Clustering and Reconstructing Large data sets

A Dissertation Presented

by

Piyush Kumar

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer Science
Stony Brook University
June 2004

Schomberg, Stony Brook, NY 11790.
E-Mail: piyush at acm dot org

Printed on the Samsung CLP-500, Joe's Press, NY.

The paper used in this publication meets the minimum requirements of the Stony Brook Graduate School.

First edition: 26 June 2004

Clustering and Reconstructing Large data sets

Approved by:

Michael Bender, Committee Chair

Joseph S.B. Mitchell, Advisor

Esther M. Arkin

Alper Yildirim

Date Approved

Clustering and Reconstructing Large data sets

ABSTRACT

by

Piyush Kumar

Doctor of Philosophy

in

Computer Science

Stony Brook University

June, 2004

Abstract

This thesis deals with problems at the intersection of computational geometry, optimization, graphics, and machine learning. Geometric clustering is one such problem we explore. We develop fast approximation algorithms for clustering problems like the k-center problem and minimum enclosing ellipsoid problem based on the idea of core sets. We also explore an application of the 1-center problem to recognition of people based on their hand outlines.

Another problem we consider in this thesis is how to reconstruct curves and surfaces from given sample points. We show implementations of algorithms that can handle noise for reconstructing curves in two dimensions. Based on Delaunay triangulations, we develop a surface reconstructor for a given set of sample points in three dimensions.

When dealing with massive data sets, it is important to consider the effect of memory hierarchies on algorithms. We explore this problem in our research on cache oblivious algorithms. We develop a practical cache oblivious algorithm to compute Delaunay triangulations of large point sets. We end the thesis with another optimization problem of approximately finding large empty convex bodies inside closed objects under various assumptions.

To my Parents and Teachers
who taught me how to learn

Contents

List of Tables

List of Figures

Acknowledgements

"Forget injuries, never forget kindnesses."

CONFUCIUS

This work could not have been completed without the help and guidance of my advisor, Joe Mitchell. I thank him for allowing me the freedom that I always wished I had. He managed to strike the perfect balance between providing direction and encouraging independence. I don't think I would ever have managed to get a PhD with anyone else.

My visits to MPI-Saarbruecken during the course of my PhD were not only a welcome distraction from my dissertation research but also a privilege to learn more than I could possibly assimilate in the time frame I learned that material. I am extremely grateful to my mentor, advisor and inspirator at MPI, Edgar Ramos, it was a distinct privilege to work with him. Special thanks go to Stefan Funke and Peter Sanders for their lucid explanations of things I would have never understood otherwise. I would also like to thank the company of Ernst Althaus, Hannah Bast, Roman Dementiev, Fritz Eisenbrand, Juha Kärkkäinen, Irit Katriel, Lutz Kettner, Piotr Krysta, Kurt Mehlhorn, Uli Meyer, Tobias Polzin, Rahul Ray, Joachim Reichel, Elmar Schömer, Naveen Sivadasan, Sven Thiel from whom I learnt a lot during my various stays at the MPI.

During my stay at Stony Brook, the company of Michael Bender infused in

me the liking for cache-efficient algorithms. Alper Yıldırım corrupted me with semidefinite programming and non-linear optimization, Esther Arkin with linear programming in general, George Hart with his projections of higher dimensional polytopes, Hong Qin with physics based modeling, Martin Held with his coding efficiency, Saurabh Sethia with his enthusiasm and Manuel Oliveira with his passion for graphics.

I acknowledge two others from a time and place that now seem remote, but which left a big impact on my life. I wish to thank Tamal Dey, who, when I was 19, introduced me to computational geometry. I also wish to thank Subir Ghosh and Sudebkumar Prasant Pal for spending with me one of the best summers I have ever had thinking non stop about geometry.

During my various tours when I was a PhD student I had the privilege to meet and discuss research with the following researchers, some of them helped me more than they may realize: Boris Aronov, Marshall Bern, Siu-Wing Cheng, Ken Clarkson, Erik Demaine, Alon Efrat, Jeff Erickson, Joachim Giesen, Sariel Har-Peled, Shankar Krishnan, Joseph O'Rourke, Sheung-Hung Poon, Amit Roy, Jonathan Richard Shewchuk, Hisao Tamaki, Kasturi R. Varadarajan, Suresh Venkatsubramanyam, Carola Wenk and the names that I forget.

Thanks to all my co-authors who made this thesis possible. Also many thanks to my friends at Stony Brook who took the torture of my puzzles with remarkable endurance and who made Stony Brook a pleasant place to stay: Uday, Vivek, Sachin, Shiva, Liz. To Petr, for getting me interested in linux hacks and kernels, and to Olaf for keeping my interest in surface reconstruction alive.

And to Eva, for getting me interested in viruses (not the ones that I used to write), for proofreading, and for keeping my graduate life more interesting than it could be. Last but not the least, to the reader, for reading my acknowledgements.

CHAPTER I

Introduction

"In theory there is no difference between theory and practice,
But in practice there is."

YOGI BERRA

One of the most exciting consequences of the rapid growth in both theoretical and practical techniques in various areas of computer science is the ability to cluster huge amounts of data. In this thesis we are mainly interested in clusterings that are geometric. One of the most interesting applications of geometric clustering is the ability to create 3D models for a virtual world from pictures. This thesis deals with *Clustering and Reconstructing* point sets in spaces and their applications. We also look at a new model that helps in the analysis of algorithms when the data is huge.

Traditionally, theoretical computer science relies on making assumptions that do not generally hold in the real world. For instance, the performance of algorithms on real data vs. worst case analysis, the assumption that all memory operations are unit cost, noise in the input, degeneracies, importance of exact solution to optimization problems are issues that need attention. Addressing some of these issues is the focus of this thesis. The work on

core sets illustrates why column generation methods perform well in optimization, both theoretically and experimentally. We give simple surface and curve-reconstruction algorithms that are *provably fast* and could handle noise in the data. The work on cache oblivious algorithms shows why certain ways to analyze algorithms compared to the RAM model of analysis is better for practical purposes. The thesis also has a chapter on the design of a classifier for biometric and other applications. We use this classifier for identifying people using the outline of their hand. The classifier is based on our research in clustering.

We now give a short introduction to the different chapters in the thesis.

1.1 Core Sets/Column Generation

Clustering is an important task in computer science, scientific research and massive-data-set applications. Finding clusters in data has many applications including pattern recognition and machine learning. Minimum enclosing balls (MEBs) and minimum volume ellipsoids (MVEs) of point sets are two measures of clustering that have been widely used in both theory and practice. Both of these problems are easy to solve in lower dimensions, but as the dimension of the data grows, these problems become hard to solve. High-dimensional data also poses other interesting questions for clustering data.

We study the MEB problem for sets of points or balls in higher dimensions. Using techniques of second-order cone programming and *core-sets*, we develop $(1+\varepsilon)$-approximation algorithms that perform well in practice, especially for very high dimensions, in addition to having provable guarantees. Our algorithm, which is simple to implement, results in fast computation of nearly optimal solutions for point sets in much higher dimension than previously computable using exact techniques. Moreover, we show that one can cleverly pick $O(1/\varepsilon)$ points from the input, compute an approximate solution for

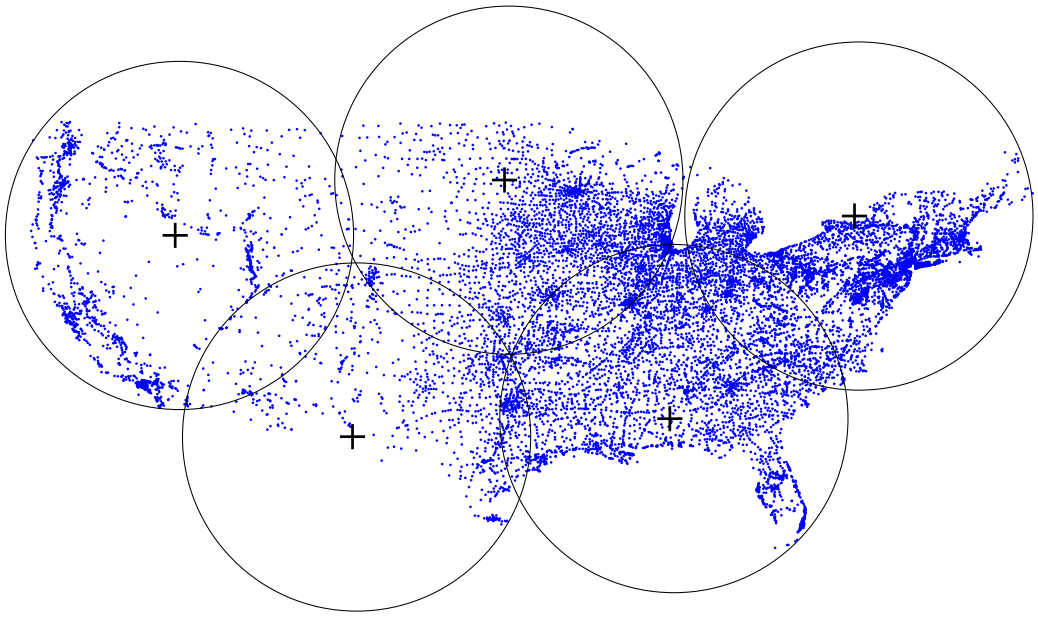


Figure 1: Example of approximate 5-center clustering of US Cities.

this set and get a guaranteed $(1+\varepsilon)$-approximation for the entire input. This is the set of points known as a *core-set*.

We improve the first order algorithm of Khachiyan for computing MVEs of a given set of points and extended it. We also develop methods to solve both these problems faster than was known before and also implement our algorithms. Our research showed that the minimum volume ellipsoid problem can also be solved approximately by choosing a few points of the input cleverly and solving a sequence of subproblems instead of solving the whole problem. We generalize our methods for the MEB/MVE problems, to solve the minimum enclosing ball of ellipsoids and minimum volume ellipsoids of ellipsoids.

The result on minimum enclosing balls also improves the previous approximation results for k-center clustering in the geometric setting (for an example, see figure 1). Subsequently, we also apply the MEB algorithm to cluster features for biometric applications.

1.2 Curve and Surface Reconstruction

The problem of curve and surface reconstruction has applications in computer graphics, computer vision, image processing, speech recognition and reverse engineering. Most provable reconstruction algorithms prior to my research assumed that there was no noise in the data set for reconstruction. This assumption is a major problem when one tries to apply the provable reconstruction algorithms to 'real' data.

Along with my collaborators, we developed a very simple algorithm to reconstruct curves from sample points when there was no noise. We then extended this simple curve reconstruction algorithm to handle noise in the data. I also report on my implementations of curve and surface reconstructors in this thesis (see figure 2 and 3).

I also maintain a web portal[1] on curve and surface reconstruction which is ranked highly on search engines and is frequently visited by people in academia and industry.

1.3 Cache Oblivious Algorithms

The cache oblivious model is a simple and elegant model to design algorithms that perform well in hierarchical memory models ubiquitous on current systems. Analyzing and designing algorithms and data structures in this model involves not only an asymptotic analysis of the number of steps executed in terms of the input size, but also the movement of data optimally among the different levels of the memory hierarchy.

[1] `http://www.compgeom.com/www.sites.html`

Figure 2: Example of Curve Reconstruction with Noise.

Delaunay triangulations are ubiquitous in computational geometry and have applications in computer graphics, vision, geographic information systems, robotics and other areas. We designed a cache oblivious Delaunay triangulation algorithm that can triangulate huge sets of points. We also implemented and experimented with different cache oblivious algorithms, such as sorting and matrix manipulation routines.

1.4 Biometrics

Biometric recognition systems find applications in security systems of varying requirements. In this work, we discuss the issues and challenges in the design of a hand outline

Figure 3: Example of a reconstructed surface.

based recognition system (see figure 4 for an example of a hand outline). Our system is easier to use, cheaper to build and more accurate than previous systems. In this project we do extensive tests on more than 700 images collected from 70 people. Classification, verification and identification of the input images is done using a geometric classifier. We achieve more than 99% success rate in verification and identification. We design and use a novel minimum enclosing ball classifier which performs well for hand recognition and could be of interest for other applications.

1.5 Finding Large Empty Convex Bodies

The last chapter of the thesis involves finding convex inner approximations of two and three dimensional shapes. Convex inner approximations function as efficient occluders as they

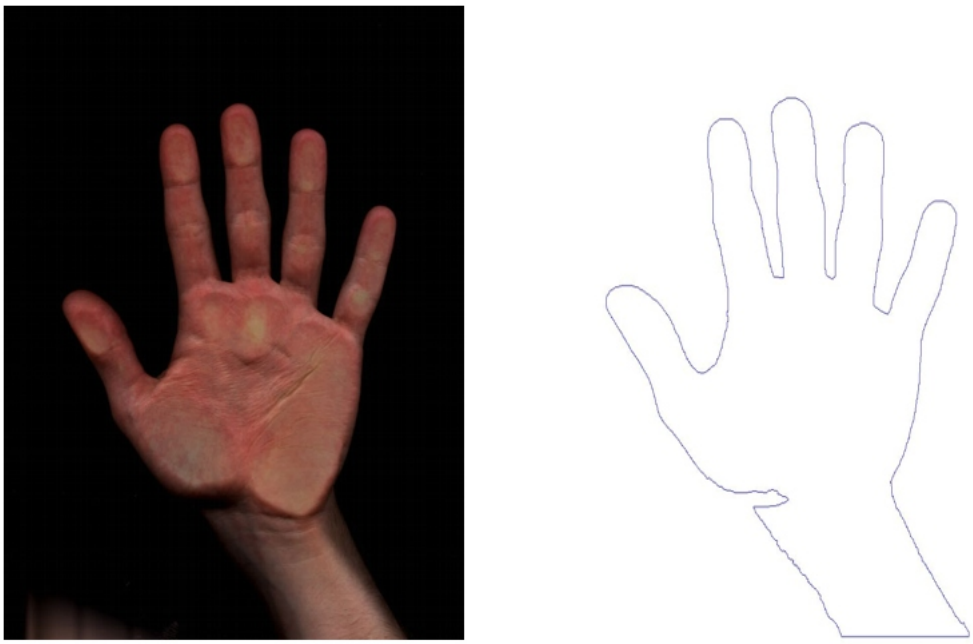


Figure 4: Left: A scanned image of a hand. Right: Hand outline extracted from the image scanned.

reduce the complexity of the shape being approximated by a considerable factor. These occluders can be used to speed up rendering by removing large amounts of data that will not be visible in the final display.

The general problem of finding a large convex body inside a shape is a non-convex optimization problem. Hence in this chapter we consider variations of this problem in two and three dimensions which are susceptible to approximation algorithms.

CHAPTER II

Approximate 1-Center

"All exact science is dominated by the idea of approximation."

BERTRAND RUSSELL

IN this chapter, we study the minimum enclosing ball (MEB) problem for sets of points or balls in high dimensions. This problem is also known as the 1-center problem. Using techniques of second-order cone programming and "core-sets", we will develop a $(1+\varepsilon)$-approximation algorithm that performs well in practice, especially for very high dimensions, in addition to having provable guarantees. We will prove the existence of core-sets of size $O(1/\varepsilon)$, improving the previous bound of $O(1/\varepsilon^2)$, and study empirically how the core-set size grows with dimension. We will see that our algorithm, which is simple to implement, results in fast computation of nearly optimal solutions for point sets in much higher dimension than previously computable using exact techniques.

In the MEB problem we are asked to compute a ball of minimum radius enclosing a given set of objects (points, balls, etc) in $\mathbb{R}^d$. The MEB problem arises in a number of important applications, often requiring that it be solved in relatively high dimensions. Applications of MEB computation include gap tolerant classifiers [48] in Machine Learning, tuning Support Vector Machine parameters [52], Support Vector Clustering [30, 194], preprocessing for fast farthest neighbor query approximation [104], k-center clustering [47],

testing of clustering [10], computing dense regions of an unknown distribution [29], solving the approximate 1-cylinder problem [47], computation of spatial hierarchies (e.g., sphere trees [114]), and other applications [88].

In this chapter, we give improved time bounds for approximation algorithms for the MEB problem in which the given set of objects consists of points or balls in high dimensions. We prove a time bound of $O(\frac{nd}{\varepsilon} + \frac{1}{\varepsilon^{4.5}} \log \frac{1}{\varepsilon})$, which is based on an improved bound of $O(1/\varepsilon)$ on the size of "core-sets" as well as the use of second-order cone programming (SOCP) for solving subproblems. We have performed an experimental investigation to determine how the core-set size tends to behave in practice for a variety of input distributions. We show that substantially larger instances, both in terms of the number n of input points and the dimension d, of the MEB problem can be solved $(1+\varepsilon)$-approximately, with very small values of $\varepsilon > 0$, compared with the best known implementations of exact solvers. (We note that, since the original appearance of our paper in ALENEX [132], Fischer, Gärtner and Kutz [123] have announced significantly improved results with a new exact solver.) We also demonstrate that the sizes of the core-sets tend to be much smaller than the worst-case theoretical upper bounds.

Preliminaries. Throughout this chapter, S will be either a set of points in $\mathbb{R}^d$ or a set of balls. We let $n = |S|$.

We let $B_{c,r}$ denote a ball of radius r centered at point $c \in \mathbb{R}^d$. Given an input set $S = \{p_1, \ldots, p_n\}$ of n objects in $\mathbb{R}^d$, the *minimum enclosing ball MEB*(S) *of* S is the unique minimum-radius ball containing S. (Uniqueness follows from results of [92, 200]: if B_1 and B_2 are distinct minimum enclosing balls for S, then one can construct a smaller ball containing $B_1 \cap B_2$ and therefore containing S.) The center, c^*, of MEB(S) is often called the *1-center* of S, since it is the point of $\mathbb{R}^d$ that minimizes the maximum distance to

points in S. We let r^* denote the radius of $\text{MEB}(S)$. A ball $B_{c,(1+\varepsilon)r}$ is said to be $(1+\varepsilon)$-*approximation of MEB*(S) if $r \leq r^*$ and $S \subset B_{c,(1+\varepsilon)r}$.

Given $\varepsilon > 0$, a subset, $X \subseteq S$, is said to be an ε-*core-set* (or *core-set*) of S if $B_{c,(1+\varepsilon)r} \supset S$, where $B_{c,r} = \text{MEB}(X)$; in other words, X is a core-set if an expansion by factor $(1+\varepsilon)$ of its MEB contains S. Since $X \subseteq S$, $r \leq r^*$; thus, the ball $B_{c,(1+\varepsilon)r}$ is a $(1+\varepsilon)$-approximation of $\text{MEB}(S)$.

Related work. For small (fixed) dimension d, the MEB problem can be solved in $O(n)$ time for n points using the fact that it is an LP-type problem [92, 141]. One of the best implementable solutions to compute the MEB exactly in moderately high dimensions is given by Gärtner and Schönherr [101]; the largest instance they solve is in dimension $d = 300$ for $n = 10,000$ points (in about 20 minutes on their platform).

In comparison, the largest instance we solve[1] $(1+\varepsilon)$-approximately is in dimension $d = 1500$ for $n = 100,000$ points, with $\varepsilon = 10^{-3}$. Other implementations of exact solvers to which we compare our method include the algorithm of Gärtner [100] and the algorithm of the CGAL[2] library (based on the algorithm of Welzl [187]). For large dimensions, our approximation algorithm is found to be much faster than these exact solvers. We are not aware of other implementations of polynomial-time approximation schemes for the MEB problem.

Very recently, in a paper that appeared after the conference publication of our paper [132], Fischer, Gärtner and Kutz [123] gave a very fast algorithm to compute exact minimum enclosing balls of point sets in high dimensions. Their method is similar to the

[1] This instance took less than 17 minutes to solve.
[2] `http://www.cgal.org` (Version 2.4)

simplex method for solving the MEB problem [101, 123]. The current version of our implementation (improved after conference publication) seems to be competitive with their implementation for moderately small values of ε. (see figure 7.)

Bădoiu et al. [47] introduced the notion of core-sets and their use in approximation algorithms for high-dimensional clustering problems. In particular, they give an $O\left(\frac{nd}{\varepsilon^2} + \frac{1}{\varepsilon^{10}} \log \frac{1}{\varepsilon}\right)$-time $(1+\varepsilon)$-approximation algorithm based on their upper bound of $O(1/\varepsilon^2)$ on the size of core-sets; the upper bound on the core-set size is remarkable in that it does not depend on d. In comparison, our time bound (Theorem 2.2.1) is $O\left(\frac{nd}{\varepsilon} + \frac{1}{\varepsilon^{4.5}} \log \frac{1}{\varepsilon}\right)$.

