pseudorandom* are understood to mean the same thing.

a certain probabilistic distribution. The LEDA point generators, for example, return points that are uniformly distributed inside a specified region; that is, each potential point has an equal probability of being generated. LEDA does not support non-uniform generation of point data; however, this is not critical because other probabilistic distributions can be obtained by transforming independent uniformly distributed random variables. For example, a convenient way to generate a random point with normal⁵ probability distribution is with the polar method for generating normal random points. This method generates random variables in pairs as follows [Knu98]:

1. Obtain $U_1, U_2 \in [0, 1]$
2. do
3. $V_1 = 2U_1 - 1, V_2 = 2U_2 - 1$
4. $S \leftarrow V_1^2 + V_2^2$
5. until $S < 1$
6. $X_1 = V_1 \sqrt{-2 \ln S/S}, X_2 = V_2 \sqrt{-2 \ln S/S}$

LEDA's approach to point generation does not provide the user with a clear interface for specifying a random seed specifically for the point generators. This implies that random point set generation with LEDA might not be repeatable. LEDA does, however, offer a clear interface to the *random source*, which provides an unbounded stream of integers in a specified range or, when assigned to a double variable, provides a stream of values v_i normalized to the unit interval (*i.e.*, $v_i \in [0, 1]$).

Using LEDA's random source as a base, the class `pointGenerator` is implemented as a template class for generating random point sets for the testing purposes of this thesis. The template parameter for this class defines the rule for generating the point data. The header file, `myrandom.h`, not only provides the `pointGenerator` class, but also the following function objects for defining the distribution rule on the generated point set:

⁵The Normal Distribution is also known as the Gaussian Distribution.

uniformInUnitCircle generates single points or lists of points that are uniformly distributed inside the unit circle; that is, $p(x, y)$ is generated such that $-1 \leq x \leq 1$ and $-1 \leq y \leq 1$.

uniformInUnitSquare generates single points or lists of points that are uniformly distributed inside a unit square; that is, $p(x, y)$ is generated such that $0 \leq x \leq 1$ and $0 \leq y \leq 1$.

normalDistribution generates single points or lists of points that are normally distributed. The distribution is not normalized to the unit circle. $p(x, y)$ is generated such that 99.985% of the points are contained in the circle centered at the origin with radius 3.4.

Function	Description
Creation	<pre>template<class DISTR> pointGenerator(); template<class DISTR> pointGenerator(const unsigned seed);</pre> <p>Creates an instance of the pointGenerator function object. The seed is provided to initialize the random source. The template parameter DISTR is one of the function objects that implements point generation under a specified distribution rule (i.e., uniformInUnitCircle, uniformInUnitSquare, or normalDistribution).</p>
Querying	<pre>void operator()(point& p);</pre> <p>Outputs a point p with distribution probability according to the template parameter DISTR supplied on creation.</p> <pre>void operator()(unsigned n, list<point>& L); void operator()(const unsigned seed, unsigned n, list<point>& L);</pre> <p>Outputs a set of points in L with points distributed according to the template parameter DISTR supplied on creation. When a seed is specified, the random source is reset with that value before generating the points.</p>

Table 7.3: Class pointGenerator public interface

7.4 Automated Testing

The number of combinations of factors is proportional to the number of test programs we require for studying all combinations of factors. It is unreasonable to write this many individual test programs; however, through encapsulation and templatization, the number of original source code files can be reduced and still produce an effective number of test programs. This section discusses such an approach.

An inventory of the software created for the testing reveals three data structures for evaluating the sum of Euclidean distances: `rangeDistApprox`, `persistentDistApprox`, and `bruteForce` (see Section 7.2), and three classes of randomly-distributed data: uniformly distributed inside the unit circle, uniformly distributed inside the unit square, and normally distributed (see Section 7.3). Combining each data structure with each class of randomly distributed data leads to nine independent test cases, covering the qualitative factors of the experiment (Table 7.1).

The interface for invoking each test case is similar for both accuracy tests and timed tests. For example:

```
testname numCones numPoints numQueries randSeed outputFile
```

where `testname` is the name of the binary file (`normalBrute`, `normalPers`, `normalRange`, `uniCrcBrute`, `uniCrcPers`, `uniCrcRange`, `uniSqrBrute`, `uniSqrPers`, `uniSqrRange`) and the parameters are assigned appropriate values, according to the descriptions in Table 7.4.

With the parameters defined, the test suite follows the following steps:

1. Set the initial seed for the `pointGenerator` to `randSeed`.
2. Generate two lists of points: *pList*, a list of `numPoints` points for creating the distance approximation structure, and *qList*, a list of `numQueries` points for querying the structure - the point distribution is hard-coded.
3. Create the query data structure (`euclideanSum`, `rangeDistApprox`, or `persistentDistApprox`) from the first list of points, *pList*.
4. For each query point *q* in *qList*, perform a query on the distance data structure.

numCones	The number of simplicial cones for the approximation algorithm. This is a dummy parameter for the brute force algorithm.
numPoints	The number of points on which the query operates, according to the hard-coded distribution of points.
numQueries	The number of queries to perform. This actually turns out to be the number of replications on each of the nine qualitative test cases.
randSeed	The value for initialising the random seed at the onset of the test execution.
outputFile	The name of an output file for capturing results.

Table 7.4: Description of test parameters

A Linux script is defined to invoke all the tests, making it simple to call each of the nine qualitative tests with various combinations of levels of the quantitative factor (number of points, number of cones) for the experiment.

The Linux script allows the user to define the levels for the number of points and for the number of cones in files to be passed to the script, with each level declared on a separate line. For each combination of the specified levels, the script calls each of the nine qualitative tests.

The binaries for the tests are generated using a parameterized approach. The test is first written in C++, using the parameters outlined in Table 7.5 for representing which point distribution and data structure to use. The test file is given the extension `.template` (e.g., `testname.template`) and a makefile is used to generate the source code. The makefile invokes a sed script to replace the parameters in the `testname.template` file and generate test-specific source code, which is then compiled. Running the sed script effectively does a search-and-replace on the parameters in the template file, exchanging the terms with valid classes or constructs for compilation.