Independent of our work, the MEB problem in high dimensions has been recently studied by Zhou et al. [200]. The authors consider two approaches, one based on reformulation as an unconstrained convex optimization problem and another based on a second-order cone programming (SOCP) formulation. Xu et al. [193] perform a comparison of four algorithms (including a randomized algorithm) for the computation of the minimum enclosing circle of circles in the plane ($d = 2$). Both studies reveal that solving the MEB problem using a direct SOCP formulation suffers from memory problems as the dimension, d, and the number of points, n, increase. Our approach in this chapter is to apply core-sets, in combination with SOCP, to design practical approximation algorithms for the MEB problem.

In parallel with our work, Bădoiu and Clarkson [46] independently obtained an $O(1/\varepsilon)$ bound on the size of core-sets. Most recently, Bădoiu and Clarkson [45] have obtained an upper bound of $\lceil 1/\varepsilon \rceil$ and shown that it is worst-case tight.

The proof of the tightness employs a lower bound construction, placing $d+1$ points at the vertices of a regular simplex in dimension $d = \lfloor 1/\varepsilon \rfloor$. Thus, the lower bound is based on having $d \geq \lfloor 1/\varepsilon \rfloor$.

In the experiments reported here, however, almost always the dimension d satisfies $d < \frac{1}{\varepsilon}$, and we find that, on a wide variety of input sets, the core-set size is smaller than $\min\{1/\varepsilon, d+1\}$; see figures 8 and 10. (Note that core-sets of size at most $d+1$ always exist for any $\varepsilon \geq 0$, since $d+1$ points suffice to determine a ball in $\mathbb{R}^d$.)

2.1 SOCP Formulation

The minimum enclosing ball (MEB) problem can be formulated as a second-order cone programming (SOCP) problem. SOCP is a class of convex optimization problems in which a linear function is minimized over an affine subset of products of second-order cones (also known as "Lorenz cones", or "quadratic cones"), defined as

$$K = \{(\sigma, x) \in \mathbb{R}^{1+d} : \|x\| \leq \sigma\}.$$

SOCP therefore can be viewed as an extension of linear programming in which the non-negative orthant is replaced by the product of second-order cones. Linear programming is a special case of SOCP since each non-negativity constraint is equivalent to a one-dimensional second-order cone constraint. As with the nonnegative orthant, K is a full-dimensional, convex cone in $\mathbb{R}^{d+1}$; however, K is not polyhedral for $d \geq 2$.

Recently, SOCP has received a lot of attention from the optimization community due to its applications in a wide variety of areas (see, e.g., [137, 9]) and due also to the existence of very efficient interior-point algorithms to solve this class of optimization problems.

The MEB problem for an input set $S = \{B_{c_i,r_i},\ i = 1,\ldots,n\}$ of n balls can be formulated as an SOCP problem as

$$\min_{c,r}\ r, \quad \text{s.t.} \quad \|c - c_i\| + r_i \leq r,\ i = 1,\ldots,n, \tag{1}$$

where $c \in \mathbb{R}^d$ and $r \in \mathbb{R}$ are the decision variables corresponding to the center and the radius of the MEB, respectively. Note that the formulation reduces to the usual MEB problem for point sets if $r_i = 0$ for $i = 1, \ldots, n$.

When applied to the nonlinear convex optimization problem (1), interior-point algorithms generate a sequence of interior feasible solutions $(c^k, r^k),\ k = 0, 1, 2, \ldots$ that converges to (c^*, r^*) in the limit, where $B_{c^*,r^*} := \text{MEB}(S)$. Note that an interior feasible solution (c^k, r^k) for (1) – i.e., a feasible solution that strictly satisfies all the inequalities – geometrically corresponds to a ball B_{c^k,r^k} that strictly contains all the balls in S. Consequently, interior-point algorithms converge to $\text{MEB}(S)$ through a sequence of strictly enclosing balls.

Given any relative error $\gamma > 0$, interior-point algorithms compute an interior feasible solution (c^k, r^k) such that

$$r^k - r^* \leq \gamma(r^0 - r^*) \tag{2}$$

in $O(\sqrt{n}\log(1/\gamma))$ iterations, where r^0 is the radius of the initial strictly enclosing ball from which the algorithm is initiated [148, 149, 162]. In the context of the MEB problem, one can easily find an initial enclosing ball with $r^0 = O(r^*)$, as the following lemma shows. This result will then be used to establish that a particular choice of γ yields a solution that is a $(1+\delta)$-approximation of $\text{MEB}(S)$.

Lemma 2.1.1 *Let $S = \{B_{c_i,r_i},\ i = 1, \ldots, n\}$ be a given set of n balls. One can compute a $\frac{1}{\sqrt{3}}$-approximation to the diameter of S in $O(nd)$ time.*

Proof. If S is viewed as an infinite collection of points in $\mathbb{R}^d$, the statement simply follows from the algorithm of Eğecioğlu and Kalantari [86] for the case of a finite point set. Pick any $p \in S$; find a point $q \in S$ that is furthest from p; find a point $q' \in S$ that is furthest from

q; output the pair (q, q'). It is easy to see that the same method applies to the case in which S is a set of balls, yielding again a $\frac{1}{\sqrt{3}}$-approximation. (Principal component analysis can be used to obtain the same approximation ratio for points but does not readily generalize to the case of balls.) Note that the furthest point in each ball $B_{c_i,r_i} \in S$ from a given point $p \in S$ can be computed in $O(d)$ time, yielding an overall time complexity of $O(nd)$. □

We now use Lemma 2.1.1 to construct an initial ball enclosing S. Let Δ denote the diameter of S and $(q, q') \in S$ denote the two points that yield the diameter approximation. Let $D := \|q - q'\|$. The following inequalities easily follow.

$$\frac{1}{\sqrt{3}} r^* \leq \frac{1}{\sqrt{3}} \Delta \leq D \leq \Delta \leq 2r^*, \tag{3}$$

where r^* is the radius of MEB(S). It follows from (3) that

$$r^* \leq \sqrt{3} D \leq 2\sqrt{3} r^*. \tag{4}$$

Consequently, the ball centered at q (or q') with radius $r_0 := 2D$ (or any radius strictly greater than D) strictly encloses S, and $r_0 \leq 4r^*$. In conjunction with (2), it follows that interior-point methods compute an enclosing ball B_{c^k,r^k} in polynomial time with the property that

$$r^k \leq r^*(1 + 3\gamma). \tag{5}$$

For any given $\delta > 0$, if we set $\gamma := \delta/3$, it follows that we get a $(1+\delta)$-approximation of MEB(S) in $O(\sqrt{n}\log(1/\delta))$ iterations.

The major work at each iteration of interior-point algorithms is the solution of a linear system involving a $(d+1) \times (d+1)$ symmetric and positive definite matrix (see, e.g., [9]). For the MEB problem, the matrix in question can be computed using $O(nd^2)$ basic arithmetic operations (flops), and its Cholesky factorization can be carried out in $O(d^3)$

flops. Therefore, the overall complexity of computing a $(1+\delta)$-approximation of MEB(S) with an interior-point method is $O(\sqrt{n}d^2(n+d)\log(1/\delta))$. In practice, we stress that the number of iterations seems to be $O(1)$ or very weakly dependent on n (see, for instance, the computational results with SDPT3 in [185]).

The worst-case complexity estimate reveals that the direct application of interior-point algorithms is not computationally feasible for large-scale instances of the MEB problem due to excessive memory requirements. In [200], the largest instance solved by an interior-point solver consists of 1000 points in 2000 dimensions and requires over 13 hours on their platform. However, large-scale instances can still be handled by an interior-point algorithm if the number of points n can somehow be decreased. This can be achieved by a filtering approach in which one eliminates points that are guaranteed to be in the interior of the MEB or by selecting a subset of points and solving a smaller problem and iterating until the computed MEB contains all the points. The latter approach is simply an extension of the well-known cutting plane approach initially developed for solving large-scale linear programs that have much fewer variables than constraints. The MEB problem formulated as in (1) above precisely fits in this framework since $n \gg d$ for instances of interest in this chapter. Due to the nonlinearity of the inequalities in (1), the boundary of each constraint is actually a nonlinear surface as opposed to a hyperplane. We therefore use the term "cutting plane" loosely in this chapter.

We use the cutting plane approach to be able to solve large-scale MEB instances, discovering a carefully selected subset of the constraints, corresponding to a core-set. The success of such an approach depends on the following factors:

- ▷ *Initialization:* The quality of the initial core-set is crucial, since a good approximation leads to fewer updates. Furthermore, a small core-set with a good approximation

yields MEB instances with relatively few points, which can efficiently be solved by an interior-point algorithm.

▷ *Subproblems:* The performance of a cutting plane approach is closely related to the efficiency with which each subproblem can be solved. We use the state-of-the-art interior-point solver SDPT3 [183] in our implementation.

▷ *Core-set Updates:* An effective approach should update the core-set in a way that seeks to minimize the number of subsequent updates.

In the following section, we describe our algorithm in more detail.

2.2 The Algorithm

Algorithm 1 Outputs a $(1+\varepsilon)$-approximation of MEB(S) and an $O(1/\varepsilon)$ -size core-set

Require: Input set $S \subset \mathbb{R}^d$ of points/balls, parameter $\varepsilon \in (0,1)$

$X \leftarrow \{q, q'\}$, where $q, q' \in S$ are given by the diameter approximation (Lemma 2.1.1)
$\delta \leftarrow \varepsilon^2/163$
loop
  Let $B_{c',r'}$ denote the $(1+\delta)$-approximation to MEB(X) returned by SOCP.
  if $S \subseteq B_{c',(1+\varepsilon/2)r'}$ **then**
    Return $B_{c',(1+\varepsilon/2)r'}$, X
  else
    $p \leftarrow \arg\max_{x \in S} \|c' - x\|$
  end if
  $X \leftarrow X \cup \{p\}$
end loop

Given a set S of n points or balls, our algorithm for approximating MEB(S) begins with computing an approximate minimum enclosing ball of a carefully chosen subset $X \subseteq S$. For our purposes, it suffices to obtain any constant factor approximation of the diameter Δ,

so we choose to use the two points given by Lemma 2.1.1 as our initial core-set X (Step 1, Algorithm 1).

Step 2 of our algorithm sets the parameter δ so that the final output is guaranteed to be a $(1+\varepsilon)$-approximation of MEB(S), as we prove in Section 2.3.

The main loop (steps 3 to 11) first computes the approximate MEB of the current subset $X \subseteq S$, using the SOCP solver. Step 5 checks if a $(1+\varepsilon/2)$-expansion of this ball contains S. If this is the case, then the algorithm returns this expanded ball and the current core-set as the solution; otherwise, the algorithm picks the furthest point in S from the center of the approximate minimum enclosing ball of X, adds it to X, and repeats the loop. (The rationale for using $\varepsilon/2$ in the expansion will be given in the next section.)

We establish the following results about our algorithm.

Theorem 2.2.1 *Algorithm 1 returns a $(1+\varepsilon)$-approximation to the MEB of a set of n balls in d dimensions in time $O\left(\frac{nd}{\varepsilon} + \frac{d^2}{\varepsilon^{3/2}}\left(\frac{1}{\varepsilon}+d\right)\log\frac{1}{\varepsilon}\right)$, which is $O\left(\frac{nd}{\varepsilon} + \frac{1}{\varepsilon^{4.5}}\log\frac{1}{\varepsilon}\right)$, if $d = O(1/\varepsilon)$ (as in [47]).*

Theorem 2.2.2 *Upon termination of Algorithm 1, X is a core-set of S and has size $O(1/\varepsilon)$.*

2.3 Analysis of the Algorithm

In this section, we present a detailed analysis of Algorithm 1. We begin with a basic lemma which holds for both sets of points and balls. The proof for the case of points (given in [47, 104]) requires only minor modification to apply to balls; we include it for completeness.

Lemma 2.3.1 *Let $B_{c,r}$ be the MEB of a set $S \subset \mathbb{R}^d$ of balls. Then any closed halfspace containing c contains at least one point in B_i, for some $i \in \{1,\ldots,n\}$, at distance r from c.*

Proof. We can assume that $B_i \neq B_{c,r}$ for all $i = 1 \ldots n$, since otherwise the proof is trivial. Let P_r be the set of points of $\cup_i B_i$ that are at distance exactly r from c; by our assumption, P_r is a discrete set of at most n points.

Suppose, to the contrary, that H is a closed halfspace containing c that contains no point of P_r. Let $\delta > 0$ be the minimum distance between P_r and H, and let $\rho < r$ be the maximum distance from c to a point of $S \cap H$. Pick a positive number $\varepsilon < \min\{\delta, r - \rho\}$. Then, if we translate the ball $B_{c,r}$ by a distance ε in the direction of the outward normal of H, the new ball, $B_{c',r}$, will still contain $\cup_i B_i$ and none of the points of balls of $\cup_i B_i$ lie on the boundary of $B_{c',r}$. (For $p \in \cup_i B_i \setminus H$, $||p - c'|| < ||p - c||$, and for $p \in \cup_i B_i \cap H$, $||p - c'|| < ||p - c|| + \varepsilon < r$.) Thus, B' can be shrunk, contradicting the optimality of $B_{c,r}$. □

The following lemma, which is a modification of the result proved in [47], establishes that each update of the core-set strictly increases the radius of the corresponding MEB.

Lemma 2.3.2 *Let $B_{c,r}$ be the MEB of a set $X \subset \mathbb{R}^d$ of balls. Let $q \in \mathbb{R}^d$ be such that $q \notin B_{c,(1+\varepsilon/3)r}$, for some $\varepsilon \in (0,1)$. Then, the radius of MEB($X \cup \{q\}$) is at least $\left(1 + \frac{\varepsilon^2}{33}\right) r$.*

Proof. Let $B_{c',r'} = \text{MEB}(X \cup \{q\})$. If $||c' - c|| < (\varepsilon/4)r$ then, by the triangle inequality, we have $||q - c'|| \geq ||q - c|| - ||c' - c|| \geq (1 + \varepsilon/3)r - (\varepsilon/4)r = (1 + \varepsilon/12)r$; thus, the radius, r', of $\text{MEB}(X \cup \{q\})$ must be at least $(1 + \varepsilon/12)r \geq (1 + \varepsilon^2/33)r$, for $\varepsilon \in (0,1)$. If $||c' - c|| \geq (\varepsilon/4)r$, then let H be the halfspace whose bounding hyperplane passes through c and is orthogonal to cc', with $c' \notin H$. By Lemma 2.3.1, there is a point $p \in H$ in some ball of X, with $||p - c|| = r$. Thus,

$$r' \geq ||p - c'|| = \sqrt{r^2 + ||c' - c||^2 - 2r||c' - c|| \cos\theta} \geq \sqrt{r^2 + \left(\frac{r\varepsilon}{4}\right)^2} \geq \left(1 + \frac{\varepsilon^2}{33}\right) r,$$

where the equality follows from the law of cosines with $\theta = \angle c'cp \geq \pi/2$, and the last inequality uses the assumption that $\varepsilon < 1$. Thus, in this case too we get that $r' \geq (1 + \varepsilon^2/33)r$, completing the proof. □

Next, we show that the fact that the interior-point algorithm of Section 2.1 returns only an approximation of the MEB does not jeopardize the asymptotic performance of our algorithm. We begin with the following lemma, which allows us to constrain any approximate MEB within a small shell that lies between a slightly smaller copy of the (exact) MEB and a slightly larger copy of the (exact) MEB.

Lemma 2.3.3 *Let $B_{c,r}$ be the MEB of a set $X \subset \mathbb{R}^d$ of balls. Let $B_{c',r'}$ be a $(1+\delta)$-approximation to the MEB. Then, $||c'-c|| \leq r\sqrt{\delta(\delta+2)}$ and*

$$B_{c,r-||c'-c||} \subseteq B_{c',r'} \subseteq B_{c,(1+\delta)r+||c'-c||}.$$

Proof. Let H be the halfspace whose bounding hyperplane passes through c and is orthogonal to cc', with $c' \notin H$. By Lemma 2.3.1, there is a point $p \in H$ in some ball of X, with $||p-c|| = r$. Since p is in some ball of X, and $X \subseteq B_{c',r'}$, we know that $||c'-p|| \leq r' \leq (1+\delta)r$. Using the law of cosines,

$$||c'-c||^2 = ||c'-p||^2 - r^2 + 2r||c'-c||\cos\theta \leq r'^2 - r^2 \leq \delta(\delta+2)r^2,$$

where $\theta = \angle c'cp \geq \pi/2$. Thus, $||c'-c|| \leq r\sqrt{\delta(\delta+2)}$, as claimed.

Now, for any $x \in B_{c,r-||c'-c||}$, we know that $||x-c|| \leq r - ||c'-c||$, so, by the triangle inequality and the fact that $r \leq r'$ we get that

$$||x-c'|| \leq ||x-c|| + ||c'-c|| \leq r \leq r',$$

implying that $x \in B_{c',r'}$. Thus, $B_{c,r-||c'-c||} \subseteq B_{c',r'}$.

Similarly, for any $x \in B_{c',r'}$, we know that $||x-c'|| \leq r'$, so, by the triangle inequality,

$$||x-c|| \leq ||x-c'|| + ||c'-c|| \leq (1+\delta)r + ||c'-c||,$$

implying that $x \in B_{c,(1+\delta)r+||c'-c||}$. Thus, $B_{c',r'} \subseteq B_{c,(1+\delta)r+||c'-c||}$. □

The following lemma establishes that a point outside of an expanded approximate MEB is guaranteed to be outside of the appropriately expanded exact MEB.

Lemma 2.3.4 *Let $B_{c,r}$ be the MEB of a set $X \subset \mathbb{R}^d$ of balls. Let $B_{c',r'}$ be a $(1+\delta)$-approximation of MEB(X). If $q \notin B_{c',(1+\varepsilon/2)r'}$, then $q \notin B_{c,(1+\varepsilon/3)r}$, provided that δ is chosen so that $\delta \leq \varepsilon^2/163$.*

Proof. By Lemma 2.3.3, $B_{c,(1+\varepsilon/2)(r-||c'-c||)} \subseteq B_{c',(1+\varepsilon/2)r'}$ (since $B_{c,r-||c'-c||} \subseteq B_{c',r'}$). For any point $q \notin B_{c',(1+\varepsilon/2)r'}$, then, we have

$$||q-c|| \geq (1+\varepsilon/2)(r-||c'-c||) \geq (1+\varepsilon/2)\left(r - r\sqrt{\delta(2+\delta)}\right),$$

using the fact, from Lemma 2.3.3, that $||c'-c|| \leq r\sqrt{\delta(\delta+2)}$. Thus, $q \notin B_{c,(1+\varepsilon/3)r}$ if

$$(1+\varepsilon/2)\left(r - r\sqrt{\delta(2+\delta)}\right) \geq (1+\varepsilon/3)r,$$

i.e., provided that

$$\sqrt{\delta(2+\delta)} \leq \frac{\varepsilon}{6+3\varepsilon},$$

or $\delta \leq \sqrt{1+\left(\frac{\varepsilon}{6+3\varepsilon}\right)^2} - 1$. For $\varepsilon < 1$, it is readily checked that $\varepsilon^2/163 \leq \sqrt{1+\left(\frac{\varepsilon}{6+3\varepsilon}\right)^2} - 1$; thus, it suffices to choose $\delta \leq \varepsilon^2/163$, as claimed. □

We next show that Algorithm 1 correctly returns a $(1+\varepsilon)$-approximation of MEB(S).

Lemma 2.3.5 *The ball returned by Algorithm 1 is a $(1+\varepsilon)$-approximation of MEB(S).*

Proof. Algorithm 1 returns the ball $B_{c',(1+\varepsilon/2)r'}$, where $B_{c',r'}$ is a $(1+\delta)$-approximation of $B_{c,r} = \text{MEB}(X)$ returned by the SOCP algorithm for the set $X \subseteq S$.

We know by step 5 that $S \subseteq B_{c',(1+\varepsilon/2)r'}$; thus, to show that $B_{c',(1+\varepsilon/2)r'}$ is a $(1+\varepsilon)$-approximation of MEB(S), we have to show that $(1+\varepsilon/2)r'$ is at most $(1+\varepsilon)$ times the radius of MEB(S). Since $X \subseteq S$, the radius of MEB(S) is at least r, the radius of MEB(X); thus, it suffices to show that $(1+\varepsilon/2)r' \leq (1+\varepsilon)r$.

By Lemma 2.3.3, $B_{c',r'} \subseteq B_{c,(1+\delta)r+||c'-c||}$, so

$$r' \leq (1+\delta)r + ||c'-c||.$$

Thus, it suffices to show that

$$(1+\varepsilon/2)(1+\delta)r + (1+\varepsilon/2)||c'-c|| \leq (1+\varepsilon)r,$$

or,

$$||c'-c|| \leq \left(\frac{\varepsilon}{2+\varepsilon} - \delta\right) r.$$

From Lemma 2.3.3, we know that $||c'-c|| \leq r\sqrt{\delta(\delta+2)}$. Then, using the assumption that $\varepsilon < 1$, we see that it suffices to show that

$$\sqrt{\delta(\delta+2)} \leq \frac{\varepsilon}{2+1} - \delta = \frac{\varepsilon}{3} - \delta.$$

This last inequality holds if $\delta \leq \varepsilon^2/(18+6\varepsilon)$, which certainly holds with our choice of $\delta = \varepsilon^2/163$ in Algorithm 1. □

Lemma 2.3.6 *The set X returned by Algorithm 1 is an ε-core-set of S.*

Proof. Let $B_{c,r} = \text{MEB}(X)$ be the MEB of the set X returned by the algorithm. We know, by step 5, that at the conclusion of the algorithm, $S \subseteq B_{c',(1+\varepsilon/2)r'}$, where $B_{c',r'}$ is a $(1+\delta)$-approximation of $B_{c,r} = \text{MEB}(X)$.

In order to prove that X is an ε-core-set of S, we must show that $S \subseteq B_{c,(1+\varepsilon)r}$. Since $S \subseteq B_{c',(1+\varepsilon/2)r'}$, it suffices to show that $B_{c',(1+\varepsilon/2)r'} \subseteq B_{c,(1+\varepsilon)r}$.

Now, by Lemma 2.3.3, $B_{c',(1+\varepsilon/2)r'} \subseteq B_{c,(1+\varepsilon/2)((1+\delta)r+||c'-c||)}$. Thus, it suffices to show that

$$(1+\varepsilon/2)\left((1+\delta)r+||c'-c||\right) \leq (1+\varepsilon)r.$$

Since, by Lemma 2.3.3, $||c'-c|| \leq r\sqrt{\delta(\delta+2)}$, and $\delta \leq \sqrt{\delta(\delta+2)}$, it suffices that

$$(1+\varepsilon/2)\left(1+2\sqrt{\delta(\delta+2)}\right) \leq 1+\varepsilon,$$

or,

$$\sqrt{\delta(\delta+2)} \leq \frac{\varepsilon}{2(2+\varepsilon)}.$$

Since $\varepsilon < 1$, it suffices if δ is chosen so that $\sqrt{\delta(\delta+2)} \leq \varepsilon/6$, which holds if $\delta \leq \varepsilon^2/73$, and therefore holds for our choice of $\delta = \varepsilon^2/163$ in Algorithm 1. □

We are now ready to prove Theorems 2.2.1 and 2.2.2 stated at the end of Section 2.2. First, we note that an upper bound of $O(1/\varepsilon^2)$ on the size of a core-set is straightforward: By Lemma 2.1.1, the radius of MEB(X) at the first iteration of the algorithm (when $X = \{q, q'\}$) is at least $\Delta/(2\sqrt{3})$, where Δ is the diameter of S. By Lemmas 2.3.2 and 2.3.4, each point added to X increases the radius of the corresponding minimum enclosing ball by at least $\frac{\Delta}{2\sqrt{3}}\frac{\varepsilon^2}{O(1)}$. Since the radius of MEB(S) is less than the diameter Δ, the loop will be executed $O(1/\varepsilon^2)$ times. We now show that a more careful analysis yields the improved bound of Theorem 2.2.2.

Proof.[of Theorem 2.2.2] Our proof is based on a careful analysis of the number of times the loop (steps 3-11) is executed in Algorithm 1. Without loss of generality, we assume that $\varepsilon = 1/2^m$. We will obtain an upper bound on the number of points added to X in order to obtain a $(1+\varepsilon_i)$-approximation of MEB(S), where $\varepsilon_i := 1/2^i$, $i = 1, \ldots, m$.

Note that, since each iteration adds a point to X, the radius of MEB(X) monotonically increases; thus, once the set X becomes an ε_i-core-set of S, it remains an ε_i-core-set. We consider the iterations of the algorithm to be partitioned into *m rounds*. We consider round i to begin when X first becomes an ε_{i-1}-core-set of S and to end when X first becomes an ε_i-core-set of S. Note that a round may start and end at the same instant, since it may be that when X first becomes an ε_{i-1}-core-set, it also may become an ε_i-core-set (indeed, it may be that MEB(X) is now equal to MEB(S)). Also note that our algorithm does not determine exactly when the transitions occur between rounds, since we compute $(1+\delta)$-approximations of the MEB, not the exact MEB; however, the decomposition into rounds as defined above is useful in the analysis of the algorithm.

Let X_i denote the set X at the conclusion of round i (and thus at the start of round $i+1$), let τ_i denote the number of points of S added to X_{i-1} during round i (i.e., τ_i is the number of iterations in round i), and let r_i denote the radius of MEB(X_i).

Consider an iteration of the algorithm during round i. Let X be the current value of the subset of S, and let $B_{c,r} = \text{MEB}(X)$. The algorithm does not compute $B_{c,r}$ but does compute $B_{c',r'}$, a $(1+\delta)$-approximation of $B_{c,r}$. The algorithm then selects a point $p \in S$ to be added to X that maximizes the distance from the center, c', of $B_{c',r'}$. Since we know, by definition of rounds, that X is *not* an ε_i-core-set until the *end* of round i, it must be that p lies outside of the ball $B_{c',(1+\varepsilon_i/2)r'}$. (Otherwise, by Lemma 2.3.6, the current set X is an ε_i-core-set, since $\delta \leq \varepsilon^2/163 \leq \varepsilon_i^2/163$, meaning that the round is over.) Thus, by Lemma 2.3.4, we know that p must lie outside of $B_{c,(1+\varepsilon_i/3)r}$. Then, by Lemma 2.3.2, we know that the radius of MEB(X) goes up by at least $r\varepsilon_i^2/33$ with the addition of p to X. The radius r goes up with each iteration of the round i; thus, at each iteration r is bounded below by r_{i-1}. Since the round starts with a set X_{i-1} whose MEB radius is r_{i-1} and ends with a set X_i whose

MEB radius is r_i, with each iteration increasing the radius by at least $r_{i-1}\varepsilon_i^2/33$, we know that the total number τ_i of iterations during round i obeys

$$\tau_i \leq \frac{r_i - r_{i-1}}{r_{i-1}\varepsilon_i^2/33} = 33\left(\frac{r_i}{r_{i-1}} - 1\right)2^i \leq 33\left(\frac{\Delta}{\Delta/2\sqrt{3}} - 1\right)2^i = 33\left(2\sqrt{3} - 1\right)2^i,$$

where Δ is the diameter of the set S, and we have used the facts that $r_i \leq \Delta$ and $r_{i-1} \geq \Delta/2\sqrt{3}$. Finally, this implies that the total number of iterations over all rounds is

$$|X_m| = 2 + \sum_{i=1}^{m} \tau_i = \mathsf{O}(2^m) = \mathsf{O}(\frac{1}{\varepsilon}) .$$

□

Proof.[of Theorem 2.2.1] Since the size of the core-set is $\mathsf{O}(1/\varepsilon)$, each call to our SOCP solver takes time $\mathsf{O}(\frac{d^2}{\sqrt{\varepsilon}}\left(\frac{1}{\varepsilon}+d\right)\log\frac{1}{\varepsilon})$. We parse through the input $\mathsf{O}(1/\varepsilon)$ times. At each iteration, it takes $\mathsf{O}(nd)$ time to identify the furthest point. Therefore, the total running time is $\mathsf{O}(\frac{nd}{\varepsilon} + \frac{d^2}{\varepsilon^{3/2}}\left(\frac{1}{\varepsilon}+d\right)\log\frac{1}{\varepsilon})$. Putting $d = \mathsf{O}(1/\varepsilon)$, as in [47], we get a total time bound of $\mathsf{O}(\frac{nd}{\varepsilon} + \frac{1}{\varepsilon^{4.5}}\log\frac{1}{\varepsilon})$. □

Remark 1: The improved core-set bound of Theorem 2.2.2 gives, as an immediate consequence, also improved time bounds over those of [47] for 2-center clustering (improving $2^{\mathsf{O}(1/\varepsilon^2)}dn$ to $2^{\mathsf{O}(1/\varepsilon)}dn$) and for k-center clustering (improving $2^{\mathsf{O}((k/\varepsilon^2)\log k)}dn$ to $2^{\mathsf{O}((k/\varepsilon)\log k)}dn$).

Remark 2: The time bound of Theorem 2.2.1 can be further reduced to $\mathsf{O}\left(\frac{nd}{\varepsilon} + \frac{1}{\varepsilon^4}\log^2\frac{1}{\varepsilon}\right)$ by using a recent algorithm due to Har-Peled [110], which can compute a $(1+\varepsilon)$-approximation to the minimum enclosing ball of n points in d dimensions in $\mathsf{O}(\frac{nd}{\varepsilon}\log^2\frac{1}{\varepsilon})$ time. This is slightly better than our running time and does not use SOCP.

Remark 3: The conference version of our paper [132] had all the experiments done with δ set to $\mathsf{O}(\varepsilon)$ instead of $\mathsf{O}(\varepsilon^2)$. For the purposes of experimentation, this is not

really an issue, since in most cases setting δ anywhere below ε results in the same radius and core-set, as we have found experimentally. However, from a theoretical perspective, there was an oversight in [132], in that our analysis was based on the assumption that the SOCP solver returned an exact MEB. We have addressed this issue here. We note that a similar oversight apparently occurs in the first core-set paper [47], in which the ellipsoid algorithm is called with δ set to $O(\varepsilon)$ instead of $O(\varepsilon^2)$. A more careful analysis, such as the approach we present here, is needed in order to guarantee that the algorithm for *k*-center clustering of [47] indeed yields a $(1+\varepsilon)$-approximate solution, given that the convex programming techniques give an inexact solution.

2.4 Implementation and Experiments

We have implemented Algorithm 1 and report below results of experimentation with it. For comparison, we have also implemented a second algorithm, based on a variant of Algorithm 1, which we devised in an attempt to improve the running time of Algorithm 1 in practice. We refer to the (original) implementation of Algorithm 1 as the *pure implementation* and refer to the variant as the *fast implementation*.

The fast implementation attempts to address the main bottleneck in Algorithm 1, which we found to be the time spent in calls to the SOCP solver that computes the (approximate) minimum enclosing ball of the set X after each new point (the furthest outlier) is added. With each iteration, this computation is performed from scratch. In an attempt to compute the "easy" core-set points more quickly, and reduce the number of calls to the SOCP solver, we developed our fast implementation based on a hybrid algorithm that combines our Algorithm 1 with some ideas of Bădoiu and Clarkson [46]. In their simple gradient-descent method, at each iteration the current center is shifted towards the furthest outlier, resulting

in a sequence of centers that converge to the center of the minimum enclosing ball. Bădoiu and Clarkson establish that a simple updating scheme returns a $(1+\varepsilon)$-approximation in $O(nd/\varepsilon^2)$ iterations. As ε decreases, the running time deteriorates (in practice), with the $1/\varepsilon^2$ term becoming quite significant. On the other hand, we find that our Algorithm 1 performs well for small ε, even better than theoretical worst-case analysis suggests. Thus, in order to maintain the advantages of both algorithms, in our fast implementation we first apply the algorithm of Bădoiu and Clarkson for a prespecified number of iterations and record the furthest outliers at each iteration; our experiments show that $d/3$ is a good choice for the number of these iterations. We then apply our Algorithm 1 as a second phase, using the initial choice of X to be the set of points that show up as furthest outliers in the $d/3$ iterations of the first phase. We note that some of the points may appear as furthest outliers in more than one iteration of the first phase. This often means that the initial size of X is smaller than the number of iterations in the first phase; e.g., $|X| = 30$ after the first phase of 85 iterations on the USPS data set (see figure 11). The fast implementation has the potential advantage of obtaining quickly a fairly good approximation to the core set using a simple algorithm (not based on an SOCP solver); then, only a few more iterations of Algorithm 1 (using an SOCP solver) are usually needed to complete the core-set computation.

Most of our code is written in Matlab. However, in order to enhance the performance, some of the subroutines (e.g., computing the furthest outlier) were written in C and linked to the Matlab code using mex files. Our software is fairly compact and is available on the web[3] . The current implementation takes only point sets as input; extending it to input sets of balls should be relatively straightforward.

For the SOCP component of the algorithm, we considered two leading SOCP solvers

[3] `http://www.compgeom.com/meb/`

that are freely available: SeDuMi [175] and SDPT3 [183]. Experimentation showed SDPT3 to be superior to SeDuMi for use in our application, so our results here are reported using SDPT3. We refer the reader to the web site[4] maintained by Hans Mittelmann for an independent benchmarking of a variety of optimization codes.

In an attempt to minimize the size of the core-set, our implementations find the furthest outlier at each iteration. We should emphasize, however, that the running times of our algorithms can be reduced by introducing random sampling at the outlier detection stage (see, e.g., Pellegrini [153]), at the expense of slightly larger core-sets for large-scale problems.

Another desirable property of our implementation is that it is I/O-efficient if we assume that we can solve $O(1/\varepsilon)$ -size subproblems in internal memory. (This was always the case for our experiments, since the size of the core-set did not even approach $1/\varepsilon$ in practice.) With this assumption, the current implementation in the I/O model does at most $O(nd/B\varepsilon)$ I/Os, where B denotes the disk block size, and the same bound also generalizes to the cache-oblivious model [97]. We believe that with an efficient implementation (e.g., in C++) of our algorithm, very large problems ($n \approx 10^7, d \approx 10^4, \varepsilon \approx 10^{-5}$) are tractable to solve in practice on current state-of-the-art systems with sufficient memory and hard disk space.

Platform. All of the experimental results reported in this chapter were done on two platforms. The main platform we used to do the experiments was a dual-processor Intel(R) Xeon(TM) 2.66GHz system with 2GB RAM, running Windows XP/Matlab 6.5.0 Release 13. Unfortunately, Matlab is a single-threaded application, so was using only 1 CPU. Figures 7–13 correspond to experiments on this platform.

Figures 14 and 15 were generated from experiments done on a Pentium III 1GHz,

[4] `http://plato.asu.edu/bench.html`

512MB notebook computer, running Windows 2000. All of the experiments in the conference version of our paper [132] were conducted also on this platform. Our new implementation, on the new platform (Xeon 2.66GHz), resulted in significantly different results than were reported in [132]; thus, all of the results reported here (except Figures 14 and 15) are new, using the Xeon 2.66GHz platform. Figures 14 and 15 report comparison results of our algorithm with two others; due to some software availability issues, we have not yet been able to conduct the same comparison on the new platform, but we expect the relative performances to be comparable.

Datasets. Most of our experiments were conducted on randomly generated point data, according to various distributions. We also experimented with the USPS data,[5] which is a dataset of feature vectors extracted from handwritten characters, made available by the US Postal service. The USPS data contains 7291 points in 256 dimensions and is a standard data set used in the clustering and machine learning literature. For generating random point data, we used Matlab to generate random matrices, with `rand` for uniformly distributed data, `randn` for normally distributed data, and `random` for other specific distributions. Specifically, we considered the following four classes of point data:

- uniformly distributed within a unit cube;
- uniformly distributed on the vertices of a unit cube;
- normally distributed in space, with each coordinate chosen independently according to a normal distribution with mean 0 and variance 1;
- Poisson distributed, with each coordinate drawn from a Poisson distribution with

[5] `http://www.kernel-machines.org/data/ups.mat.gz`, 29MB

parameter $\lambda = 1$.

Methods for comparison. Bernd Gärtner [100] provides a code on his web site that we used for comparison. We also used the CGAL 2.4 implementation (based on Welzl's algorithm with move-to-front for small instances and a heuristic for large instances). We were not able to compile code available from David White's web page[6] . We were unable to replicate the timings reported in the paper of Gärtner and Schönherr [101], since the version of the implementation of their algorithm in CGAL 2.4 was not robust. While preparing this chapter, a recently updated version of the implementation became available in the latest release of CGAL; in future work, we will be conducting experiments for comparison with it.

Fischer, Gärtner and Kutz [123] recently presented a very fast exact method to solve the MEB problem for points. A direct comparison of running times of their method to our method does not seem appropriate, since we compute approximate solutions while they compute exact solutions. Our implementation is in Matlab (which prevents us from doing experiments on very large data sets in very high dimension), while their implementation is in C++. A theoretical drawback of their simplex approach for solving MEBs is that a polynomial running time cannot be guaranteed, although in practice they show that the method is very fast, apparently much faster than the earlier implementation of our algorithm reported in [132].

Experimental results. We begin with a comparison of the fast implementation of Algorithm 1 with the pure implementation of Algorithm 1. In figures 5 and 6 we show how the running times and the core-set sizes vary with the dimension, for $n = 10^4$ points that are

[6] http://vision.ucsd.edu/~dwhite

normally distributed (with mean $\mu = 0$ and variance $\sigma = 1$) and $\varepsilon = 0.001$. Note that the fast implementation generates core-sets of essentially the same size as the pure implementation, but does so much more quickly.

In figure 7 we show how the running time of our fast implementation of Algorithm 1 varies with dimension, for n points that are normally distributed (with mean $\mu = 0$ and variance $\sigma = 1$) and $\varepsilon = 0.001$. The plot shows two choices of n: $n = 10^4$ and 10^5. Corresponding to the same experiment, also with the fast implementation, figure 8 shows how the core-set size varies with dimension.

In figures 9 and 10 we show how the running time and the core-set size varies with dimension for each of the four distributions of $n = 10^4$ input points, with $\varepsilon = 0.001$. Notable are the timings for points randomly chosen from the vertices of a hypercube; this distribution of cospherical points represented the most time-consuming instances for the algorithm. While we do not fully understand the non-monotone behavior with respect to dimension in this case, it seems to be related to the similar phenomenon observed and discussed (briefly) in [123].

We note that, while the core-set size is seen to increase somewhat with dimension, the observed size of the core-set in all of our experiments is substantially less than the worst-case (dimension-independent) theoretical upper bound of $O(1/\varepsilon)$. The upper bound, $\lceil 1/\varepsilon \rceil$, of [45] is 1000 (for our choice of $\varepsilon = 0.001$), while the core-set sizes experimentally are observed to be in the range 30-170.

Figures 11 and 12 plot the running times and core-set sizes, as a function of $\log_2(1/\varepsilon)$, for points that correspond to feature vectors extracted from handwritten characters, provided by the USPS. For comparison, we also plot the results for normally distributed points

of the same dimension, $d = 256$. On the USPS data, the core-set size increases approximately logarithmically in $1/\varepsilon$, as compared with the theoretical linear upper bound $O(1/\varepsilon)$. Note that the fast implementation always returns a core-set of size 30, even though it runs for 85 iterations. We also noted that the fast implementation did not add any points to the core-set in the second phase; all 30 points were inserted during the first phase.

In figure 13 we show timing results for low dimensions ($d = 2, 3$), as a function of $\log_{10} n$, for n normally distributed points. Plots are shown with two choices of ε ($\varepsilon = 10^{-3}$ and $\varepsilon = 10^{-6}$); however, the running times are essentially independent of the choice of ε. In all of these experiments, the core-set size was always less than 10. This suggests that approximate 2-center clusterings may be computed in time $O(2^{10}n)$ for low dimensions. It would be interesting to see if this result can lead to a truly practical method for approximate 2-center clustering in low dimensions; such a method may have applications in constructing effective hierarchies of bounding spheres.

Figure 14 shows a timing comparison between our algorithm, the CGAL 2.4 implementation, and Bernd Gärtner's code available from his website. (The experiments are done on a Pentium III 1GHz, 512MB notebook computer, running Windows 2000.) Both of these codes assume that the dimension of the input point set is fixed, and each has a threshold dimension beyond which the computation time is seen to increase sharply.

In figure 15 we compare the running times, as a function of dimension, of our pure implementation (using $\varepsilon = 0.001$) with the simple method of Bădoiu and Clarkson [46] (based on Claim 3.1 in that paper), using three choices of ε ($\varepsilon = 0.1, 0.05, 0.03$). These experiments are done on a set of $n = 1000$ points that are normally distributed ($\mu = 0$, $\sigma = 1$). (The experiments are done on a Pentium III 1GHz, 512MB notebook computer, running Windows 2000.) Note that for $\varepsilon = 0.03$, the Bădoiu-Clarkson algorithm is already

very slow compared to Algorithm 1. We have not yet implemented the main algorithm proposed in [46], which has a slightly lower running time ($O(nd/\varepsilon + (1/\varepsilon)^5)$) than our Algorithm 1, but it seems that when ε is small, its running time may suffer because of the base case solver (the simple method, based on Claim 3.1, which we tested). We suspect that the improved algorithm suggested by Har-Peled [110] (Remark 2, Section 2.3) is a better candidate for implementation.