Parameter	Replace With
POINT_DISTRIBUTION	normalDistribution, uniformInUnitCircle, uniformInUnitSquare
DISTANCE_EVALUATOR	euclideanSum, rangeDistApprox, persistentDistApprox
DSLABEL	For output purposes, a label to identify which data structure is used.
DISTRIBLABEL	For output purposes, a label to identify which point distribution is used.

Table 7.5: Parameters for generating source code

7.5 Analysis of the Test Output

The goal of this section is to summarize the data collected from the actual behaviour of the Euclidean sum approximation algorithm and to answer the questions posed in the introductory section of this chapter. The behaviour of the algorithm is characterized in terms of the creation and query times of the data structures implementing it, and also in terms of the accuracy of the approximated sum with respect to the computed Euclidean sum (see algorithm on page 81). The algorithm is run on randomly generated points with three different methods for computing the sum: naïve, persistent tree, and range tree.

For the aggregated timing tests, the time for an individual query is computed as a mean; that is, we divide the total time, t_{Query} , to perform one set of independent queries by the number of queries performed, numQueries . Means alone are not normally sufficient to statistically refute candidate hypotheses, but individual query times may be subject to machine and operating system effects, which could add significant variability to the observations without much illuminating the topic at hand. Instead, we will validate the replicability of our results by rerunning tests for a single set of experimental factors to determine the variability of our mean query time estimate. We choose a large number of random queries to get an accurate value for the estimated mean query time.

In some situations, it is useful to evaluate whether there is a significant difference between at least two means in a set of data for which more than one mean can

be computed. A common statistical procedure for analysing such a situation is the analysis of variance and is outlined in the next section. The procedure applies to analyses in the sections that follow.

The observed data are collected by programs written in C++ and compiled with the gnu C++ compiler with optimization set at level 3. The programs are run on a Pentium III 60MHz processor running Linux Mandrake 7.0 with 256Mb of RAM.

7.5.1 Overview of the Analysis of Variance (ANOVA)

An **analysis of variance** (ANOVA) is used in situations where more than two populations or treatments are being compared. Single-factor ANOVA is used when the characteristic that differentiates the treatments is just one factor of the experiment under study. The different treatments are the levels of the factor and the ANOVA helps to determine whether the different treatments have an overall effect on the experiment.

In general, an ANOVA tests the hypothesis:

$$\text{Let } H_0 : \mu_1 = \mu_2 = \dots = \mu_I$$

$$H_A : H_0 \text{ is false (at least two } \mu_i \text{ are different)}$$

where μ_i is either the true average response when treatment i is applied (for testing multiple treatments) or the true mean of population i (for testing multiple populations).

The analysis of variance tests the null hypothesis H_0 by comparing two different estimates of variance computed from the same data: the *mean square for error* (MSE) and the *mean square between treatments* (MSB). The MSE represents the variances within the treatment samples, while the MSB describes the variance of the means of the treatment samples. The test statistic for single-factor ANOVA is based on the F probability distribution and is computed $F = MSB/MSE$. The null hypothesis is accepted for $F \leq F_{critical}$, where $F_{critical} = F_{\alpha, df_1, df_2}$ is the critical value for an F density curve corresponding to the given number of degrees of freedom df_1 and df_2 and the desired significance level α ; otherwise, the null hypothesis is rejected in favour of the alternative hypothesis H_a . The degrees of freedom of an ANOVA is a function of both the number of treatments and the number of observations per treatment.

ANOVA						
Source of variation	<i>SS</i>	<i>df</i>	<i>MS</i>	<i>F</i>	<i>P-value</i>	<i>F_{critical}</i>
Between groups	5.970E-09	7	8.528E-10	0.31266	0.94783	2.7171
Within groups	6.328E-07	232	2.728E-09			
Total	6.388E-07	239				

Table 7.6: Summary table of Analysis of Variance (Example)

Often, an ANOVA table includes a *P-value*, which supports the level of significance at which a hypothesis is accepted or rejected. The level of significance represents the probability of rejecting the null hypothesis when the null hypothesis is true. Typically, one chooses the level of significance α before performing an ANOVA, setting it to a value such as 0.01. The *P-value* is the smallest α at which the null hypothesis would be rejected.

The results of a single-factor ANOVA are commonly represented in a summary table similar to Table 7.6. The table summarizes the sample data with the sum of squares (*SS*) and the mean square (*MS*) for within treatments and between treatments, the degrees of freedom (for the number of treatments per observation, and the number of observations) and the observed *F*-statistic. In addition, the table displays the critical *F*-value for which we reject the null hypothesis and the *P*-value for indicating the significance of accepting or rejecting the null hypothesis. Based on the summary table for this example, we accept the null hypothesis that the factor under study is not a significant factor in the variance of the means. We accept this hypothesis since the computed *P*-value 0.948 is greater than our cutoff probability 0.01; that is, the probability $P(F_{\text{observed}} \leq F_{\text{critical}}) = 0.948$.

7.5.2 Fitting a Curve

In some cases, we attempt to fit an algebraic expression or function to a given set of data; that is, given a set of discrete events, we determine a continuous model that best fits the data. This section describes, by use of example, the process used for fitting a model to the data. Figure 7.1 shows the original data, generated from the function $f(x) = 2x^2 + 10$.

Figure 7.1: Original Data for Example Curve Fitting

Even without knowledge of how the data is generated, the curve appears quadratic. To test the hypothesis that the data exhibits quadratic behaviour, the observed values $f(x)$ are plotted against $x' = x^2$ on a uniformly scaled x -axis and then fitted to a line using a least squares fit of data. If the resulting plot appears linear, the hypothesis is accepted. Figure 7.2 shows the original (dependent) data values plotted against the squared (independent) values and the corresponding least-squares fit line computed by gnuplot Version 3.7,⁶ which returns estimated parameters for the slope and intercept of the fitted line. The line is a perfect fit to the data in this example and the computed values for the slope and intercept are returned from gnuplot as 2 and 10, respectively. This exact fit is expected since the data are generated from a smooth, continuous function.

A good model for our data is described by the expression $f(x) = 2x' + 10$. The model is linear, but the hypothesis involves a quadratic expression. Recall, however, that the model is fitted for $x' = x^2$ and, therefore, simple substitution yields the corresponding quadratic model, $f(x) = x^2 + 10$. This continuous function models the original discrete data.

This method of fit is well-suited for testing the asymptotic behaviour of a set of

⁶ **gnuplot** is a command-driven interactive function plotting program. It can be used to plot functions and data points in both two- and three-dimensional plots in many different formats, and accommodates many needs for graphic data representation. **gnuplot** is copyrighted, but freely distributable [FAQ].

Figure 7.2: Least-Squares Fit of Data for Example Curve Fitting

Figure 7.3: (Continuous) Model of (Discrete) Data for Example Curve Fitting

data. For example, if we have data sampled from a set that is known to be $O(x^2)$, we can fit a line for the plot $f(x)$ versus $x' = x^2$ for the larger values of x' , as in Figure 7.4.

The first chart shows the least-squares linear fit for $x' > 40000$ (or $x > 200$), which does not model the behaviour of the curve for lower values of x' but seems like a good fit for larger values. The goodness of the fit is measured by R^2 , the coefficient of determination. A larger R^2 implies a better fit. The least-squares fit performed on the upper range of data results in a goodness of fit parameter $R^2 = 0.99992$, while a fit performed on all the data results in $R^2 = 0.99612$. Recall that for an asymptotic

Figure 7.4: Fitting a Curve for an Asymptotic Expression

expression in terms of x the observed behaviour will be more evident at large values of x .

The method of fitting a curve to data explained in this section is useful for modeling an expression with just one term plus a constant. It is a simple method that can be performed with pencil-and-paper calculations and by *eyeballing* the fit of the straight line. Fitting a straight line immediately implies that the observed behaviour matches the tested hypothesis (expression). Methods of fitting non-linear curves exist, but are not pursued here.

7.5.3 Accuracy of the Approximated Weight

An approximation isn't good if it doesn't compute a value close to the "true" value. The accuracy tests attempt to characterize the effect that the number of cones k has on the error of the approximated weight. The error is computed as the ratio of the approximated distance versus the Euclidean distance and is denoted ρ . Theoretically, this value should not be less than 1 and not more than $1/\cos \frac{\theta}{2}$ or, equivalently in terms of k , $1/\cos \frac{\pi}{k}$ (see Theorem 4.4.1).

Each test is performed on an nk -pair; that is, data is collected while keeping the size n of the point set and the number of cones k in the approximation scheme constant. We perform two series of tests: one where n is held constant and k is varied, and another where k is held constant and n is varied.

We show the analysis for one nk -pair and summarize the results for the other

combinations in a table following the analysis.

Example Analysis

The following example demonstrates the analysis for the accuracy of the distance approximation scheme when $k = 9$ and $n = 512$ on 60000 observations (*i.e.*, 2000 queries each from 30 sets of points, 10 each for the 3 different point distributions).

It is usually preferred that performance doesn't vary much from the mean, so the first statistics we compute are the sample mean and variance of the error ratios. Computing a confidence interval on this sample mean gives us probabilistic bounds that the true mean lies in an interval (c_1, c_2) ; however, a confidence interval can only be computed from model parameters if the parametric distribution of the data is already known.⁷

We test the hypothesis of whether our errors are normally distributed. The frequency of the occurrence of the ratios are plotted and fitted to a normal curve by gnu-plot (see Footnote 6) which returns estimated parameters for the mean and standard deviation. The plotted error appears to be approximately normal (see Figure 7.5).

Figure 7.5: Sample Approximation Error for 9 Cones, Range Tree

⁷There do exist non-parametric estimates of confidence intervals that overcome this limitation, but they are not pursued here.

To check the hypothesis that our numerical sample comes from a normal probability distribution, we construct a normal probability plot (Figure 7.6). Here, each sample data is plotted against its expected value under normality. If the points in the plot fall on the line $y = x$, it suggests that the data are normally distributed; otherwise, the points will depart from a linear pattern.

Figure 7.6: Normal Probability Plot ($k = 9$, $n = 512$, Range Tree)

To compute the expected values of the error ratios under normality, we use the following approximation from statistical theory of the expected value of the j^{th} smallest observation in a random sample of n observations [NKNW96, p.107]:

$$\bar{x} + s \left[z \left(\frac{j - 0.375}{n + 0.25} \right) \right]$$

where $\bar{x}$ and s are the sample mean and standard deviation, respectively, j is the rank of observation, n is the number of observations, and $z(X)$ is the $100X^{th}$ percentile of the standard normal distribution.

We fit a straight line to the resulting plot and compute the coefficient of determination R^2 , which is a measure of the goodness of the fit of the line to the data. The higher the value of R^2 , the better the fit. Although we get a fairly high value for R^2 ,

we see that our fit appears symmetrical with heavy tails, suggesting that the normal distribution may not be the best fit.

Figure 7.7: Normal Probability Plot ($k = 9$, $n = 32768$, Range Tree)

Table 7.7 summarizes the results for other nk -pairs. For each pair, the observed mean percentage error and standard deviation are shown, followed by the corresponding estimated parameters for the best fitting normal curve to the data and the R^2 value for the goodness of fit of the normal approximation. From the summary table, we see that as the number of points in the set S of points being queried increases, the value for R^2 decreases, suggesting that our distribution departs from normality. Figure 7.7 demonstrates the poor fit for a point set S of size 32768.