Finally, we remark that in all of our experiments, we set $\delta = \varepsilon^2$, ignoring the constant that we derived in the theoretical analysis. In justification of this choice, we verified first experimentally that varying δ had little or no effect: Running times varied only slightly, and core-set sizes and MEB radii did not change at all. For instance, for $n = 5000$ points in dimension $d = 500$, with $\varepsilon = 0.001$, we varied δ from 10^{-3} to 10^{-9} for two different distributions of input points. For a set of points generated from a normal distribution, for all choices of δ, the core-set size was 75, the radius was 24.094 and the running time varied from 81.328 seconds to 81.922 seconds. For a set of points generated randomly from the vertices of a hypercube, the core-set size and the radius were again constant for all choices of δ, while the running time varied from 691.7 to 669.64 seconds. (Note that actually it took less time to compute using $\delta = 10^{-9}$ than using $\delta = 10^{-3}$.) In fact, we have not yet found a point set on which setting $\delta = \varepsilon^2$ instead of $\delta = \varepsilon$ made any change in either the core-set size or the MEB radius. This suggests that our theoretical analysis justifying the choice of δ to guarantee a $(1+\varepsilon)$-approximation is in fact overly conservative.

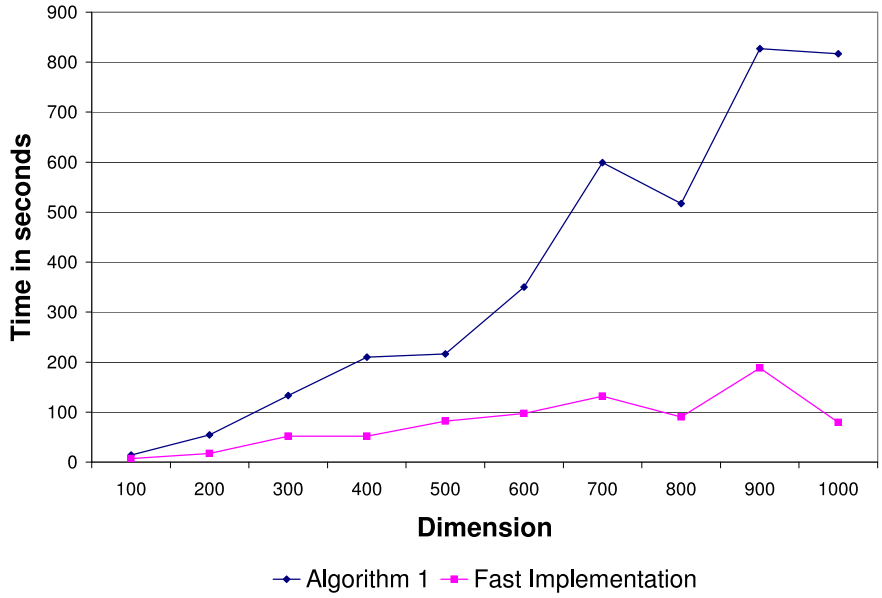


Figure 5: Running time (in seconds) as a function of dimension, for $n = 10^4$ input points that are normally distributed ($\mu = 0, \sigma = 1$). For comparison, we plot both the pure implementation ("Algorithm 1") and the fast implementation. Here, $\varepsilon = 0.001$.

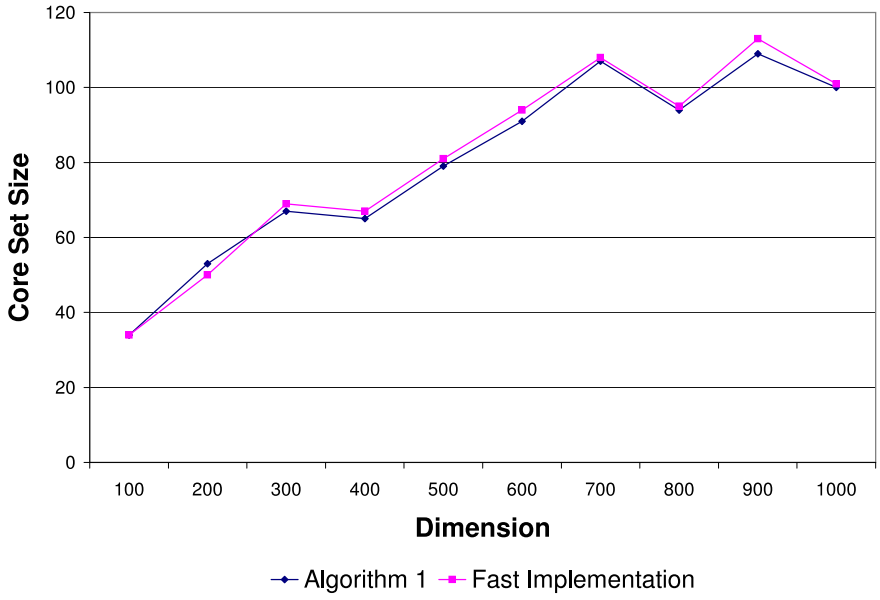


Figure 6: Core-set size as a function of dimension, for $n = 10^4$ input points that are normally distributed ($\mu = 0, \sigma = 1$). For comparison, we plot both the pure implementation ("Algorithm 1") and the fast implementation. Here, $\varepsilon = 0.001$.

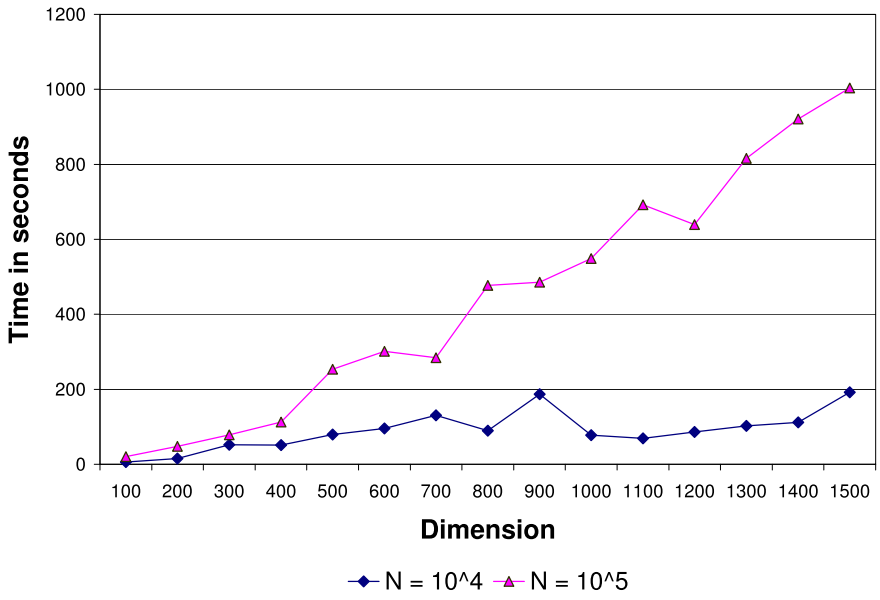


Figure 7: Running time (in seconds) of the fast implementation of Algorithm 1 as a function of dimension, for two choices of n ($n = 10^4$, $n = 10^5$). Here, $\varepsilon = 0.001$, and the input points are normally distributed ($\mu = 0, \sigma = 1$).

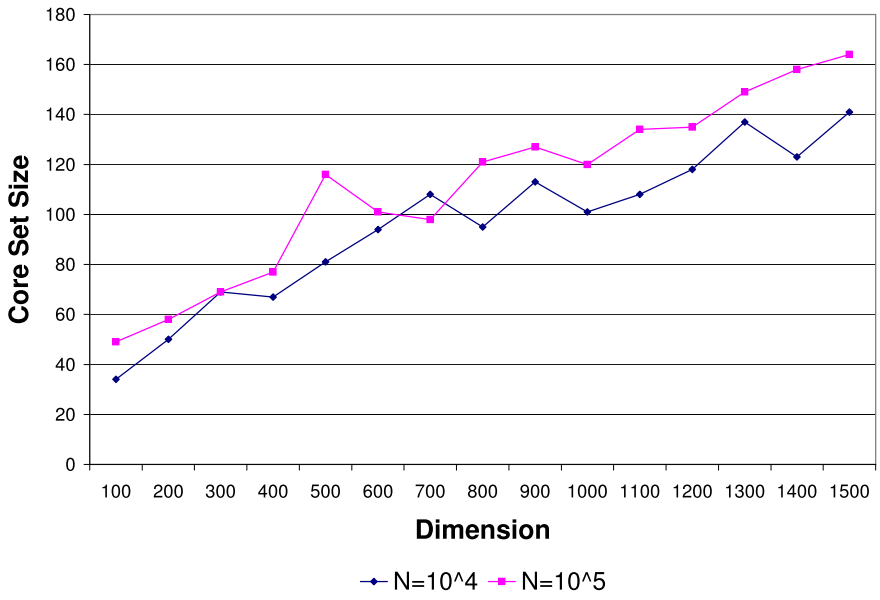


Figure 8: Core-set size as a function of dimension, for two choices of n ($n = 10^4$, $n = 10^5$). Here, $\varepsilon = 0.001$, and the input points are normally distributed ($\mu = 0, \sigma = 1$). The fast implementation of Algorithm 1 was used in this experiment.

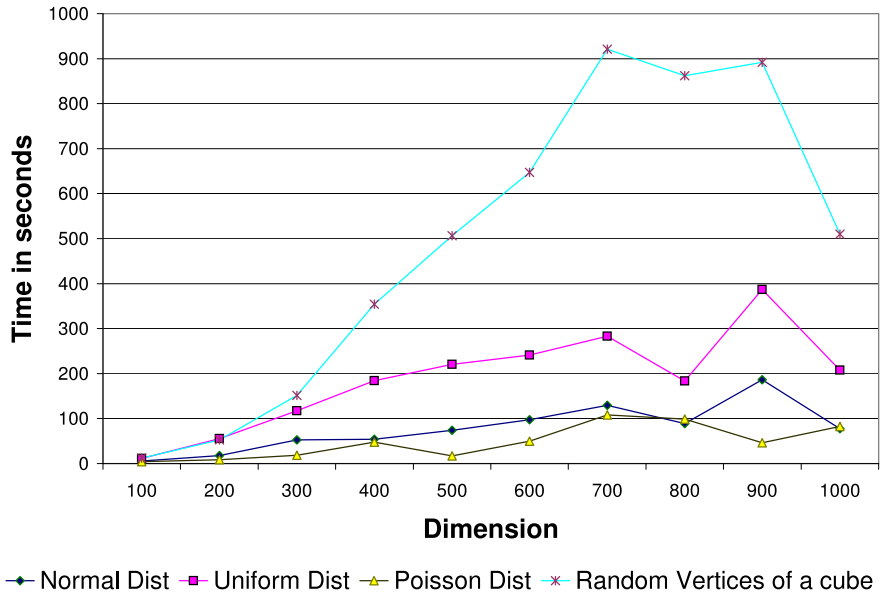


Figure 9: Running time (in seconds) of the fast implementation of Algorithm 1 as a function of dimension, for $n = 10^4$ input points from each of four distributions: uniform, normal, Poisson, and random vertices of a cube. Here, $\varepsilon = 0.001$.

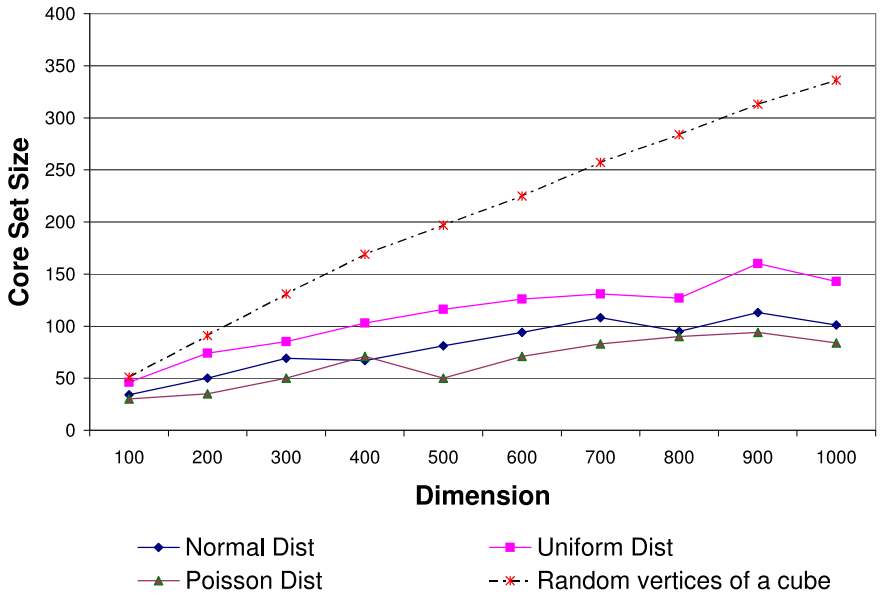


Figure 10: Core-set size for the fast implementation of Algorithm 1 as a function of dimension, for $n = 10^4$ input points from each of four distributions: uniform, normal, Poisson, and random vertices of a cube. Here, $\varepsilon = 0.001$.

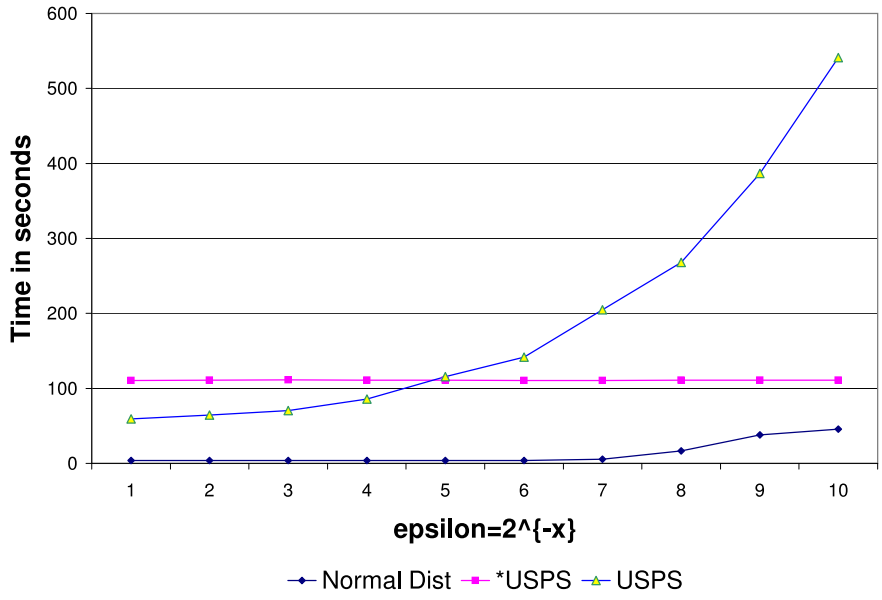


Figure 11: Running time (in seconds) of the fast implementation of Algorithm 1 as a function of $\log_2(1/\varepsilon)$, for input points that are normally distributed ($\mu = 0, \sigma = 1$) in dimension $d = 256$ and for input points from the USPS. For the normally distributed points, the fast implementation is used. For the USPS data, we plot results both for the pure implementation (indicated by "USPS") and for the fast implementation (indicated by "*USPS"). The USPS data contains 7291 points in 256 dimensions and is a standard data set, based on digitized hand-written characters, used in the clustering and machine learning literature.

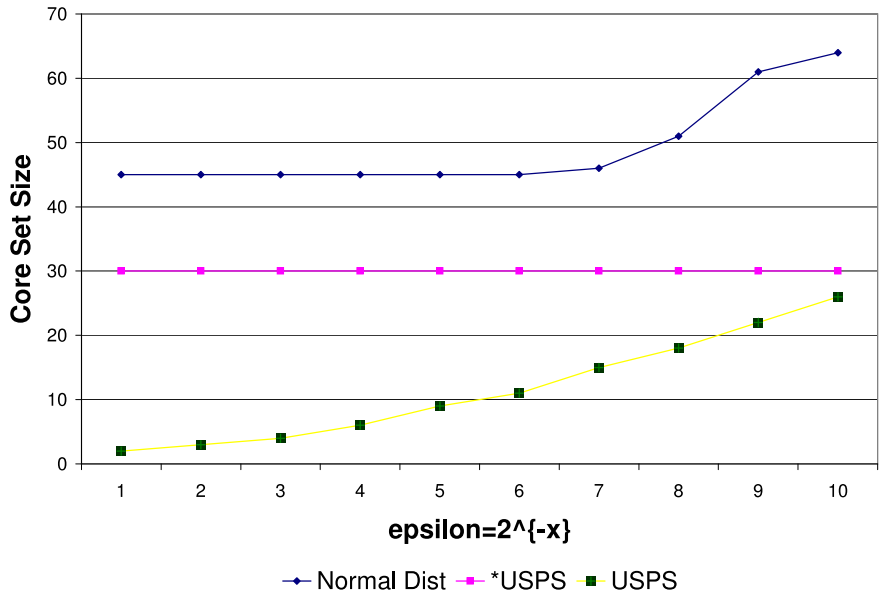


Figure 12: Core-set size as a function of $\log_2(1/\varepsilon)$, for input points that are normally distributed ($\mu = 0, \sigma = 1$) in dimension $d = 256$ and for input points from the USPS. For the normally distributed points, the fast implementation is used. For the USPS data, we plot results both for the pure implementation (indicated by "USPS") and for the fast implementation (indicated by "*USPS").

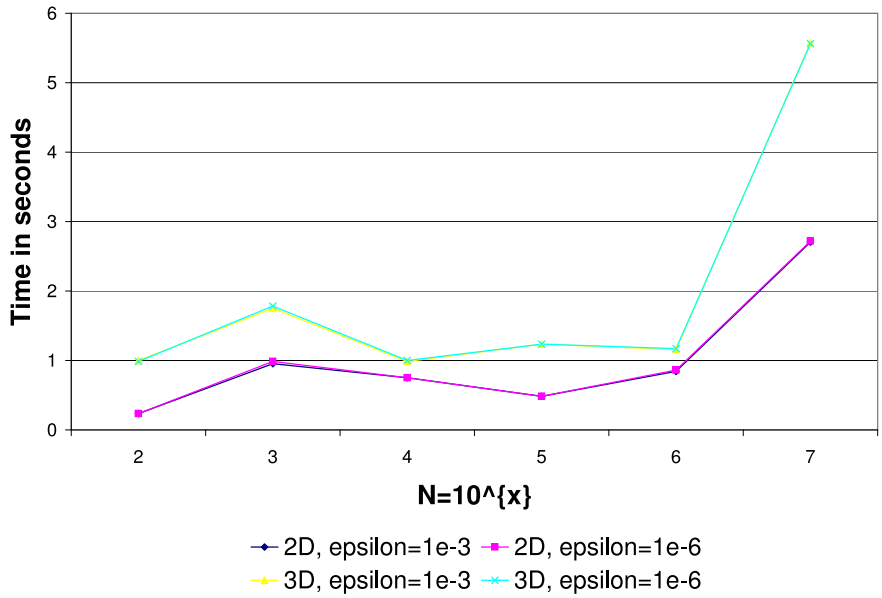


Figure 13: Running time (in seconds) of the fast implementation of Algorithm 1 as a function of $\log_{10} n$ for n input points that are normally distributed ($\mu = 0, \sigma = 1$) in dimensions $d = 2$ and $d = 3$. For each choice of d, plots are shown for two choices of ε ($\varepsilon = 10^{-3}$ and $\varepsilon = 10^{-6}$), but they are essentially identical, with no discernible difference. In every case, the core-set size was less than 10.