While the distribution appears to depart from normality, the mean error does not appear to be affected by the size of the point set on which the queries are taking place; that is, the accuracy of the approximation does not appear to be affected by the number of points in the point set being queried. An ANOVA is performed on 8 different point-set sizes and 30 responses for the average error from 2000 queries with the hypothesis:

$$\text{Let } H_0 : \mu_1 = \mu_2 = \dots = \mu_I$$

$$H_A : H_0 \text{ is false (at least two } \mu_i \text{ are different)}$$

where μ_i is the true ratio of error when point set i is used. After completing the ANOVA with significance level $\alpha = 0.01$ (see summary in Table 7.6), we accept the null hypothesis that the size of the point set is not a significant factor in the accuracy

k	n	Approximation Error					Normal LSF R^2
		worst-case error	Observed		Best-fit from model		
			mean	std. dev.	mean	std. dev.	
5	512	23.60%	15.63%	0.62%	15.68%	0.48%	0.96503
7	512	10.99%	7.30%	0.27%	7.31%	0.21%	0.98223
9	512	6.41%	4.27%	0.12%	4.27%	0.10%	0.99176
11	512	4.22%	2.81%	0.06%	2.81%	0.06%	0.99967
13	512	2.99%	1.99%	0.04%	1.99%	0.04%	0.99923
15	512	2.23%	1.48%	0.03%	1.49%	0.03%	0.99900
9	64	6.41%	4.27%	0.26%	4.28%	0.26%	0.99858
9	128	6.41%	4.26%	0.20%	4.27%	0.19%	0.99890
9	512	6.41%	4.27%	0.12%	4.27%	0.10%	0.99176
9	1024	6.41%	4.27%	0.09%	4.27%	0.08%	0.97746
9	2048	6.41%	4.27%	0.08%	4.27%	0.05%	0.94012
9	4096	6.41%	4.26%	0.07%	4.28%	0.04%	0.89659
9	8192	6.41%	4.27%	0.07%	4.28%	0.03%	0.83826
9	16384	6.41%	4.27%	0.07%	4.27%	0.02%	0.79445
9	32768	6.41%	4.27%	0.07%	4.27%	0.01%	0.75695

Table 7.7: Summary of Accuracy Analysis for different nk -pairs (Range Tree)

of the approximation. We accept this hypothesis since the computed P -value 0.948 is greater than our cutoff probability 0.01.

Since the range tree and persistent tree both employ exactly the same transformations from Cartesian coordinates to k -oriented coordinates, we expect to find identical results for approximation error. We find, however, that there are a number of *outliers* in the frequency distribution for the persistent tree (see Figure 7.8). These outliers have an approximation constant less than 1, which is unexpected, and lie several standard deviations from the mean. Except for the small fraction of queries that result in the outliers, all range tree and persistent tree queries of the same query point q on the same set of points S produce identical results. Less than 3% of the overall distance computations differ between the two data structures.

It is difficult to say what causes the inconsistent results between the range tree and the persistent search tree queries for the differing fraction of the results without further investigation. We speculate, however, that the differences might be caused by the handling of duplicate data in the data structures. Table 7.8 presents a summary of the sample mean, and minimum and maximum percentage error for both data structures.

Figure 7.8: Persistent Tree Approximation Error Outliers for $k = 5$

As previously described, the minimum value for the approximation constants differ, but more importantly the maximum values appear the same. The small fraction of outliers is emphasized by the similarity of the means between both data structures.

k	n	Observed Approximation Error					
		Range Tree			Persistent Tree		
		mean	min	max	mean	min	max
5	512	15.63%	11.31%	19.88%	15.50%	-70.5%	19.88%
7	512	7.30%	6.10%	8.47%	7.19%	-56.1%	8.47%
9	512	4.27%	3.66%	4.82%	4.13%	-46.5%	4.82%
11	512	2.81%	2.52%	3.07%	2.67%	-47.1%	3.07%
13	512	1.99%	1.78%	2.20%	1.87%	-46.3%	2.20%
15	512	1.48%	1.32%	1.64%	1.34%	-39.6%	1.64%
9	64	4.27%	3.18%	5.20%	2.60%	-59.3%	5.20%
9	128	4.26%	3.27%	5.04%	3.53%	-54.3%	5.05%
9	512	4.27%	3.66%	4.82%	4.13%	-46.5%	4.82%
9	1024	4.27%	3.79%	4.81%	4.21%	-46.5%	4.81%
9	2048	4.27%	3.83%	4.72%	4.25%	-39.1%	4.72%
9	4096	4.26%	3.84%	4.70%	4.26%	-37.8%	4.70%
9	8192	4.27%	3.84%	4.72%	4.26%	-22.8%	4.72%
9	16384	4.27%	3.85%	4.65%	4.26%	-31.2%	4.65%
9	32768	4.27%	3.85%	4.68%	4.27%	3.67%	4.68%

Table 7.8: Summary of Accuracy Analysis for different nk -pairs

7.5.4 Significance of the Effect of Point Distribution on Performance

The tests in this section are intended to answer whether the point distribution of the set of points S being queried is a significant factor contributing to both the accuracy (or error) of the approximation and the observed measured response time for a query. We use an analysis of variance (ANOVA) test (§7.5.1) for analysing the data.

The treatment for which the ANOVA is performed is the distribution of the points in the set S of n points with the following levels: *normally distributed*, *uniformly distributed in the unit circle*, and *uniformly distributed in the unit square*. The hypothesis is stated as follows, where r is the response time of a query under the specified distribution:

$$\text{Let } H_0 : r_{\text{Normal}} = r_{\text{unitCircle}} = r_{\text{unitSquare}}$$

$$H_A : H_0 \text{ is false (at least two } r_L \text{ are different)}$$

For an arbitrary nk -pair, we measure the response time for 500 queries

To determine whether the distribution of the points in the set S has a significant effect on the accuracy (error) of the approximation an ANOVA is performed. Table 7.9 summarizes the statistics for various nk -pairs.