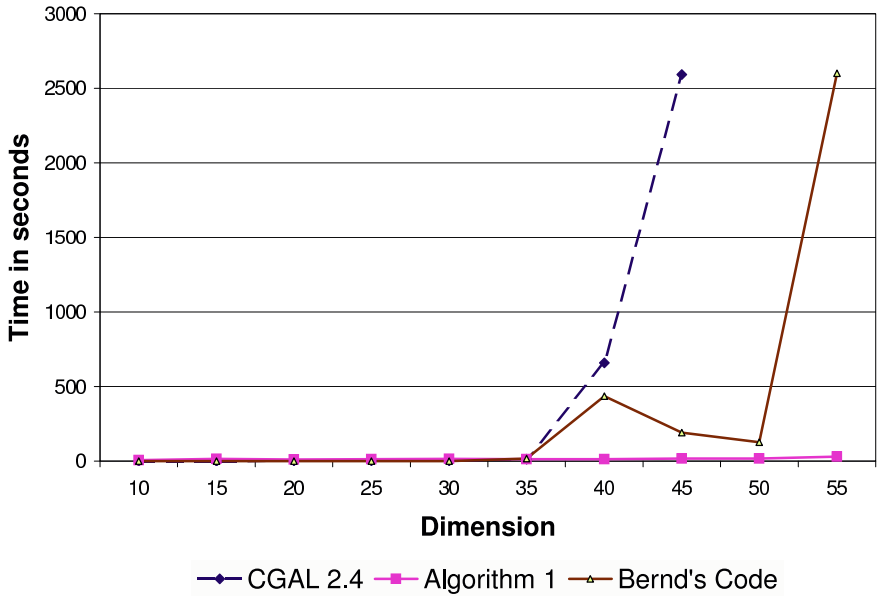


Figure 14: Timing comparison, as a function of dimension, for $n = 1000$ points that are normally distributed ($\mu = 0, \sigma = 1$). We compare the *pure implementation* of Algorithm 1 with CGAL 2.4 and with Bernd Gärtner's code. Here, $\varepsilon = 10^{-6}$. These experiments were done on a Pentium III 1GHz, 512MB notebook computer, running Windows 2000.

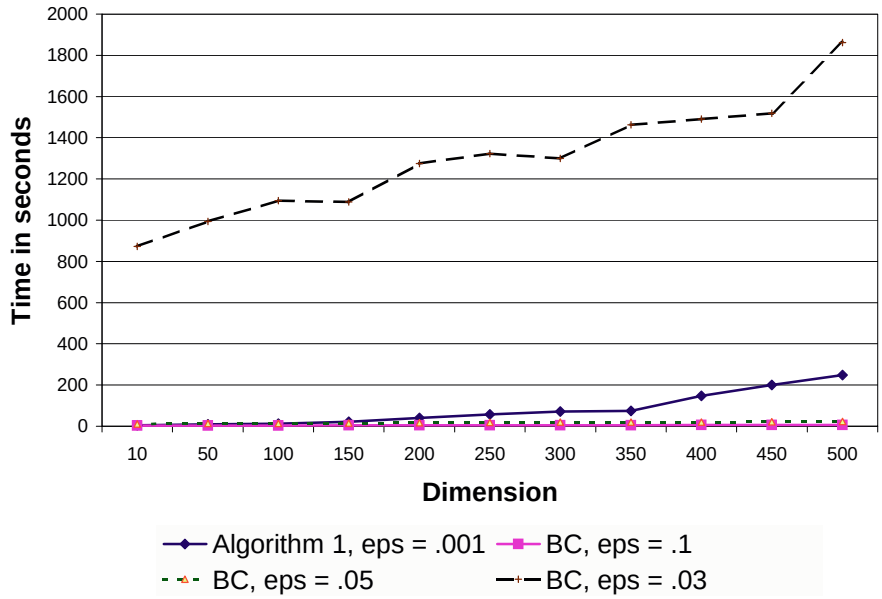


Figure 15: Timing comparison, as a function of dimension, for $n = 1000$ points that are normally distributed ($\mu = 0, \sigma = 1$). We compare the *pure implementation* of Algorithm 1 (using $\varepsilon = 0.001$) with the simple method of Bădoiu and Clarkson (BC), using three choices of ε ($\varepsilon = 0.1, 0.05, 0.03$). These experiments were done on a Pentium III 1GHz, 512MB notebook computer, running Windows 2000. As ε approaches zero, the performance of the BC algorithm degrades substantially.

2.5 k-center clustering

We also implemented a simple $(1+\varepsilon)$-approximation algorithm for k-center clustering for fixed d and k and report experimental results here. Although the theoretical bounds on the running time are hopeless for $1+\varepsilon$ approximating k-center clusters, we note that for dimensions 2 and 3, k-center is practical for $\varepsilon \geq 0.01$ and $k \leq 4$.

The running time of our implementation is $O(2^{\frac{k \log k}{\varepsilon}} dn)$. The k-center algorithm is based upon exhaustive enumeration of the core sets. The algorithm works for balls and constant complexity polytopes ($\mathcal{V}$-representation). It can be extended to work with ellipses and non convex polygons. It works in higher dimensions though due to the increase of core set size as the dimension goes high, its almost infeasible to compute k-centers for higher dimensions as opposed to 1-center. The algorithm we present here is extremely easy to code.

Before we move on to the algorithm, let's look at an easy instance of the problem and analyze its running time. For the 5-center clustering of German cities shown in figure 17, the number of cities is $n = 15112$, $\varepsilon = 0.01$ and the dimension is 2. Hence the number of atomic operations required to solve this problem is of the order $= 2^{\frac{k \log k}{\varepsilon}} dn \approx O(2^{500})$, which clearly seems to be hopeless to solve. Note that this problem is considerably harder than solving the Traveling Salesman Problem approximately for this point set.

Algorithm 2 works by exhaustively coloring the core set to k-colors and checking which coloring produces the best k-center clustering. It is incremental in nature and adds the furthest point from the current k-center clustering in each step (p in Line 4). If the addition of the point p is contained in the $1+\varepsilon$ expansion of the current clustering, the algorithm terminates. Otherwise, we explore the tree by adding p in each of the k balls. We prune the tree by maintaining a upper bound on the radius of the clustering and pruning out the

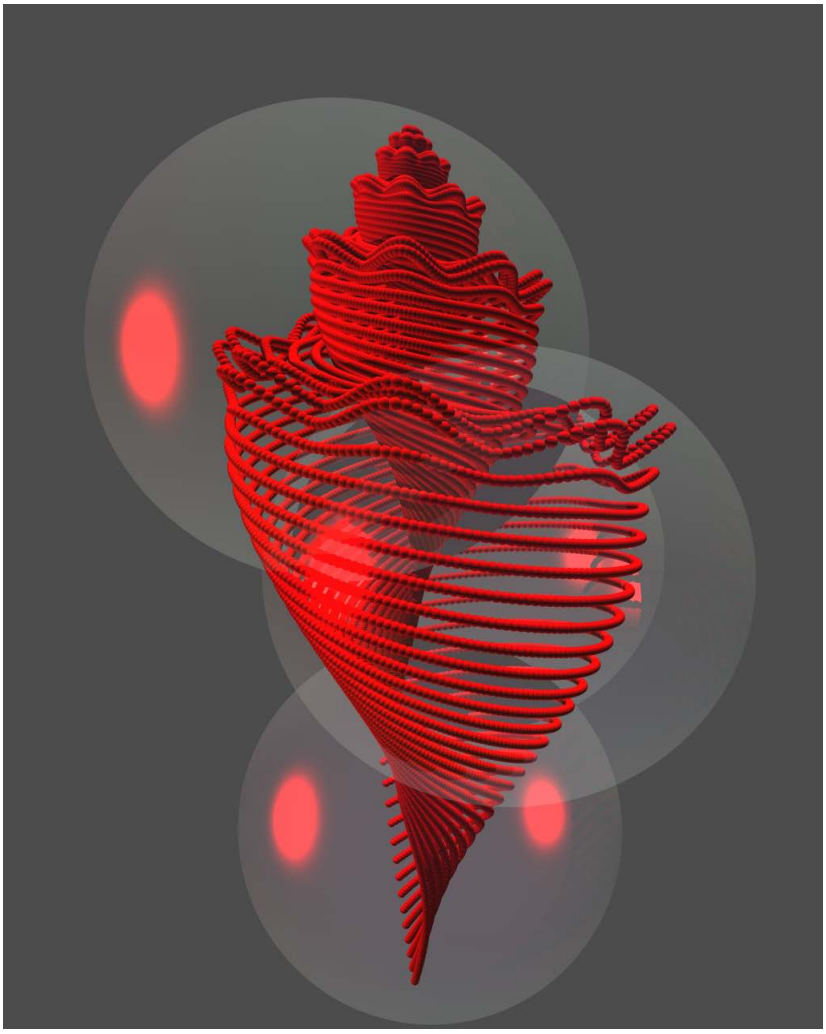
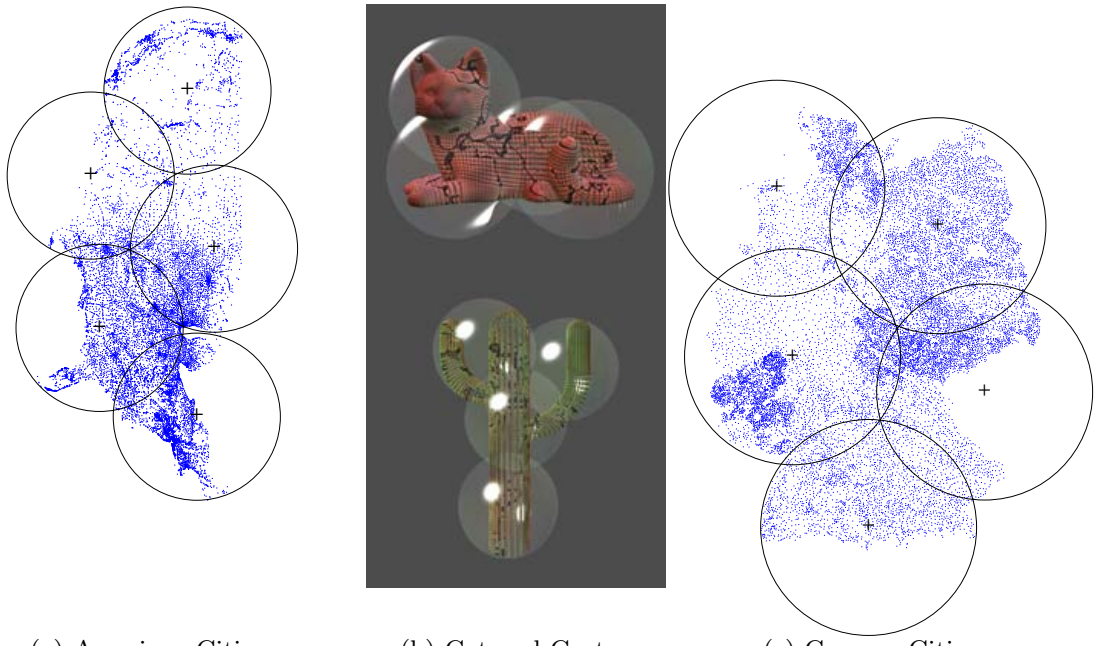


Figure 16: An example of 4-center clustering in 3D for $\varepsilon = 0.01$.

Algorithm 2 Outputs a $(1+\varepsilon)$-approximation of k-center of S and an $O(1/\varepsilon)$ -size core-set.

Require: Input: $S \in \mathbb{R}^d$, $\mathscr{M} = \{M_1, \ldots, M_k\}$, $\mathscr{B} = \{B_1, \ldots, B_k\}$, $\varepsilon > 0$

$\sigma \leftarrow$ Radius of 2-factor approximation for the problem instance.
$\mathscr{M}_o \leftarrow \mathscr{M}$, $\mathscr{B}_o \leftarrow \mathscr{B}$
loop
  $p \leftarrow$ point $q \in S$ furthest from $\mathscr{B}$.
  if $p \in (1+\varepsilon)\mathscr{B}$ **then**
    Return $\mathscr{M}_o, \mathscr{B}_o$
  else
    $\Pi \leftarrow$ Sort B_i using distance from p.
    for $j = 1$ to k **do**
      If $dist(p, \pi_i) > \sigma$ continue
      Put p in M_j corresponding to π_j
      Call k-center with $\mathscr{M}, \mathscr{B}$.
    end for
    $\sigma \leftarrow$ Min Radii of the above loop.
    Return the $\mathscr{M}, \mathscr{B}$ corresponding to σ.
  end if
end loop



(a) American Cities (b) Cat and Cactus (c) German Cities

Figure 17: Examples of k-center clustering in 2D and 3D for k = 5 and k = 4, $\varepsilon = 0.01$.

branches of the tree that need not be explored (line 10). The upper bound is updated as the algorithm progresses to better solutions (line 14).

To our knowledge, this is the first implementation of approximate *k*-center clustering that could solve such big instances of *k*-center clustering in practice. The running times for the algorithm in reality are still dismal and far from what we would like to have. For instance some of the data sets we solved in 3D (see for example figure 16), ran for almost a week on an Intel Itanium system (the system was running many other processes, and hence reporting timing is not meaningful). It seems that there is big difference in solving 2D problems from 3D problems for *k*-center clustering. Since the core set size grows in 3D, the running times are considerably higher for 3D problems than for 2D problems.

2.6 Future Work

There are many interesting theoretical and practical problems that are motivated by our investigations.

1. Several problems deserve experimental study. Are there practical methods for computing an MEB with outliers? Can one practically compute approximate solutions for 2-center and k-center problems in high dimensions? Are there coresets for support vector machines[7] ? How efficiently can one compute approximate minimum-volume enclosing ellipsoids? Are there core-sets for ellipsoids of size less than $\Theta(d^2)$ (we answer some of the questions on ellipsoids partially in the next chapter)? Does dimension reduction [120] help us to solve high-dimensional problems in practice? Can one use "warm start" strategies [197] to improve running times by giving a good starting point at every iteration? How does the improved algorithm, with running time $O\left(\frac{nd}{\varepsilon} + \frac{1}{\varepsilon^4}\log^2\frac{1}{\varepsilon}\right)$, suggested by Har-Peled [110] in remark 2 of section 2.3, compare with our implementations based on algorithm 1?

2. For which LP-type problems can one prove the existence of dimension-independent core-sets? What are the tightest core-set bounds one can prove for various distributions of points, with $d < 1/\varepsilon$?

[7] This question was partially answered in Kowalczyk [173] who showed that there are core sets of size $O\left(\left(\frac{D}{\varepsilon\rho}\right)^2 \ln\left(\frac{D}{\rho}\right)\right)$ for support vector machines. It is not very hard to improve the core set size to $O\left(\left(\frac{D}{\varepsilon\rho}\right)^2\right)$.

2.7 Software

The software for computing approximate minimum enclosing balls is available for download from

http://www.compgeom.com/meb/.

CHAPTER III

Minimum Volume Ellipsoids

"A circle no doubt has a certain appealing simplicity at first glance, but one look at a healthy ellipse should have convinced even the most mystical of astronomers that the perfect simplicity of the circle is akin to the vacant smile of complete idiocy. Compared to what an ellipse can tell us, a circle has nothing to say."

E. T. BELL

Computing the minimum volume enclosing ellipsoid (MVEE) of a given point set $\mathscr{S} = \{p^1, \ldots, p^n\} \subseteq \mathbb{R}^d$, denoted by MVEE($\mathscr{S}$), also known as the Löwner ellipsoid for $\mathscr{S}$ plays an important role in several diverse applications such as optimal design [171, 180], computational geometry [188, 58, 27], convex optimization [109], computer graphics [84, 39], pattern recognition [103], and statistics [170]. Variations of this problem such as MVEE with outliers [145] have many other applications [64, 121, 129].

In this chapter we study the problem of computing a $(1+\varepsilon)$-*approximation* to the minimum volume enclosing ellipsoid of a given point set $\mathscr{S} = \{p^1, p^2, \ldots, p^n\} \subseteq \mathbb{R}^d$. Based on a simple, initial volume approximation method, we propose a modification of Khachiyan's first-order algorithm. Our analysis leads to a slightly improved complexity

bound of $O(nd^3/\varepsilon)$ operations for $\varepsilon \in (0,1)$. As a byproduct, our algorithm returns a core set $\mathscr{X} \subseteq \mathscr{S}$ with the property that the minimum volume enclosing ellipsoid of $\mathscr{X}$ provides a good approximation to that of $\mathscr{S}$. Furthermore, the size of $\mathscr{X}$ depends only on the dimension d and ε, but not on the number of points n. In particular, our results imply that $|\mathscr{X}| = O(d^2/\varepsilon)$ for $\varepsilon \in (0,1)$.

A (full-dimensional) ellipsoid $\mathscr{E}_{Q,c}$ in $\mathbb{R}^d$ is specified by a $d \times d$ symmetric positive definite matrix Q and a center $c \in \mathbb{R}^d$ and is defined as

$$\mathscr{E}_{Q,c} = \{x \in \mathbb{R}^d : (x-c)^T Q(x-c) \leq 1\}. \tag{6}$$

The volume of an ellipsoid $\mathscr{E}_{Q,c}$, denoted by vol $\mathscr{E}_{Q,c}$, is given by vol $\mathscr{E}_{Q,c} = \eta$ det $Q^{-\frac{1}{2}}$, where η is the volume of the unit ball in $\mathbb{R}^d$ [109].

F. John [119] proved that MVEE($\mathscr{S}$) satisfies

$$\frac{1}{d}\text{MVEE}(\mathscr{S}) \subseteq \text{conv}(\mathscr{S}) \subseteq \text{MVEE}(\mathscr{S}), \tag{7}$$

where conv($\mathscr{S}$) denotes the convex hull of $\mathscr{S}$ and the ellipsoid on the left-hand side is obtained by scaling MVEE($\mathscr{S}$) around its center by a factor of $1/d$. Therefore, if $\mathscr{S}$ is viewed as the set of vertices of a full-dimensional polytope $\mathscr{P} \subseteq \mathbb{R}^d$, then MVEE($\mathscr{S}$) yields a rounded approximation of $\mathscr{P}$.

Given $\varepsilon > 0$, an ellipsoid $\mathscr{E}_{Q,c}$ is said to be a $(1+\varepsilon)$-approximation to MVEE($\mathscr{S}$) if

$$\mathscr{E}_{Q,c} \supseteq \mathscr{S}, \quad \text{vol } \mathscr{E}_{Q,c} \leq (1+\varepsilon) \text{ vol MVEE}(\mathscr{S}). \tag{8}$$

Several algorithms have been developed for the MVEE problem. These algorithms can be categorized as first-order algorithms [180, 181, 170, 124], second-order interior-point algorithms [148, 176], and a combination of the two[124]. For small dimensions d, the MVEE problem can be solved in $O(d^{O(d)}n)$ operations using randomized [142, 188, 1] or

deterministic [58] algorithms. A fast implementation is also available in the CGAL library[1] for solving the problem in two dimensions [99]. Khachiyan and Todd [125] established a linear-time reduction of the MVEE problem to the problem of computing a maximum volume inscribed ellipsoid (MVIE) in a polytope described by a finite number of inequalities. Therefore, the MVEE problem can also be solved using the algorithms developed for the MVIE problem [125, 147, 198, 20, 199]. Since the MVEE problem can be formulated as a maximum determinant problem, more general algorithms of Vandenberghe et. al. [186] and Toh [182] can be applied.

In contrast to the previous results, we mainly focus on the instances of the MVEE problem with $|\mathcal{S}| = n \gg d$, which is satisfied by several applications such as data mining and clustering. In particular, our goal is to compute a small subset $\mathcal{X} \subseteq \mathcal{S}$ such that $\mathcal{X}$ provides a good approximation of $\mathcal{S}$. For a point set $\mathcal{S}$, we say that $\mathcal{X} \subseteq \mathcal{S}$ is an ε-core set (or a core set) for $\mathcal{S}$ [47, 46, 132] if there exists an ellipsoid $\mathcal{E}_{Q,c} \subseteq \mathbb{R}^d$ such that $\mathcal{S} \subseteq \mathcal{E}_{Q,c}$ and $\mathcal{E}_{Q,c}$ is a $(1+\varepsilon)$-approximation to MVEE$(\mathcal{X})$. It follows from this definition that a core set $\mathcal{X}$ satisfies

$$\text{vol MVEE}(\mathcal{X}) \leq \text{vol MVEE}(\mathcal{S}) \leq \text{vol } \mathcal{E}_{Q,c} \leq (1+\varepsilon)\text{vol MVEE}(\mathcal{X}) \leq (1+\varepsilon)\text{vol MVEE}(\mathcal{S}),$$

which implies that $\mathcal{E}_{Q,c}$ is simultaneously a $(1+\varepsilon)$-approximation to MVEE$(\mathcal{X})$ and MVEE$(\mathcal{S})$.

The identification of small core sets is an important step towards solving larger problems. Recently, core sets have received significant attention and small core set results have been established for several geometric optimization problems such as the minimum enclosing ball problem [132] and related clustering problems [47, 46, 50, 2]. Small core set results

[1] http://www.cgal.org

form a basis for developing practical algorithms for large-scale problems since many geometric optimization problems can be solved efficiently for small input sets. In particular, the MVEE problem for an input set of m points in $\mathbb{R}^d$ can be solved in $O(m^{3.5}\log(m/\varepsilon))$ arithmetic operations [124], which is the best known complexity result if d is not fixed.

In this chapter, we propose a modification of Khachiyan's first-order algorithm [124], which computes a $(1+\varepsilon)$-approximation to MVEE($\mathscr{S}$) in

$$\Phi(n,d,\varepsilon) := O\left(nd^2\left([(1+\varepsilon)^{(2/d+1)}-1]^{-1}+\log d+\log\log n\right)\right) \tag{9}$$

operations. Based on a simple initial volume approximation algorithm, our modification yields a complexity bound of

$$\Xi(n,d,\varepsilon) := O\left(nd^2\left([(1+\varepsilon)^{(2/d+1)}-1]^{-1}+\log d\right)\right) \tag{10}$$

arithmetic operations, which reduces (9) by $O(nd^2\log\log n)$. In particular, our algorithm terminates in $O(nd^3/\varepsilon)$ operations for $\varepsilon\in(0,1]$. As a byproduct, we establish the existence of an ε-core set $\mathscr{X}\subseteq\mathscr{S}$ such that

$$|\mathscr{X}| = O\left(d[(1+\varepsilon)^{2/d+1}-1]^{-1}+d\log d\right), \tag{11}$$

independent of n, the number of points in $\mathscr{S}$. In particular, $|\mathscr{X}| = O(d^2/\varepsilon)$ for $\varepsilon\in(0,1]$.

We remark that any ellipsoid in $\mathbb{R}^d$ is determined by at most $d(d+1)/2$ points, which implies that, in theory, there always exists a core set of size $O(d^2)$ for any $\varepsilon\geq 0$. In comparison, our algorithm can efficiently compute a core set $\mathscr{X}$ satisfying (11).

The chapter is organized as follows. We define notation in the next section. In Section 3.2, we review formulations of the MVEE problem as an optimization problem. Section 3.3 is devoted to a deterministic volume approximation algorithm that will be the basis

for our algorithm. In Section 3.4, we review Khachiyan's first-order algorithm and its analysis and propose a new interpretation. We present our modification and establish a slightly improved complexity bound. As a byproduct, our algorithm returns a core set whose size is independent of n. Section 3.5 concludes the chapter with future research directions.

3.1 Notation

Vectors will be denoted by lower-case Roman letters. For a vector u, u_i denotes the ith component. Inequalities on vectors will apply to each component. e will be reserved for the vector of ones in the appropriate dimension, which will be clear from the context. e_j is the jth unit vector. For a vector u, U will denote the diagonal matrix whose entries are given by components of u. Upper-case Roman letters will be reserved for matrices. The identity matrix will be denoted by I. $\mathrm{trace}(U)$ will denote the sum of the diagonal entries of U. For a finite set of vectors $\mathcal{V}$, $\mathrm{span}(\mathcal{V})$ denotes the linear subspace spanned by $\mathcal{V}$. Functions and operators will be denoted by upper-case Greek letters. Scalars except for n and d will be represented by lower-case Greek letters unless they represent components of a vector or a sequence of scalars, vectors or matrices. i, j, and k will be reserved for indexing purposes. Upper-case script letters will be used for all other objects such as sets, polytopes, and ellipsoids.

3.2 Formulations

In this section, we discuss formulations of the MVEE problem as an optimization problem. Throughout the rest of this chapter, we make the following assumption, which guarantees that the minimum volume enclosing ellipsoid is full-dimensional.

Assumption 3.2.1 *The affine hull of $p^1, \ldots, p^n$ is $\mathbb{R}^d$.*

The MVEE problem can be formulated as an optimization problem in several different ways (see, e.g., [176]). We consider two formulations in this section.

Given a set $\mathcal{S} \subseteq \mathbb{R}^d$ of n points $p^1, \ldots, p^n$, we define a "lifting" of $\mathcal{S}$ to $\mathbb{R}^{d+1}$ via

$$\mathcal{S}' := \{\pm q^1, \ldots, \pm q^n\}, \quad \text{where} \quad q^i := \begin{bmatrix} p^i \\ 1 \end{bmatrix}, \quad i = 1, \ldots, n. \tag{12}$$

It follows from the results of [125, 148] that

$$\text{MVEE}(\mathcal{S}) = \text{MVEE}(\mathcal{S}') \cap \mathcal{H}, \tag{13}$$

where

$$\mathcal{H} := \{x \in \mathbb{R}^{d+1} : x_{d+1} = 1\}. \tag{14}$$

Since $\mathcal{S}'$ is centrally symmetric, $\text{MVEE}(\mathcal{S}')$ is centered at the origin. This observation gives rise to the following convex optimization problem to compute $\text{MVEE}(\mathcal{S}')$, whose solution can be used to compute $\text{MVEE}(\mathcal{S})$ via (13):

$$\begin{array}{lll} (\mathbf{P}(\mathcal{S})) \quad \min_M & -\log\det M & \\ \text{s.t.} & (q^i)^T M\, q^i \leq 1, & i = 1, \ldots, n, \\ & M \in \mathbb{R}^{(d+1)\times(d+1)} & \text{is symmetric and positive definite,} \end{array}$$

where $M \in \mathbb{R}^{(d+1)\times(d+1)}$ is the decision variable. A positive definite matrix $M^* \in \mathbb{R}^{(d+1)\times(d+1)}$ is optimal for $(\mathbf{P}(\mathcal{S}))$ along with Lagrange multipliers $z^* \in \mathbb{R}^n$ if and only if

$$-(M^*)^{-1} + \Pi(z^*) = 0, \tag{15a}$$

$$z_i^* \left(1 - (q^i)^T M^*\, q^i\right) = 0, \quad i = 1, \ldots, n, \tag{15b}$$

$$(q^i)^T M^*\, q^i \leq 1, \quad i = 1, \ldots, n, \tag{15c}$$

$$z^* \geq 0, \tag{15d}$$

where $\Pi : \mathbb{R}^n \to \mathbb{R}^{(d+1)\times(d+1)}$ is a linear operator given by

$$\Pi(z) := \sum_{i=1}^{n} z_i q^i (q^i)^T. \tag{16}$$

The Lagrangian dual of $(\mathbf{P}(\mathscr{S}))$ is equivalent to

$$\begin{aligned} (\mathbf{D}(\mathscr{S})) \quad \max_u \quad & \log\det\Pi(u) \\ \text{s.t.} \quad & e^T u = 1, \\ & u \geq 0, \end{aligned}$$

where $u \in \mathbb{R}^n$ is the decision variable. Since $(\mathbf{D}(\mathscr{S}))$ is a concave maximization problem, $u^* \in \mathbb{R}^n$ is an optimal solution (along with dual solutions $s^* \in \mathbb{R}^n$ and $\lambda^* \in \mathbb{R}$) if and only if the following optimality conditions are satisfied:

$$(q^i)^T \Pi(u^*)^{-1} q^i + s_i^* = \lambda^*, \quad i = 1,\ldots,n, \tag{18a}$$

$$e^T u^* = 1, \tag{18b}$$

$$u_i^* s_i^* = 0, \quad i = 1,\ldots,n, \tag{18c}$$

together with $u^* \geq 0$ and $s^* \geq 0$. Multiplying both sides of (18a) by u_i^* and summing up for $i = 1,\ldots,n$ yields

$$\sum_{i=1}^{n} u_i^* (q^i)^T \Pi(u^*)^{-1} q^i = \operatorname{trace}\left(\Pi(u^*)^{-1}\left[\sum_{i=1}^{n} u_i^* q^i (q^i)^T\right]\right) = \operatorname{trace}(I) = d+1,$$

which implies $\lambda^* = d+1$ by (18b) and (18c). Consequently,

$$M^* := \frac{1}{d+1}\Pi(u^*)^{-1} \tag{19}$$

is a feasible solution for $(\mathbf{P}(\mathscr{S}))$ and satisfies the optimality conditions (15) for $(\mathbf{P}(\mathscr{S}))$ together with $z^* := (d+1)u^*$.

It follows from (13) and (19) that an optimal solution u^* for $(\mathbf{D}(\mathscr{S}))$ can be used to compute MVEE$(\mathscr{S})$ as follows:

$$\text{MVEE}(\mathscr{S}) = \{x \in \mathbb{R}^d : \left(\frac{1}{d+1}\right) [x^T \quad 1]\Pi(u^*)^{-1} \begin{bmatrix} x \\ 1 \end{bmatrix} \leq 1\}. \tag{20}$$

Let $P \in \mathbb{R}^{d\times n}$ be the matrix whose ith column is given by p^i. By (16), we have

$$\Pi(u^*) = \begin{bmatrix} PU^*P^T & Pu^* \\ (Pu^*)^T & 1 \end{bmatrix} = \begin{bmatrix} I & Pu^* \\ 0 & 1 \end{bmatrix} \begin{bmatrix} PU^*P^T - Pu^*(Pu^*)^T & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I & 0 \\ (Pu^*)^T & 1 \end{bmatrix}. \tag{21}$$

Inverting both sides in (21) yields

$$\Pi(u^*)^{-1} = \begin{bmatrix} I & 0 \\ -(Pu^*)^T & 1 \end{bmatrix} \begin{bmatrix} \left(PU^*P^T - Pu^*(Pu^*)^T\right)^{-1} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I & -Pu^* \\ 0 & 1 \end{bmatrix}, \tag{22}$$

Substituting (22) in (20), we obtain

$$\text{MVEE}(\mathscr{S}) = \mathscr{E}_{Q^*,c^*} := \{x \in \mathbb{R}^d : (x-c^*)^T Q^*(x-c^*) \leq 1\}, \tag{23}$$

where

$$Q^* := \frac{1}{d}\left(PU^*P^T - Pu^*(Pu^*)^T\right)^{-1}, \quad c^* := Pu^*. \tag{24}$$

This establishes the following result.

Lemma 3.2.1 *Let $u^* \in \mathbb{R}^n$ be an optimal solution of $(\mathbf{D}(\mathscr{S}))$ and let $P \in \mathbb{R}^{d\times n}$ be the matrix whose ith column is given by p^i. Then, MVEE$(\mathscr{S}) = \mathscr{E}_{Q^*,c^*}$, where $Q^* \in \mathbb{R}^{d\times d}$ and $c^* \in \mathbb{R}^d$ are given by (24). Furthermore,*

$$\log vol\ MVEE(\mathscr{S}) = \log \eta + \frac{d}{2}\log d + \frac{1}{2}\log\det\Pi(u^*), \tag{25}$$

where η is the volume of the unit ball in $\mathbb{R}^d$.

Proof. We only need to prove (25). Note that vol MVEE$(\mathcal{S}) = \eta \det(Q^*)^{-\frac{1}{2}}$, where Q^* is defined as in (24). Therefore,

$$\log \text{vol MVEE}(\mathcal{S}) = \log \eta + \frac{d}{2}\log d + \frac{1}{2}\log\det\left(PU^*P^T - Pu^*(Pu^*)^T\right). \tag{26}$$

By (21), $\log\det\Pi(u^*) = \log\det(PU^*P^T - Pu^*(Pu^*)^T)$, establishing (25). □

3.3 Initial Volume Approximation

Given $\mathcal{S} = \{p^1, \ldots, p^n\} \subseteq \mathbb{R}^d$, we present a simple deterministic algorithm that identifies a subset $\mathcal{X}_0 \subseteq \mathcal{S}$ of size at most $2d$ such that vol MVEE$(\mathcal{X}_0)$ is a provable approximation to vol MVEE$(\mathcal{S})$.

Algorithm 3 Volume approximation algorithm

Require: Input set of points $\mathcal{S} = \{p^1, \ldots, p^n\} \subseteq \mathbb{R}^d$

If $n \leq 2d$, then $\mathcal{X}_0 \leftarrow \mathcal{S}$. Return.
$\Psi \leftarrow \{0\}$, $\mathcal{X}_0 \leftarrow \emptyset$, $i \leftarrow 0$.
While $\mathbb{R}^d \setminus \Psi \neq \emptyset$ do
loop
  $i \leftarrow i+1$. Pick an arbitrary direction $b^i \in \mathbb{R}^d$ in the orthogonal complement of Ψ.
  $\alpha \leftarrow \arg\max_{k=1,\ldots,n}(b^i)^T p^k$, $\mathcal{X}_0 \leftarrow \mathcal{X}_0 \cup \{p^\alpha\}$.
  $\beta \leftarrow \arg\min_{k=1,\ldots,n}(b^i)^T p^k$, $\mathcal{X}_0 \leftarrow \mathcal{X}_0 \cup \{p^\beta\}$.
  $\Psi \leftarrow \text{span}(\Psi, \{p^\beta - p^\alpha\})$.
end loop

Lemma 3.3.1 *Algorithm 3 terminates in $O(nd^2)$ time with a subset $\mathcal{X}_0 \subseteq \mathcal{S}$ with $|\mathcal{X}_0| \leq 2d$ such that*

$$\textit{vol MVEE}(\mathcal{S}) \leq d^{2d}\,\textit{vol MVEE}(\mathcal{X}_0). \tag{27}$$

Proof. If $n \leq 2d$, then the result trivially holds. For $n > 2d$, the proof is based on the results of [35, 139]. At step k of Algorithm 3, Ψ is given by the span of k linearly independent vectors by Assumption 3.2.1. Hence, upon termination, $\Psi = \mathbb{R}^d$. It follows that $|\mathscr{X}_0| = 2d$. Note that each step requires $O(nd)$ operations, giving an overall running time of $O(nd^2)$ at the end of d steps. It follows from the results of [35] that vol conv$(\mathscr{S}) \leq d!$ vol conv$(\mathscr{X}_0)$. Combining this inequality with (7), we have

$$\frac{1}{d^d} \text{ vol MVEE}(\mathscr{S}) \leq \text{vol conv}(\mathscr{S}) \leq d! \text{ vol conv}(\mathscr{X}_0) \leq d! \text{ vol MVEE}(\mathscr{X}_0),$$

which implies that vol MVEE$(\mathscr{S}) \leq d!d^d$vol MVEE$(\mathscr{X}_0) \leq d^{2d}$ MVEE$(\mathscr{X}_0)$.

□

3.4 A First-Order Algorithm

In this section, we present a modification of Khachiyan's first-order algorithm for approximating the minimum volume enclosing ellipsoid of a given set $\mathscr{S} = \{p^1, \ldots, p^n\} \subseteq \mathbb{R}^d$. Our modification leads to a slightly improved complexity bound. As a byproduct, our analysis establishes the existence of a core set $\mathscr{X} \subseteq \mathscr{S}$ whose size depends only on d and ε, but not on n, the number of points.

3.4.1 Khachiyan's Algorithm Revisited

Khachiyan's first-order algorithm [124] can be interpreted in several different ways (e.g. barycentric coordinate descent [124], conditional gradient method [176]). In this chapter, we present our interpretation of this algorithm.

For a set of points $\mathscr{S} \subseteq \mathbb{R}^d$, consider the nonlinear optimization problem $(\mathbf{D}(\mathscr{S}))$.

Given a feasible solution $u^i \in \mathbb{R}^n$, consider the following linearization of $(\mathbf{D}(\mathscr{S}))$ at u^i:

$$(\mathbf{LP}_i) \quad \max_{v \in \mathbb{R}^n} \sum_{k=1}^{n} v_k \, (q^k)^T \Pi(u^i)^{-1} q^k, \quad \text{s.t.} \quad e^T v = 1, \quad v \geq 0.$$

Since the feasible region of $(\mathbf{LP}_i)$ is the unit simplex in $\mathbb{R}^n$, the optimal solution v^* is the unit vector e_j, where

$$j := \arg \max_{k=1,\ldots,n} (q^k)^T \Pi(u^i)^{-1} q^k. \tag{28}$$

Let

$$\kappa^i := \max_{k=1,\ldots,n} (q^k)^T \Pi(u^i)^{-1} q^k. \tag{29}$$

The next iterate u^{i+1} is given by a convex combination of u^i and e_j, i.e., $u^{i+1} := (1 - \beta^i)u^i + \beta^i e_j$, where β^i is the maximizer of the following one-dimensional optimization problem [124]:

$$\beta^i := \arg \max_{\beta \in [0,1]} \log\det \Pi((1-\beta)u^i + \beta e_j) = \frac{\kappa^i - (d+1)}{(d+1)(\kappa^i - 1)}. \tag{30}$$

The algorithm continues in an iterative manner starting with u^{i+1}. Consequently, Khachiyan's first-order method can be viewed as a sequential linear programming algorithm for the non-linear optimization problem $\mathbf{D}(\mathscr{S})$.

Upon termination, the algorithm returns an ellipsoid $\mathscr{E}_{Q,c}$ with the property that

$$\mathscr{S} \subseteq \mathscr{E}_{Q,c}, \quad \text{vol}\, \mathscr{E}_{Q,c} \leq (1+\varepsilon)\, \text{vol MVEE}(\mathscr{S}), \tag{31}$$

where $\varepsilon > 0$.

Below, we outline Khachiyan's algorithm.

3.4.2 Analysis of Khachiyan's Algorithm

Khachiyan proved the following complexity result.

Algorithm 4 Khachiyan's first-order algorithm

Require: Input set of points $\mathscr{S} = \{p^1, \ldots, p^n\} \subseteq \mathbb{R}^d, \varepsilon > 0$

$i \leftarrow 0$, $u^0 \leftarrow (1/n)e$

While not converged

loop

  $j \leftarrow \arg\max_{k=1,\ldots,n} (q^k)^T \Pi(u^i)^{-1} q^k$, $\kappa \leftarrow \max_{k=1,\ldots,n} (q^k)^T \Pi(u^i)^{-1} q^k$.

  $\beta \leftarrow \frac{\kappa-(d+1)}{(d+1)(\kappa-1)}$

  $u^{i+1} \leftarrow (1-\beta)u^i + \beta e_j$, $i \leftarrow i+1$.

end loop

Theorem 3.4.1 (Khachiyan (1996)) *Let $\varepsilon > 0$. Algorithm 4 returns an ellipsoid $\mathscr{E}_{Q,c}$ that satisfies the conditions in (31) in $\Phi(n,d,\varepsilon)$ operations, where Φ is defined by (9). In particular, if $\varepsilon \in (0,1)$, Algorithm 4 terminates after $O(nd^2(d/\varepsilon + \log\log n))$ operations.*

In this section, we present our interpretation of the analysis of Algorithm 4. Let $\mathscr{S}' \subseteq \mathbb{R}^{d+1}$ denote the lifting of the point set $\mathscr{S} \subseteq \mathbb{R}^d$ given by (12). It follows from (13) that MVEE($\mathscr{S}$) can be recovered from MVEE($\mathscr{S}'$). Furthermore, if the ellipsoid $\tilde{\mathscr{E}} \subseteq \mathbb{R}^{d+1}$ is a $(1+\varepsilon)$-approximation of MVEE($\mathscr{S}'$), then $\mathscr{E} := \tilde{\mathscr{E}} \cap \mathscr{H} \subseteq \mathbb{R}^d$ is a $(1+\varepsilon)$-approximation of MVEE($\mathscr{S}$), where $\mathscr{H}$ is given by (14) [125, 148]. Therefore, we analyze Algorithm 4 for $\mathscr{S}' \subseteq \mathbb{R}^{d+1}$.

At iteration i, we define a "trial" ellipsoid $\tilde{\mathscr{E}}^i := \tilde{\mathscr{E}}_{M^i,0} \subseteq \mathbb{R}^{d+1}$ (cf. (19)), where

$$M^i := \frac{1}{d+1}\Pi(u^i)^{-1}, \quad i = 0,1,2,\ldots. \tag{32}$$

Note that one can define a corresponding ellipsoid $\mathscr{E}^i := \mathscr{E}_{Q^i,c^i} \subseteq \mathbb{R}^d$ via (24):

$$Q^i := \frac{1}{d}\left(PU^iP^T - (Pu^i)(Pu^i)^T\right)^{-1}, \quad c^i := Pu^i, \quad i = 0,1,2,\ldots. \tag{33}$$

Furthermore, by (32), $\operatorname{vol} \tilde{\mathscr{E}}^i = \eta' \det(M^i)^{-1/2} = \eta'(d+1)^{(d+1)/2} \det\Pi(u^i)^{1/2}$, where η' is the volume of the unit ball in $\mathbb{R}^{d+1}$. Therefore,

$$\log\operatorname{vol} \tilde{\mathscr{E}}^i = \log\eta' + \frac{d+1}{2}\log(d+1) + \frac{1}{2}\log\det\Pi(u^i), \quad i = 0,1,2,\ldots. \tag{34}$$

Similarly to (25), we have

$$\log \text{vol}\, \mathscr{E}^i = \log \eta + \frac{d}{2} \log d + \frac{1}{2} \log \det \Pi(u^i), \quad i = 0, 1, 2, \ldots, \tag{35}$$

which, together with (34), implies that $\log \text{vol}\, \tilde{\mathscr{E}}^i$ and $\log \text{vol}\, \mathscr{E}^i$ differ by a constant that depends only on d.