Range Tree								
k	n	$F_{observed}$	P -value	$F_{critical}$	Effect			ω^2
5	8	3.7126	0.02463	3.0017	GN-UC	GN-US	UC-US	0.003604
5	1024	3.1619	0.04263	3.0017	0.15717	-0.09968	0.05747	0.002874
7	20480	3.9266	0.01991	3.0017	0.14805	0.01035	0.15841	0.003887
9	2048	15.5812	2.01E-07	3.0017	0.35058	-0.13910	0.21148	0.001907
11	4096	9.9906	4.90E-05	3.0017	0.24658	-0.24306	0.003520	0.01185
11	16384	19.5105	4.32E-09	3.0017	0.36057	-0.04045	0.32012	0.02409

Persistent Search Tree								
k	n	$F_{observed}$	P -value	$F_{critical}$	Effect			ω^2
5	8	12.8331	2.98E-06	3.0017	GN-UC	GN-US	UC-US	0.01553
5	1024	9.9831	4.93E-05	3.0017	-0.28260	0.14076	-0.14184	0.01184
7	20480	4.74765	0.00880	3.0017	-0.14801	-0.03579	-0.18380	0.004972
9	2048	1.67585	0.1875	3.0017	-0.10596	0.01255	-0.0934	0.00090
11	4096	1.26738	0.2819	3.0017	-0.06669	-0.03199	-0.09868	0.00036
11	16384	67.4012	9.37E-29	3.0017	-0.02965	-0.62059	-0.65024	0.08133

Table 7.9: Summary of ANOVAs on nk pairs for significance of point distribution

The values summarized in the table include the observed F -statistic, the critical F , and the P -value from the ANOVA table. In most cases, $F_{observed} > F_{critical}$ and $P\text{-value} < \alpha$, $\alpha = 0.01$, indicating the null hypothesis should be rejected in favor of the alternate hypothesis; however, the mean response times appear similar, so additional tests are run on the ANOVA results in order to explain the variance in the results. The following two tests are performed:

Cohen's d' The number of standard deviations separating two group means, this can be used as a measure of effect size. We compute Cohen's d' for each pair of distributions. The values are summarized in Table 7.9 under the headings GN-UC, GN-US, and UC-US, where GN indicates the (Gaussian) normal distribution of points, and UC and US are the uniform distributions inside the unit circle and unit square, respectively. While there are no generally accepted criteria for

determining whether a given d' is large enough to be important, Cohen made recommendation that a d' of .25 is a small effect, a d' of .50 is a medium sized effect, and a d' of .75 is a large effect [Lan01]. With most values falling below 0.25, we conclude that the effect of the point distribution on the response time is small.

omega squared An unbiased estimator of the percentage of variance explained by the effect variable (the point distribution). This is computed from the following formula:

$$\omega^2 = \frac{SSB - (n - 1)MSE}{SST + MSE}$$

Again, there are no generally accepted criteria for determining whether a given w^2 is large enough to be important; however, we consider values of less than 1%, and even up to 10% as unimportant in explaining the variance. For most of our trials, less than 1% of the variance is explained by the effect variable, implying that up to 99% of the variance might be explained by unconsidered factors.

The two tests support each other's conclusions.

The persistent tree data structure shows slightly more variability between groups for query response time than the range tree data structure. This may be an indirect result of the difficulties explained with the persistent tree in the previous section.

The approximation is found to overestimate the Euclidean sum by no more than 9% for $k \geq 8$, in theory. When looking at uniformly or normally distributed points, we find the overestimate to be roughly two-thirds that of the predicted maximal error. At this time, we feel it is important to emphasize that the point sets on which the tests are performed are fairly well-conditioned. The points are smoothly distributed according to some rule throughout a certain region. Future work might consider different distributions such as a star whose points coincide with the bisecting axes of the cones in the simplicial cone approximation structure, and points on the perimeter of the unit square and unit circle. We expect in those situations that the effect of the point distribution on the observed performance might be exaggerated.

7.5.5 Observed vs. Theoretical Creation Time

The goal of this section is to show that the creation times for both the persistent search tree and the range tree implementations of the weight computation exhibit $O(n \lg^2 n)$ behaviour. To check the plausibility of this claim the observed creation time for each data structure is plotted against the corresponding expression denoting the asymptotic query time (as a function of n). The analysis employs the following logic: if the distribution on which the plot is based follows the analytic result, the points in the plot will tend toward a straight line (§7.5.2).

Figure 7.9 shows the observed creation times for each data structure over different sizes of the point set for which the range query structure is built. The time taken to create the naïve structure appears negligible compared to the other data structures. In fact, the naïve query data structure is simply an interface that does not even depend on the size of the point set being queried and hence should be constant. The creation times for range tree structure and persistent tree structure appear to be linear in the size of the input. Thus, we perform a least squares fit of a linear function to each data set. We note that for smaller values of n , the creation times lie below the best-fit line and tend to lie above it as n grows. This suggests that there exists an unconsidered, slowly growing factor, say $\lg n$. We proceed to plot the observed creation times against the theoretical expression representing the expected behaviour (in Figure 7.10).

The linear least squares fit appears to model the data better in the theoretical plot, which implies that the observed creation time exhibits theoretical behaviour. This observation is supported by the fact that the computed SSR (sum of squares of the residuals) is smaller for both data structures when the data are plotted against the hypothesis for the theoretical creation time is plotted.

For the cost-benefit analysis (§7.5.7), it is useful to have algebraic expressions defining the observed creation times. A linear function is denoted $f(x) = mx + b$, where m is the slope of the line and b is the y -intercept. The data is modelled by a linear function using a least-squares-fit for $x \equiv n \lg n$. Direct substitution results in a simple theoretical model of the performance with the form of the expected asymptotic expression of the data structure's performance: $f(n) = m \cdot n \lg n + b$. The result also gives an approximation for the constant factor that precedes the asymptotic

Figure 7.9: Observed Creation Times ($k = 9$)

expression. We caution the reader that this constant is valid only for the machine on which the tests are run and showing that the algorithm exhibits theoretical behavior is a more conclusive statement on the performance. Additionally, it should be noted that additional terms that are dominated by the asymptotic largest term are neglected in this simple model. The models for the creation time of the persistent search tree data structure and the range tree data structure for $k = 9$ are found to be approximately $f(n) = 70400n \lg n - 146$ and $f(n) = 51800n \lg n - 878$, respectively, where $f(n)$ is the observed creation time in clock ticks per second and n is the size of the point set.

7.5.6 Observed vs. Theoretical Query Time

The goal of this section is to show that the query times for the persistent search tree, the range tree and the naïve implementations of the weight computation exhibit $O(\lg n)$, $O(\lg^2 n)$ and $O(n)$ behaviour, respectively. To check the plausibility of this claim the average query time for each data structure is plotted against the corresponding expression denoting the asymptotic query time (as a function of n). The analysis employs the same logic as the probability plots of the previous section. If the distribution on which the plot is based is correct, the points in the plot will fall close to a straight line.