Let u^* denote the optimal solution of $(\mathbf{D}(\mathscr{S}))$. Since u^i is a feasible solution of $(\mathbf{D}(\mathscr{S}))$, it follows from (34) and (19) that

$$\text{vol}\, \tilde{\mathscr{E}}^i \leq \text{vol MVEE}(\mathscr{S}'), \quad i = 0, 1, 2, \ldots. \tag{36}$$

We define

$$\varepsilon_i := \min\{\upsilon \geq 0 : (q^k)^T M^i (q^k) \leq 1 + \upsilon,\ k = 1, \ldots, n\}, \quad i = 0, 1, 2, \ldots \tag{37}$$

so that $\mathscr{S}' \subseteq \sqrt{1+\varepsilon_i}\ \tilde{\mathscr{E}}^i$. ε_i can be viewed as a quality measure of iterate i. Combining (36) and (37), we obtain

$$\text{vol}\, \tilde{\mathscr{E}}^i \leq \text{vol MVEE}(\mathscr{S}') \leq (1+\varepsilon_i)^{(d+1)/2} \text{vol}\, \tilde{\mathscr{E}}^i, \quad i = 0, 1, 2, \ldots. \tag{38}$$

Taking logarithms in (38), it follows from (34) that

$$v_i \leq v^* \leq (d+1) \log(1+\varepsilon_i) + v_i, \quad i = 0, 1, 2, \ldots, \tag{39}$$

where v_i denotes the objective function value corresponding to the feasible solution u^i of $(\mathbf{D}(\mathscr{S}))$, i.e.,

$$v_i := \log \det \Pi(u^i), \quad i = 0, 1, 2, \ldots, \tag{40}$$

and v^* denotes the optimal value of $(\mathbf{D}(\mathscr{S}))$.

By (37) and (32),

$$\kappa^i := \max_{k=1,\ldots,n} (q^k)^T \Pi(u^i)^{-1} q^k = (d+1)(1+\varepsilon_i), \quad i = 0, 1, 2, \ldots \tag{41}$$

so that

$$\beta^i := \frac{\kappa^i - (d+1)}{(d+1)(\kappa^i - 1)} = \frac{\varepsilon_i}{\kappa^i - 1}, \quad i = 0, 1, 2, \ldots. \tag{42}$$

The next iterate u^{i+1} is defined as

$$u^{i+1} := (1 - \beta^i)u^i + \beta^i e_j, \tag{43}$$

where j and β^i are given by (28) and (30), respectively. Since $\Pi(u)$ is linear, we obtain

$$\Pi(u^{i+1}) = (1-\beta^i)\Pi(u^i) + \beta^i \Pi(e_j) = \Pi(u^i)\left[(1-\beta^i)I + \beta^i \Pi(u^i)^{-1}\Pi(e_j)\right]. \tag{44}$$

Arguing similarly to Lemma 3 of [124], we have

$$\begin{aligned}
\log\det\Pi(u^{i+1}) &= \log\det\Pi(u^i) + d\log(1-\beta^i) + \log(1+\varepsilon_i), \\
&= \log\det\Pi(u^i) - d\log\left(1 + \frac{\varepsilon_i}{d(1+\varepsilon_i)}\right) + \log(1+\varepsilon_i), \\
&\geq \log\det\Pi(u^i) - \frac{\varepsilon_i}{1+\varepsilon_i} + \log(1+\varepsilon_i), \quad i = 0, 1, 2, \ldots, \\
&\geq \log\det\Pi(u^i) + \begin{cases} \log 2 - \frac{1}{2} > 0 & \text{if } \varepsilon_i \geq 1, \\ \frac{1}{8}\varepsilon_i^2 & \text{if } \varepsilon_i < 1. \end{cases}
\end{aligned} \tag{45}$$

which implies that the objective function value strictly increases at each iteration.

Note that

$$\kappa^0 = \max_{k=1,\ldots,n} (q^k)^T \Pi(u^0)^{-1} q^k \leq \sum_{k=1}^{n} (q^k)^T \Pi(u^0)^{-1} q^k = n \operatorname{trace}(\Pi(e)^{-1}\Pi(e)) = n(d+1).$$

By (41), it follows that

$$\varepsilon_0 \leq n - 1. \tag{46}$$

The following inequalities follow from (39), (45), and (46):

$$\begin{aligned} v_0 &> -\infty, && \text{(47a)} \\ \delta_i := v^* - v_i &\leq (d+1)\log(1+\varepsilon_i), \quad i = 0,1,\ldots, && \text{(47b)} \\ \pi_i := v_{i+1} - v_i &\geq \log(1+\varepsilon_i) - \frac{\varepsilon_i}{1+\varepsilon_i}, \quad i = 0,1,\ldots, && \text{(47c)} \\ \delta_0 = v^* - v_0 &\leq (d+1)\log n. && \text{(47d)} \end{aligned}$$

Khachiyan's analysis of Algorithm 4 (see Lemma 4 in [124]) consists of two stages. In the first stage, an upper bound is derived on the smallest index k such that $\varepsilon_k \leq 1$. Using (47b) and (47c), Khachiyan establishes that

$$k = O(d \log \delta_0), \tag{48}$$

which implies that $k = O(d(\log d + \log\log n))$ by (47d).

The second stage of Khachiyan's analysis consists of bounding the number of iterations to halve ε_i assuming $\varepsilon_i \leq 1$. Khachiyan shows that it takes $O(d/\mu)$ iterations to obtain $\varepsilon_i \leq \mu$ for any $\mu \in (0,1)$. It follows from (38) that Algorithm 4 needs to run until

$$\varepsilon_i \leq (1+\varepsilon)^{2/d+1} - 1 \tag{49}$$

in order to obtain a $(1+\varepsilon)$-approximation to MVEE($\mathscr{S}$).

Combining the two parts together with the fact that each iteration can be performed in $O(nd)$ operations via updating $\Pi(u^i)^{-1}$ using (44) yields the complexity result of Theorem 3.4.1. If $\varepsilon \in (0,1)$, then $(1+\varepsilon)^{2/(d+1)} - 1 = O(\varepsilon/d)$, proving the second part of Theorem 3.4.1.

3.4.3 A Different Interpretation of Khachiyan's Algorithm

Our presentation of the analysis of Khachiyan's algorithm gives rise to another interpretation. Consider the trial ellipsoid $\mathscr{E}^i \subseteq \mathbb{R}^d$ corresponding to u^i defined by (33). By (22),

$$(q^k)^T \Pi (u^i)^{-1} q^k = d(p^k - c^i)^T Q^i (p^k - c^i) + 1, \quad k = 1, \ldots, n. \tag{50}$$

At each iteration, it follows from (41) that the algorithm computes the farthest point p^j from the current trial ellipsoid $\mathscr{E}^i$ using its ellipsoidal norm. By (43) and (33), the center c^{i+1} of the next trial ellipsoid $\mathscr{E}^{i+1} := \mathscr{E}_{Q^{i+1}, c^{i+1}}$ is shifted towards p^j, i.e.,

$$c^{i+1} = (1 - \beta^i) c^i + \beta^i p^j. \tag{51}$$

Similarly, by (33),

$$(Q^i)^{-1} = d(PU^iP^T - (Pu^i)(Pu^i)^T) = d \sum_{k=1}^{n} u_k (p^k - c^i)(p^k - c^i)^T,$$

where we used $\sum_{k=1}^{n} u_k = 1$ and $Pu^i = c^i$. By (43), it follows that

$$(Q^{i+1})^{-1} = (1 - \beta^i)(Q^i)^{-1} + d\beta (p^j - c^i)(p^j - c^i)^T, \tag{52}$$

which implies that the next trial ellipsoid is obtained by "expanding" the current trial ellipsoid towards p^j. In particular, if $p^j - c^i$ coincides with one of the axes of $\mathscr{E}^i$ (i.e., one of the eigenvectors of Q^i), then $\mathscr{E}^{i+1}$ is simply obtained by expanding $\mathscr{E}^i$ along that axis and shrinking it along the remaining axes.

Therefore, Khachiyan's algorithm implicitly generates a sequence of ellipsoids with the property that the next ellipsoid in the sequence is given by shifting and expanding the current ellipsoid towards the farthest outlier.

3.4.4 A Modification

In this subsection, we present a modification of Khachiyan's first-order algorithm described in Section 3.4.1. Our algorithm leads to a slightly improved complexity bound.

The following theorem gives a complexity bound for Algorithm 5.

Algorithm 5 Outputs a $(1+\varepsilon)$-approximation of MVEE($\mathscr{S}$)

Require: Input set of points $\mathscr{S} = \{p^1, \ldots, p^n\} \subseteq \mathbb{R}^d, \varepsilon \in (0,1)$

1: Run Algorithm 3 on $\mathscr{S}$ to get output $\mathscr{X}_0$.
2: Let $u^0 \in \mathbb{R}^n$ be such that $u^0_j = 1/|\mathscr{X}_0|$ for $p^j \in \mathscr{X}_0$ and $u^0_j = 0$ otherwise.
3: Run Algorithm 4 on $\mathscr{S}$ starting with u^0.

Theorem 3.4.2 *Let $\varepsilon > 0$. Algorithm 5 returns a $(1+\varepsilon)$-approximation of MVEE($\mathscr{S}$) in $\Xi(n,d,\varepsilon)$ operations, where Ξ is defined by (10). In particular, if $\varepsilon \in (0,1)$, Algorithm 5 terminates after $O(nd^3/\varepsilon)$ operations.*

Proof. By Lemma 3.3.1, Algorithm 3 returns $\mathscr{X}_0 \subseteq \mathscr{S}$ of size at most $2d$ in $O(nd^2)$ operations. Let $u_{\mathscr{X}_0} \in \mathbb{R}^{|\mathscr{X}_0|}$ denote the restriction of u^0 to its positive components. Since $u_{\mathscr{X}_0}$ coincides with the initial iterate in Algorithm 3 applied to $(\mathbf{D}(\mathscr{X}_\mathbf{0}))$, it follows from (47d) that

$$\log\det\Pi(u_*) - \log\det\Pi(u^0) = O(d\log d), \tag{53}$$

where $u_* \in \mathbb{R}^n$ denotes the vector whose restriction to its components in $\mathscr{X}_0$ yields the optimal solution of $(\mathbf{D}(\mathscr{X}_\mathbf{0}))$.

By Lemma 3.3.1, vol MVEE($\mathscr{S}$) $\leq d^{2d}$vol MVEE($\mathscr{X}_0$). Taking logarithms on both sides, it follows from (35) that

$$\log\det\Pi(u^*) - \log\det\Pi(u_*) = O(d\log d), \tag{54}$$

where u^* denotes the optimal solution of $(\mathbf{D}(\mathscr{S}))$. By (53) and (54), we obtain

$$\log\det\Pi(u^*) - \log\det\Pi(u^0) = O(d\log d).$$

Since u^0 is used as an initial iterate in Algorithm 4, it follows from (47d) and (48) that Algorithm 5 requires $O(d\log d)$ iterations to obtain an iterate u^i with $\varepsilon_i \leq 1$. Combining this

result with the second part of the analysis of Algorithm 4, we obtain the desired complexity result. □

3.4.5 A Core Set Result

In this subsection, we establish that, upon termination, our algorithm produces a core set $\mathscr{X} \subseteq \mathscr{S}$ whose size depends only on d and ε, but not on n.

Theorem 3.4.3 *Let $\varepsilon > 0$. For a given set $\mathscr{S} = \{p^1, \ldots, p^n\} \subseteq \mathbb{R}^d$ of n points, let u^f denote the final iterate returned by Algorithm 5 applied to $\mathscr{S}$. Let $\mathscr{X} := \{p^k \in \mathscr{S} : u_k^f > 0,\ k = 1, \ldots, n\}$. Then, $\mathscr{X}$ is an ε-core set of $\mathscr{S}$. Furthermore,*

$$|\mathscr{X}| = O(d[(1+\varepsilon)^{2/d+1} - 1]^{-1} + d \log d). \tag{55}$$

In particular, if $\varepsilon \in (0,1)$, then $|\mathscr{X}| = O(d^2/\varepsilon)$.

Proof. We first prove (55). Note that u^0 in Algorithm 5 has at most $2d$ positive components and at each iteration, at most one component becomes positive. Therefore,

$$|\mathscr{X}| \leq 2d + O\left(d \log d + d[(1+\varepsilon)^{2/d+1} - 1]^{-1}\right) = O\left(d[(1+\varepsilon)^{2/d+1} - 1]^{-1} + d \log d\right).$$

Let $\tilde{\mathscr{E}}^f \subseteq \mathbb{R}^{d+1}$ denote the trial ellipsoid corresponding to u^f defined via (32) and let $\tilde{\mathscr{E}} := (1+\varepsilon)^{1/d+1} \tilde{\mathscr{E}}^f$. By (49),

$$\operatorname{conv}(\mathscr{X}') \subseteq \operatorname{conv}(\mathscr{S}') \subseteq \tilde{\mathscr{E}}, \tag{56}$$

where $\mathscr{X}'$ and $\mathscr{S}'$ denote the lifting of $\mathscr{X}$ and $\mathscr{S}$ to $\mathbb{R}^{d+1}$, respectively. By (38), we have

$$\operatorname{vol} \tilde{\mathscr{E}}^f \leq \operatorname{vol} \operatorname{MVEE}(\mathscr{S}') \leq (1+\varepsilon) \operatorname{vol} \tilde{\mathscr{E}}^f = \operatorname{vol} \tilde{\mathscr{E}}, \tag{57}$$

which implies that $\operatorname{vol} \tilde{\mathscr{E}} \leq (1+\varepsilon) \operatorname{vol} \operatorname{MVEE}(\mathscr{S}')$.

Let $u^f_{\mathscr{X}} \in \mathbb{R}^{|\mathscr{X}|}$ be the restriction of u^f to its positive components. Note that $u^f_{\mathscr{X}}$ is a feasible solution of $(\mathbf{D}(\mathscr{X}))$. Furthermore, the trial ellipsoid corresponding to $u^f_{\mathscr{X}}$ coincides with $\tilde{\mathscr{E}}^f$ by (32). Therefore, arguing similarly to (36), we obtain

$$\frac{1}{1+\varepsilon}\text{vol}\,\tilde{\mathscr{E}} = \text{vol}\,\tilde{\mathscr{E}}^f \leq \text{vol MVEE}(\mathscr{X}'), \tag{58}$$

which, together with (57), implies that

$$\text{vol MVEE}(\mathscr{X}') \leq \text{vol MVEE}(\mathscr{S}') \leq \text{vol}\,\tilde{\mathscr{E}} \leq (1+\varepsilon)\text{vol MVEE}(\mathscr{X}') \leq (1+\varepsilon)\text{vol MVEE}(\mathscr{S}').$$

Since lifting preserves the approximation factor, it follows from (56), (57), and (58) that the ellipsoid $\mathscr{E} := \tilde{\mathscr{E}} \cap \mathscr{H}$, where $\mathscr{H}$ is defined by (14), is simultaneously a $(1+\varepsilon)$-approximation to MVEE$(\mathscr{X})$ and to MVEE$(\mathscr{S})$. Therefore, $\mathscr{X}$ is an ε-core set of $\mathscr{S}$.

□

Remark 1: The size of the core set $\mathscr{X}$ in Theorem 3.4.3 depends only on d and ε and is independent of n. In several applications with $n \gg d$, our algorithm finds a $(1+\varepsilon)$-approximation in linear time in n and returns a core set whose size is independent of n. The identification of such a small set may play an important role in applications such as data classification.

Remark 2: The proof of Theorem 3.4.3 can be applied to Khachiyan's algorithm (i.e., Algorithm 4) as well and a core set can similarly be defined upon termination. However, Algorithm 4 uses an initial iterate all of whose components are positive. Therefore, for instances with $n \gg d$, Algorithm 4 will return $\mathscr{S}$ itself as a trivial core set. The reduction in the size of the core set is a consequence of using Algorithm 3 to obtain an initial iterate with a provably better lower bound on the optimal value of $(\mathbf{D}(\mathscr{S}))$.

Remark 3: Algorithm 5 does not lead to an improvement of the currently best known complexity result of Khachiyan in Ref. 17 for the minimum volume enclosing ellipsoid problem. The complexity results of the algorithms preceding Khachiyan's work (see Refs. 37, 18, 23) depend on $\log(R/r)$, where r and R denote the radii of two balls inscribed in and circumscribing the convex hull of the given point set, respectively. In particular, Nesterov and Nemirovskii in Ref. 18 develop an interior-point algorithm that computes a $(1+\varepsilon)$-approximation to the minimum volume enclosing ellipsoid of a set of n points in $\mathbb{R}^d$ in $O(n^{3.5}\log(Rn/r\varepsilon))$ operations. Khachiyan's algorithm calls Algorithm 4 with $\varepsilon = 1$, whose solution is used to obtain two balls with $R/r \leq 2d$, in $O(nd^2(\log d + \log\log n))$ operations and then uses the approximate solution as a warm-start in the interior-point algorithm, thereby reducing the overall complexity to $O(n^{3.5}\log(n/\varepsilon))$ operations. The use of Algorithm 5 instead of Algorithm 4 in Khachiyan's algorithm would only reduce the complexity of the first stage to $O(nd^2\log d)$ without having any effect on the complexity of the second stage, which determines the overall complexity.

3.5 Conclusions

In this chapter, we proposed and analyzed a first-order algorithm to compute an approximate minimum volume enclosing ellipsoid of a given set of n points in $\mathbb{R}^d$. We established that our algorithm returns a core set whose size depends only on d and ε. Especially for instances of the MVEE problem with $n \gg d$, our algorithm is capable of efficiently computing a small subset which provides a good representation of the input point set.

This chapter is an addition to the recent thread of works on core sets for several geometric optimization problems [132, 47, 46, 50, 2] and introduces, for the first time, the notion of core sets for minimum volume enclosing ellipsoids.

Since most applications of the MVEE problem have relatively small dimension d and ε is usually fixed, our algorithm has a complexity bound with the desirable property that its dependence on the number of points n is linear.

On the other hand, it is well-known that first-order algorithms suffer from slow convergence in practice – especially for smaller values of ε. Several interior-point methods developed for the MVEE problem perform well in practice and can achieve higher accuracy in reasonable time [125, 176, 199]. For instances of the MVEE problem with $n \gg d$, Sun and Freund [176] propose and implement a practical column generation approach using an interior-point algorithm to solve each subproblem. Motivated by the core set result established in this chapter and the encouraging computational results based on a core set result for the minimum enclosing ball problem [132], we intend to work on a column generation algorithm for the minimum volume enclosing ellipsoid problem with an emphasis on obtaining an upper bound on the number of columns generated to obtain a desired accuracy.

3.6 Future Work

For future work, we intend to explore practical implementations based on the idea of core sets. We report some preliminary results in figure 18. There are several interesting problems associated with core sets. For instance, does there exist an input set of points that provides a lower bound on the size of core sets? Can one establish similar core set results for other geometric optimization problems? Does there exist a unifying framework for core sets in general? We intend to pursue these research problems in the near future.