Figure 7.10: Observed vs. Theoretical Creation Times ($k = 9$)

Figure 7.11 compares the behaviour of the distance approximation schemes with the naïve approach in terms of the observed average query times with the goal of showing that the approximation performs better than the naïve approach (after some threshold). The plot shows the average observed query times over different point set sizes for all three approaches to computing the weight of a point. The naïve approach is fitted to a linear curve, as expected. Additionally, as n gets larger, the observed average query time of the naïve approach appears to significantly depart from the observed average query times of the approximation approaches.

The query times for the range and persistent based data structures are shown in Figure 7.12. These values are plotted against the expressions describing their theoretical query times in Figure 7.13. We see that for larger values of n , the least-squares-fit line appears to describe the data and so, in both cases, the observed average query time exhibits the expected theoretical behaviour. The line depicted for the range tree is fit considering only those observations from point sets with more than 1024 points, while the persistent tree regression line considers point sets with more than 64 points.

Some reasons for which the model might not appear to be a good fit against the expected theoretical expression include:

Figure 7.11: Observed Average Query Times ($k = 9$)

1. The experimental upper bound for n , point set size, is restricted by the physical limitations imposed by the available computing resources, specifically, internal memory; and,
2. The expression defining the distribution on which the plot is based is a theoretical *asymptotic* expression; that is, we expect the average query response time to exhibit behaviour proportional to the limit of the expression as n approaches infinity. This might explain the divergence from the linear fit as n moves toward 0.

The previous section presented algebraic models for the observed creation times of the data structures. It is useful to have these models defined for the cost-benefit analysis. The same procedure is used here for defining a model on query times. The average observed query times for the persistent search tree data structure and the range tree data structure for $k = 9$ are approximately modelled as $f(n) = 6500 \lg n + 51567$ and $f(n) = 1560 \lg^2 n - 1916$, respectively, where $f(n)$ is the observed query time in clock ticks per second and n is the size of the point set. Similarly, we model the observed query time for the naïve computation from Figure 7.11 as $f(n) = 265n - 97546$.

Figure 7.12: Observed Average Query Times ($k = 9$) Figure 7.13: Observed Average Query Time vs. Theoretical ($k = 9$)

Table 7.10 summarizes observed creation and query times for three different sized-point sets (n) and using different numbers of cones (k) in the approximation structure. The creation times are represented in seconds; the query times are represented in terms of queries per second.⁸ The naïve approach does not incur a creation cost. The table demonstrates that on a set of 512 points, the naïve approach outperforms the approximation scheme in terms of query time for 5 through 18 cones. On a set of 4096 points, the scheme for approximating the Euclidean sum outperforms the naïve approach. The benefit of the lower query time is therefore established on point sets containing between roughly 500 and 4000 points. The next section presents an approach for including the amortization of the creation cost when judging the benefit of using the approximation scheme for the Euclidean sum.

Note that the observed number of queries per second in Table 7.10 is very small on 32768 points for $k \geq 13$ (range tree) and $k \geq 18$ (persistent tree). The cause is that the data structure became too large to store in primary memory⁸ and swapping occurred, thus greatly increasing the cost of querying.

⁸The observations were measured on a 600Mhz computer running Linux kernel version 2.2.17 with a Pentium III Processor and 256Mb of SDRAM. Tests were compiled using gcc version 2.95.3 with optimization settings -O3.

n	k	Creation (sec)		naïve	#Queries/sec	
		range	persistent		range	persistent
512	5	0.152	0.284	10 105	7 583	9 461
	6	0.180	0.337		6 334	8 028
	7	0.214	0.394		5 402	6 866
	8	0.245	0.438		4 838	6 281
	9	0.274	0.505		4 184	5 292
	10	0.307	0.560		3 808	4 850
	13	0.403	0.722		2 927	3 696
	16	0.495	0.870		2 392	3 066
	18	0.551	0.990		2 112	2 699
4096	5	1.994	3.102	904	4 922	8 191
	6	2.368	3.700		4 105	6 945
	7	2.787	4.335		3 464	5 954
	8	3.202	4.808		3 123	5 412
	9	3.583	5.570		2 696	4 642
	10	3.948	6.169		2 443	4 201
	13	5.153	8.001		1 868	3 234
	16	6.391	9.686		1 505	2 657
	18	7.140	10.998		1 326	2 341
32768	5	23.492	32.436	71	3 131	7 001
	6	28.139	38.731		2 597	5 927
	7	32.846	45.388		2 207	5 070
	8	37.861	50.476		1 946	4 591
	9	41.986	58.371		1 711	3 963
	10	46.671	64.704		1 533	3 583
	13	69.382	84.162		3.0	2 755
	16	124.978	102.054		0.5	2 170
	18	157.208	128.132		0.3	7.5

Table 7.10: Summary of observed creation and query times over different k and n

7.5.7 Cost-Benefit Analysis

In Sections 7.5.5 and 7.5.6, we discussed the observed query and creation times of the distance approximation scheme data structures. In this section, we give an example of the cost-benefit analysis of using the approximation schemes; that is, for an arbitrary nk -pair, we compute an approximate value of n for which the cost of the creation time is recovered by the faster query times of the distance approximation scheme to the naïve approach.

The total cost for computing the weight of a point can be measured in terms of the time it uses the CPU:

$$T_x = C_x + m \cdot Q_x$$

where T_x is the total cost of running the algorithm (on the data structure) x , C_x is the observed preprocessing or creation time for the data structures used by the algorithm, Q_x is the observed average query time for a single query and m is the number of queries performed on the same data structure.

If an algorithm x has a greater creation time but smaller query time than an algorithm y , we can describe a function that computes the value m for which the (creation) cost of using scheme x is recovered through faster queries. This is done by setting $T_x = T_y$ and solving for m . For example, the persistent search structure has faster observed query times than the naïve approach once the point set size passes some threshold. Setting $T_{naive} = T_{persistent}$ results in the following expression for m :

$$\begin{aligned} T_{naive} &= T_{persistent} \\ C_{naive} + m(265n - 97546) &= 70400n \lg n - 146 + m(6500 \lg n + 51567) \\ m &\approx \frac{70400n \lg n}{265n - 6500 \lg n} \end{aligned}$$

where m is the approximate number of queries to perform on the persistent data structure in order to regain the creation cost and see the benefit of the faster query time of the persistent structure over the naïve structure. The constant terms are dropped since they become less significant as n grows. A similar analysis reveals the following expression for m for the range tree data structure:

$$\begin{aligned} T_{naive} &= T_{range} \\ m &\approx \frac{51800n \lg n}{265n - 1560 \lg^2 n} \end{aligned}$$

The functions are plotted, along with the computed observations on which the creation and query time models are based, in Figure 7.14.

Figure 7.14: Plots of the cost-benefit analysis ($k = 9$)

The plots indicate that the number of queries needed to perform in order to recover the observed creation cost is around 4000 for the persistent tree and 3000 for the range tree for a large number of the point sets tested. When the function or data have negative values, it is not possible to recover the cost of creation through queries since the cost of creation exceeds the observed query time of the naïve approach. The crossover point, from negative values to positive, indicates the point set size at which a benefit of using the data structures with smaller query times is first observed. In practice, these crossover points occur between 500 and 1000 points for the persistent tree and between 1000 and 2000 points for the range tree.

The computed recovery function is a slowing increasing function, as shown in Figure 7.15. Assuming unlimited resources, the plot shows the number of queries needed to perform on a point set to recover the creation cost of using the tested data structures.

Similar analyses are performed for $k = 5$ and $k = 7$. The results are summarized in Table 7.11.

Figure 7.15: Projection of the cost-benefit analysis ($k = 9$)

k	Data Structure	$T_{DS} = C_{DS} + m \cdot Q_{DS}$
5	Persistent Tree	$T_{persistent} \approx 39700n \lg n + m \cdot 3660 \lg n$
5	Range Tree	$T_{range} \approx 28900n \lg n + m \cdot 856 \lg^2 n$
7	Persistent Tree	$T_{persistent} \approx 54950n \lg n + m \cdot 5080 \lg n$
7	Range Tree	$T_{range} \approx 40030n \lg n + m \cdot 1210 \lg^2 n$
9	Persistent Tree	$T_{persistent} \approx 70400n \lg n + m \cdot 6500 \lg n$
9	Range Tree	$T_{range} \approx 51800n \lg n + m \cdot 1560 \lg^2 n$

k	Data Structure / Approach	Observed Crossover Range	Number of Queries to Recover Cost
5	Persistent Tree	(250, 500)	≈ 2500 for $n \in (6000, 45000)$
5	Range Tree	(500, 1000)	≈ 1800 for $n \in (6000, 45000)$
7	Persistent Tree	(500, 1000)	≈ 3000 for $n \in (6000, 45000)$
7	Range Tree	(1000, 2000)	≈ 2400 for $n \in (6000, 45000)$
9	Persistent Tree	(500, 1000)	≈ 4000 for $n \in (6000, 45000)$
9	Range Tree	(1000, 2000)	≈ 3000 for $n \in (6000, 45000)$

Table 7.11: Summary of cost-benefit results for $k = 5$ and $k = 7$

7.6 Summary Remark

The goal of the experiments and statistical tests outlined in this chapter was to answer the questions posed at the beginning of the chapter. The conclusions drawn from the results attempt to characterize, in general terms, the performance of the implemented Euclidean sum approximation algorithm using two different data structures. To avoid repetition, we refer the reader to the next chapter, Conclusions, for a summary of the results presented in response to the original questions.

Chapter 8

Summary and Conclusions

In this thesis, we have studied the approximation scheme for evaluating the sum of Euclidean distances (or the weight of a point) proposed in a paper by Bose *et al* [BMM01]. The motivation for Bose *et al*'s research is to efficiently solve the Weber (single facility location) problem. The purpose of this thesis has been to characterize the performance, in practice, of the theoretical behaviour of the approximation scheme.

We conclude the thesis with a summary and focus largely on stating the conclusions drawn from the previous chapter's analysis of the approximation scheme.

8.1 Summary of the Implementation

The data structures used for the Euclidean sum approximation scheme include the range tree and the persistent search tree. The first implementation of the Euclidean sum approximation scheme attempted to augment LEDA's range tree implementation with additional information in the nodes. The results were disastrous and the implementation was abandoned. LEDA's separate internal and external node approach to implementing binary tree data types does not lend itself easily to reuse. The LEDA range tree possesses qualities indicating that it is meant to be extended; in fact, the LEDA project sees an extension of the range tree through inheritance with its own `d2_dictionary` class. The `d2_dictionary` class does not, however, attempt to modify the nodes of the tree. Additionally, the `range_tree` class is already the third class deep in LEDA's inheritance hierarchy, and complex inheritance structures

create many opportunities for faults [Bin00]. LEDA's geometric primitives are useful, however, for working with basic geometric objects.

Both the persistent tree and range tree used for collecting data for the analysis of the performance of the Euclidean sum approximation scheme were implemented from scratch. Each data structure builds from an existing extensible implementation of a red-black tree, not through inheritance, but instead by adopting the behaviour of the basic data structure and introducing data-structure-specific functionality (*e.g.*, *memory* for the different versions of the persistent search tree). The implementation from scratch is not without setbacks; however, the development process was more straightforward than the attempt to tailor an existing software (LEDA) artifact.

8.2 Summary of the Experimental Results

The statistical analysis of the data structures focused on characterizing the behaviour of the implemented Euclidean sum approximation scheme. Emphasis was placed on quantifying the accuracy of the approximated sum and observing whether the performance in theory and in practice exhibit the same behaviour. We summarize the results of Chapter 7 here.

The accuracy of the approximated weight of a point is affected by the value chosen for k in the k -oriented approximation scheme. We observe through experimentation that, as k increases, the approximated weight evaluates closer to the *true* value, as computed by a naïve approach. In practice, the approximation error is smaller than the computed worst case error (see Tables 7.7 and 4.1, respectively). In practice, the approximated error never exceeds the theoretical upper bound, which is $1/\cos \frac{\pi}{k}$; however, for the persistent tree, the approximation constant falls below 1, which is unexpected. We speculate that this may be the result of an oversight in the implementation of the insertion or query algorithm for the persistent search tree.

For each constant k tested, neither the size of the point set nor the distribution of the points seems to affect the (mean) approximation error; however, future work might consider different, less smoothly distributed sets of points that might exaggerate the effect of the point distribution on the observed approximation error. When looking at uniformly or normally distributed points, we find the overestimate to be roughly

two-thirds that of the predicted maximal error.

The observed creation and query times convincingly exhibit theoretical behaviour as the size of the point set gets large. The difficulty in making a conclusive statement on how the data structures and query algorithms perform in practice (compared with how they perform in theory) is that the computer has only limited resources; specifically, the amount of available memory limits the size of the tests that can be run.

Based on the observed data, we speculate on the number of queries that need to be executed on the same data structure in order to regain the cost of preprocessing (*i.e.*, the creation of the data structure for storing the approximation information). Experimentally, when the point set is *large enough*, the number of queries to regain the creation cost is in the thousands, as outlined in Table 7.11. The quantities vary slightly for different values of k .

The persistent search tree appears to exhibit faster query times than the range tree data structure; however, the observed creation time of the persistent search tree is consistently greater than the creation time of the range tree. Consequently, the number of queries to regain the cost of using the persistent tree is greater than the number of queries needed to perform in order to regain the cost of using the range tree. To be useful in practice, many evaluations of the Euclidean sum approximation need to be performed on the same point set.

In order to use the approximation scheme in practice, a number of factors must be considered:

1. how much approximation error will be tolerated?
2. what are the limitations on the available memory for storing the approximation data structure?
3. how many queries will be performed on the same point set?

Answers to these questions might lead the user of the approximation scheme to make a decision on the number of cones to use in the application of the data structure. Summary tables (Tables 7.7, 7.8, 7.10, and 7.11) in the previous chapter might help the user make these decisions.

8.2.1 Future Work

Future work might consider:

1. What are suitable applications for the approximation? One such application might be the Weber facility location problem, which aims to find the point q' that minimizes the sum $w(q')$. In finding a solution, $w(q)$ may have to be computed many times.
2. How does the approximation scheme generalize to and perform in higher dimensions? What are suitable applications for the approximation scheme on higher-dimensioned point data?

Bibliography

- [Aga96] Pankaj K. Agarwal. Range searching. Technical Report CS-1996-05, Duke University Department of Computer Science, Durham, North Carolina, September 1996.
- [AU92] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science*, chapter 1. Computer Science Press, New York, 1992.
- [Baj88] Chanderit Bajaj. The algebraic degree of geometric optimization problems. *Discrete and Computational Geometry*, 3:177–191, 1988.
- [BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39(10):104–116, October 1996.
- [BDGR95] Shirley Browne, Jack Dongarra, Eric Grosse, and Tom Rowan. The netlib mathematical software repository. *D-Lib Magazine*, September 1995. <http://www.dlib.org/dlib/september95/09contents.html>.
- [Bin00] Robert V. Binder. *Testing Object-Oriented Systems. Models, Patterns, and Tools*. Addison-Wesley, 2000.
- [BMM01] P. Bose, A. Maheshwari, and P. Morin. Fast approximation of sums of distances, clustering and the fermat-weber problem. *Computational Geometry: Theory and Applications*, 2001. To appear.
- [CG86] B. Chazelle and L. Guibas. Fractional cascading: I a data structuring technique. *Algorithmica*, 1:133–162, 1986.

- [Cha85] Bernard Chazelle. How to search in history. *Information and Control*, 64(1-3):77–99, January/February/March 1985.
- [CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.
- [CT90] R. Chandrasekaran and Arie Tamir. Algebraic optimization: The fermat-weber location problem. *Mathematical Programming*, 46(2):219–224, 1990.
- [dBvKOS97] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry Algorithms and Applications*, chapter 5, pages 103–109. Springer-Verlag, 1997.
- [DSST89] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.
- [FAQ] Gnuplot faq. <http://www.gnuplot.org/gnuplot/gnuplot-faq.html>.
- [FW74] Richard L. Francis and John A. White. *Facility Layout and Location*, chapter 4. Prentice-Hall, 1974.
- [GKP94] R. Graham, D. Knuth, and O. Patashnik. *Concrete mathematics. A foundation for computer science*. Addison-Wesley, second edition, 1994.
- [Jai91] Raj Jain. *The Art of Computer Systems and Performance Analysis*. John Wiley & Sons, 1991.
- [Knu98] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Seminumerical Algorithms. Addison-Wesley, 3rd edition, 1998.
- [Kuh73] Harold W. Kuhn. A note on fermat’s problem. *Mathematical Programming*, 4(1):98–107, 1973.
- [Lan01] David M. Lane. Hyperstat online textbook. <http://davidmlane.com/hyperstat/index.html>, ©1993-2001.

- [LK82] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. McGraw-Hill Book Company, 1982.
- [Mey82] Bertrand Meyer. Principles of package design. *Communications of the ACM*, 25(7):419–428, July 1982.
- [MN99] Kurt Mehlhorn and Stefan Näher. *LEDA A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
- [Mon97] Douglas C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, fourth edition, 1997.
- [Mor] Pat Morin. An extensible implementation of a balanced binary tree. <http://cgm.cs.mcgill.ca/~morin/>.
- [NKNW96] John Neter, Michael H. Kutner, Christopher J. Nachtsheim, and William Wasserman. *Applied Linear Statistical Models*. WCB McGraw-Hill, 4 edition, 1996.
- [O'R94] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1994.
- [PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry An Introduction*. Springer-Verlag, 1985.
- [PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C, The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.
- [Smi97] Michiel Smid. Vorlesungsskript geometrische netzwerkprobleme. Lecture notes, 1997.
- [ST86] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.
- [Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

- [Wei92] Mark Allen Weiss. *Data Structures and Algorithm Analysis*. The Benjamin/Cummings Publishing Company, Inc., 1992.
- [Wei00] Mark Allen Weiss. *Data Structures and Problem Solving Using C++*. Addison Wesley Longman, Inc., 2000.