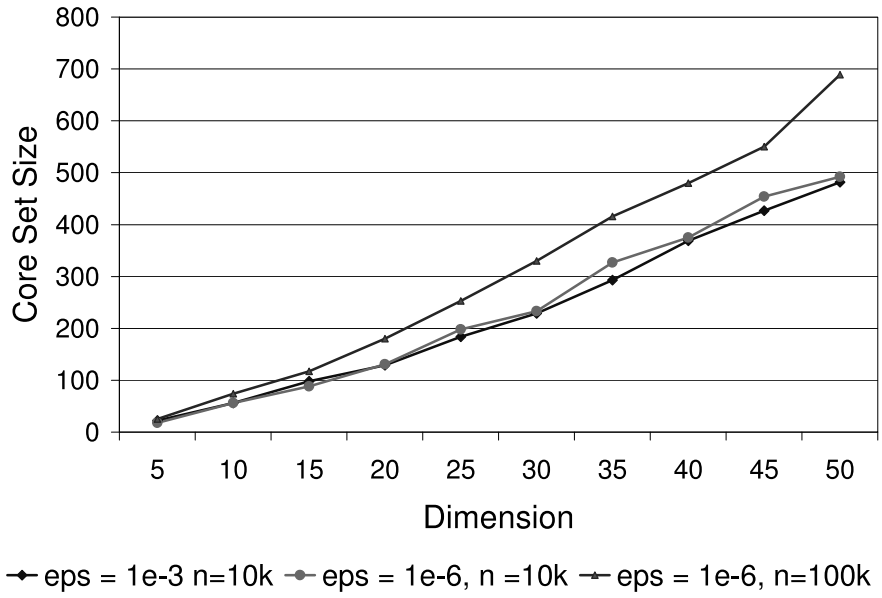


Figure 18: Preliminary results on the computation of core sets for the MVE problem. The implementation was done in MATLAB.

CHAPTER IV

Reconstruction

"Spring is a true reconstructionist."

HENRY TIMROD

The problem of curve and surface reconstruction from a given finite set of points has applications in computer graphics, computer vision, image processing, speech recognition and reverse engineering. In this chapter we describe a curve and surface reconstructor that takes unorganized points as input and outputs a piecewise-linear curve/surface interpolating the input point set.

Problem Statement: Given a finite set of points, called *sample points*, a shape reconstruction algorithm returns an approximation to the sample points. If the points are sampled from a curve and we want to output a reconstructed curve, the problem is known as the curve reconstruction problem. Surface reconstruction is analogous to curve reconstruction but here we want to output a manifold of some given higher dimension.

Our curve reconstructor is based on a provable reconstruction algorithm that is able to handle noise [59, 154]. The implementation is based on an algorithm to reconstruct a smooth closed curve from noisy samples. Our noise model assumes that the samples are obtained by first drawing points on the curve according to a locally uniform distribution,

followed by a uniform perturbation in the normal directions. Our reconstruction is faithful with probability approaching 1 as the sampling density increases. We expect that our approach can lead to provable algorithms that apply under less restrictive noise models and for handling non-smooth features.

We call our surface reconstructor, *Reviver* . The algorithm used in the design of *Reviver* is simple, fast, and provable. It first constructs a Delaunay triangulation of the input and then extracts the surface provably, assuming the sampling is dense. The theoretical framework provided by the pioneering work in surface reconstruction by Amenta and Bern[18] is used in the design of *Reviver* 's algorithm.

Figure 19: A point set with 55k points reconstructed by Reviver in less than one minute. This dataset has "sharp edges" and "borders."

We also maintain a web portal[1] on curve and surface reconstruction, which is ranked highly on search engines and is frequently visited by people in academia and industry.

[1] `http://www.compgeom.com/www.sites.html`

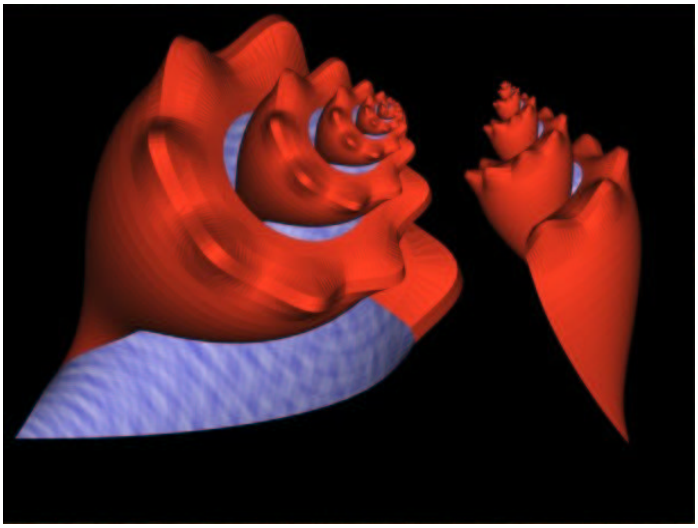


Figure 20: A point set with 30k points reconstructed by Reviver in 45 seconds. This dataset has "sharp edges" and "borders."

4.1 Curve Reconstruction

The combinatorial curve reconstruction problem has recently been extensively studied by computational geometers and has many applications, including image interpretation, surface reconstruction from contours, and terrain reconstruction in geographic analysis. The input to this problem consists of sample points on a collection, M, of unknown disjoint smooth closed curves. The problem calls for computing a set of polygonal curves that are provably *faithful*, meaning that, as the sampling density increases, the polygonal curves should converge to M.

Amenta et al. [19] obtained the first provable results on this problem. They proposed a *2D crust* algorithm whose output is provably faithful to reconstruct a set of disjoint smooth closed curves. They prove that the reconstruction is faithful if the input satisfies the *r-sampling* condition for any $r < 0.252$. For each point $x \in M$, the *local feature size* $f(x)$

at x is defined as the distance from x to the medial axis of M. For $0 < r < 1$, a set S of samples is an r-sampling of M if, for any point $x \in M$, there exists a sample $s \in S$ such that $\|s - x\| \leq r \cdot f(x)$ [19]. The algorithm by Amenta et al. invokes the computation of a Voronoi diagram or Delaunay triangulation twice. Gold and Snoeyink [105] simplified the algorithm and invoke the computation of a Voronoi diagram or Delaunay triangulation only once. Later, Dey and Kumar [73] proposed the *NN-crust* algorithm for this problem. This algorithm is simple and proceeds as follows. For each sample s in S, connect s to its nearest neighbor in S. Then, if a sample s is incident on only one edge e, connect s to the closest sample among all samples u such that su makes an obtuse angle with e. The output curve is faithful for any $r \leq 1/3$ [73].

Dey, Mehlhorn, and Ramos [74] proposed a *conservative-crust* algorithm to handle curves with endpoints. Funke and Ramos [98] proposed an algorithm to handle curves that may have sharp corners and endpoints. Dey and Wenger [75, 76] also described algorithms and implementation for handling sharp corners. Giesen [102] discovered that the traveling salesperson tour through the samples is a faithful reconstruction, but this approach cannot handle more than one curve. Althaus and Mehlhorn [13] showed that such a traveling salesperson tour can be constructed in polynomial time. Although the TSP-based approach handles corners naturally, it can only reconstruct a single curve. All other previously mentioned methods reconstruct a set of disjoint curves as well.

Noise often arises in collecting the input samples, e.g. when the input samples are obtained from 2D images by scanning. The noisy samples are typically classified into two types. The first type are samples that cluster around M but they generally do not lie on M. The second type are outliers that lie relatively far from M. To date, no combinatorial algorithm is known, that can compute a faithful reconstruction in the presence of noise. In

the paper with Cheng et.al. [59, 154], we propose a method that can handle noise of the first type for a set of disjoint smooth closed curves. We assume that the input does not contain outliers. Proving a deterministic result seems difficult as arbitrary noisy samples can collaborate to form patterns to fool any reconstruction algorithm. Instead, we assumed a particular model of noise distribution and proved that our reconstruction is faithful with probability approaching 1 as the number of samples increases. For simplicity and notational convenience, we will assume that $\min_{x \in M} f(x) = 1$ and that M consists of a single smooth closed curve, although our algorithm works when M contains more than one curve.

In our model, a sample is generated by drawing a point from M, followed by randomly perturbing the point in the normal direction. Let $L = \int_M \frac{1}{f(x)} dx$. The drawing of points from M follows the probability density function $\frac{1}{L \cdot f(x)}$. That is, the probability of drawing a point from a curve segment η is equal to $\int_\eta \frac{1}{f(x)} dx$ divided by L. A point p drawn from M is then perturbed in the normal direction. The perturbation is uniformly distributed within an interval that has p as the midpoint, width 2δ, and aligns with the normal direction at p. The distribution of each sample is independently identical. δ is the noise amplitude and we assume that $\delta \leq 1/(9\rho^2)$, where $\rho \geq 4$, is a constant chosen a priori by our algorithm. We assume throughout this chapter that $\delta > 0$. We emphasize that the value of δ is unknown to our algorithm. Although the perturbation along the normal direction is restrictive, it isolates the effect of noise from the distribution of samples on M. This facilitates an initial study of curve reconstruction in the presence of noise. Note that if $\delta > 1$, then the perturbed points from different parts of M will "mix up" at some place and it seems very difficult to estimate the unknown curve M around such a neighborhood.

We proved that our algorithm returns a reconstruction that is faithful with a probability of at least $1 - O\left(n^{-\Omega\left(\frac{\ln \omega n}{f_{\max}} - 1\right)}\right)$, where n is the number of input samples, ω is an arbitrary

positive constant, and $f_{\max} = \max_{x \in M} f(x)$. Our algorithm works for noisy samples from a collection of disjoint smooth closed curves. The novelty of our algorithm is a method to cluster samples so that each cluster comes from a relatively flat portion of M. This allows us to estimate points that lie close to M. We believe that this clustering approach will also be useful for less restrictive noise models and recognizing non-smooth features. We also expect that this clustering approach can be generalized to 3D for surface reconstruction problems.

4.1.1 The Algorithm

We only highlight the key ideas here. The reader is referred to [154] for details. Our algorithm works by growing a disk neighborhood around each sample p until the samples inside the disk fit in a strip whose width is small relative to the radius of the disk. The final disk is the *coarse neighborhood* of p denoted by $\mathit{coarse}(p)$. $\mathit{coarse}(p)$ provides a first estimate of the curve locally and of its normal. A better estimation is possible. We shrink $\mathit{coarse}(p)$ by a certain factor. We take a slab bounded by two parallel tangent lines of the shrunken $\mathit{coarse}(p)$. The slab is the *refined neighborhood* of p denoted by $\mathit{refined}(p)$. We rotate $\mathit{refined}(p)$ around p to minimize the spread of the samples in $\mathit{refined}(p)$ along the direction of $\mathit{refined}(p)$. The final orientation of $\mathit{refined}(p)$ provides a good normal estimation and it also allows us to estimate a *center point* close to M in place of p.

Next, we decimate the center points as follows. We scan the center points in decreasing order of the widths of their corresponding refined neighborhoods. When we add the current center point p^* to the decimated set, we delete the other center points that are too close to p^*. Finally, we can run any reconstruction algorithm that is correct for a noise free sampling on the remaining center points. For example, the NN-Crust algorithm by Dey and Kumar

[73].

In the following, we give a more detailed version of the algorithm. Let n be the total number of input samples. Let $\omega > 0$ and $\rho \geq 4$ be two predefined constants.

Point Estimation: For each sample s, we estimate a point as follows.

Coarse neighborhood: Let D be the disk that is centered at s and contains $\ln^{1+\omega} n$ samples. Let $\mathit{initial}(s)$ be the disk centered at s with radius $\sqrt{\text{radius}(D)}$. We initialize $\mathit{coarse}(s) = \mathit{initial}(s)$ and compute an infinite strip $\mathit{strip}(s)$ of minimum width that contains all samples inside $\mathit{coarse}(s)$. We grow $\mathit{coarse}(s)$ and maintain $\mathit{strip}(s)$ until $\frac{\text{radius}(\mathit{coarse}(s))}{\text{width}(\mathit{strip}(s))} \geq \rho$. The final disk $\mathit{coarse}(s)$ is the *coarse neighborhood* of s.

Refined neighborhood: Let N_s be a direction perpendicular to the long side of $\mathit{strip}(s)$. The *refined neighborhood* $\mathit{refined}(s)$ is the slab that contains s in the middle, parallel to N_s, and has width equal to $\min\{\sqrt{\text{radius}(\mathit{initial}(s))}, \text{radius}(\mathit{coarse}(s))/3\}$. We enclose the samples in $\mathit{refined}(s)$ by two parallel lines that are orthogonal to N_s. These two lines form a rectangle $\mathit{rectangle}(s)$ with the boundary lines of $\mathit{refined}(s)$. We rotate $\mathit{refined}(s)$ around s in the clockwise and counterclockwise directions and maintain $\mathit{rectangle}(s)$. The range of the rotation is $[0, \pi/10]$. Within this range, we position $\mathit{refined}(s)$ such that the height of $\mathit{rectangle}(s)$ in the direction N_s is minimized. We return the center point s^* of the final $\mathit{rectangle}(s)$.

Pruning: We sort the center points s^* in decreasing order of $\text{width}(\mathit{refined}(s))$. Then we scan the sorted list and select a subset of center points: when we select the current

center point s^*, we delete all center points u^* from the sorted list such that $\|s^* - u^*\| \leq \text{width}(\textit{refined}(s))^{1/3}$.

Output: We run the NN-crust algorithm on the selected center points and return the output curve.

4.1.2 A Crude Implementation

We report here on a preliminary implementation of the above algorithm. We made some changes to the above algorithm mostly due to ease of implementation. We do not estimate the refined neighborhood, but only use the coarse neighborhood in our code. It seems that, in practice, a coarse neighborhood could already give good results.

In the implementation, we find the line that minimizes the sum of the distances to the points, instead of solving the optimization problem that minimizes the width. We can solve this optimization problem using the eigenvectors of the covariance matrix. If $\lambda_1 > \lambda_2$ are the two eigenvectors of the covariance matrix, then we set $\textit{strip}(s) = \frac{1}{\sqrt{\lambda_1}}$ and we use this definition of $\textit{strip}(s)$ to reach the coarse neighborhood of s.

The current implementation does not include the pruning step. Therefore, for each sample point, the center of the coarse neighborhood is fed to the reconstruction algorithm at the next stage. This makes the output look jaggy, instead of being smooth.

Once the centers of the coarse neighborhood are estimated, we feed them to the reconstruction algorithm in the second stage. Instead of using the NN-Crust algorithm, we just output an approximate TSP of the input point set (see figure 21 and 22 for example outputs). We compute the TSP approximation using the concorde TSP solver (available at http://www.math.princeton.edu/tsp/concorde.html). We believe that implementing the pruning step of the algorithm should considerably improve the output as compared to the

Figure 21: An example reconstruction. Note the jaggedness of the reconstruction.

results of this crude implementation.

4.2 Surface Reconstruction

In this section we look at the algorithm and its implementation that goes into our surface reconstructor. For the case of surface reconstruction we will use M to denote a closed smooth manifold in $\mathbb{R}^3$. The algorithm can easily be generalized to the case in which M is a collection of disjoint, smooth, closed manifolds. The input to the problem is, as in curve reconstruction, a set of sample points, now given as a sampling of the manifold M. We use the following additional definitions:

- A *sliver* is a tetrahedron that is the convex hull of four points obtained, e.g., by picking a (fourth) point on the circumcircle of three points (in three dimensions) and perturbing it slightly out of the plane of the circle. The dihedral angles of a sliver are

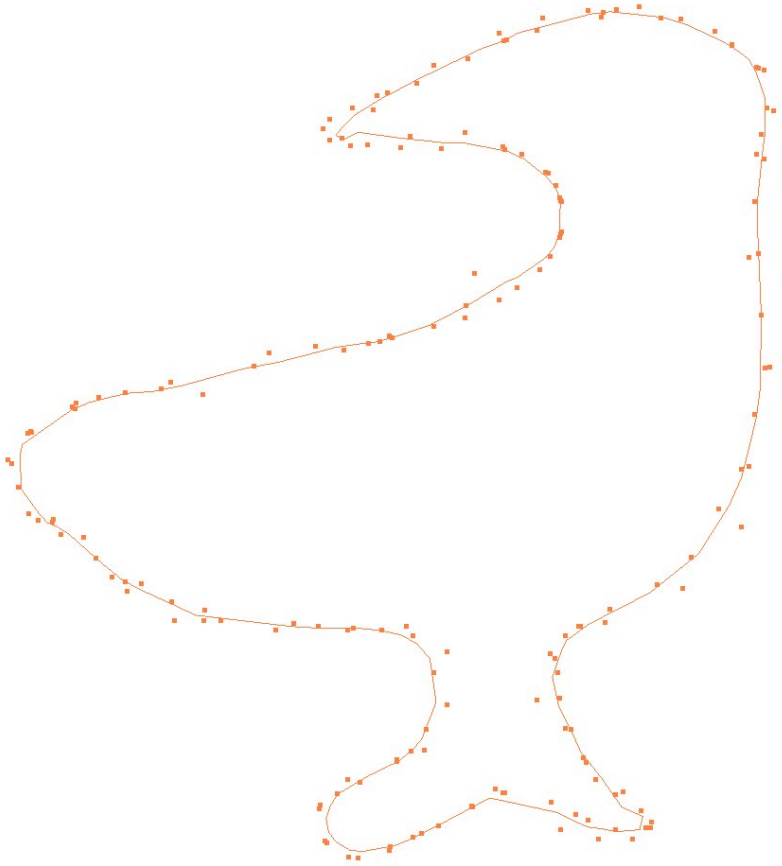


Figure 22: An example reconstruction of a noisy dataset. Note that the output is acceptable at sharp turns even though the algorithm is not guaranteed to work with non-smooth curves.

arbitrarily close to 0° and 180°; see figure 23.

✪ The *equatorial sphere* S_t is the smallest sphere passing through the vertices of a triangle t in $\mathbb{R}^3$. We define the *shrunk equatorial sphere* as a slight contraction of the equatorial sphere. More formally, a shrunk equatorial sphere for a triangle t is defined as the sphere S_t^ε with center at the circumcenter of t and radius equal to $1-\varepsilon$ times the circumradius. Here ε is a positive real number that is a function of r, where r is the parameter defining the r-sampling of the smooth manifold. The use of a shrunk equatorial sphere was first suggested in a theoretical result by Amenta and Choi [16]. We encountered the notion while examining a sliver in the *Cactus* dataset. We are not aware of any prior practical work in surface reconstruction that makes use of shrunk equatorial spheres. We will refer to a shrunk equatorial sphere test as a test for a triangle t that returns true if and only if S_t^ε is empty of sample points. This test can be done in $O(1)$ time, assuming that one has access to pointers to the two tetrahedra incident on each side of the triangle of the Delaunay tetrahedralization of the input point set. Algorithm 6 preprocesses the triangles of the Delaunay triangulation of the input points to create a subset of the triangles from which the surface is extracted. This subset of triangles passes the shrunk equatorial sphere test.

Algorithm 6 The shrunk equatorial sphere test on $\mathscr{D}$

Require: A Delaunay tetrahedralization $\mathscr{D}$ of P

1: $T_S^c = \phi$
2: **for** Each triangle $t \in \mathscr{D}_T$ **do**
3: **if** empty($\mathbb{S}_t^\varepsilon$) **then**
4: $T_S^c = T_S^c \cup t$
5: **end if**
6: **end for**
7: Return T_S^c

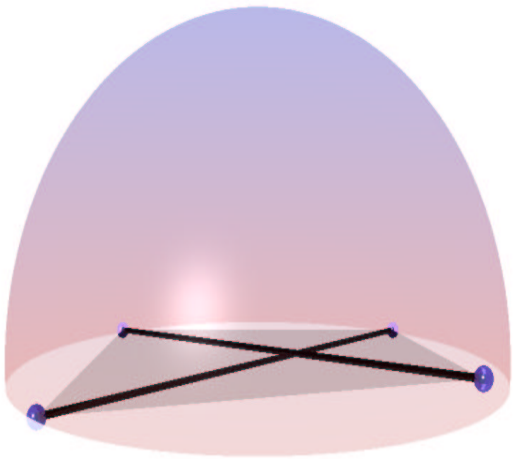


Figure 23: Example of a Sliver Tetrahedron

The basic algorithm is very similar to a minimum spanning tree (MST) algorithm. Of course, the minimum spanning tree in $\mathbb{R}^3$ is not a surface but a set of edges, which is quite sparse [113]. *Reviver* starts by creating the Delaunay tetrahedralization of the input point set. We will denote the set of triangles in the Delaunay tetrahedralization by $\mathscr{D}_T$. *Reviver* takes as input a "good" triangle, used as a seed triangle, t_g, that is in the output surface reconstructed. We will see later how to select this first triangle. *Reviver* at every instant of the construction of the surface maintains two sets, a set T_S consisting of good triangles and another set T_S^c consisting of the triangles to be considered for inclusion in T_S. T_S is always maintained with a correct orientation. The set T_S is initialized by the input triangle. T_S^c is initialized by computing the Delaunay triangulation $\mathscr{D}$ of the input point set, filtering the triangles in $\mathscr{D}$ using Algorithm 6 and then removing $\{t_g\}$ from the output. After this step, we greedily add triangles to T_S from T_S^c. Note that this is very similar to greedy algorithms used in constructing minimum spanning trees. Another interpretation of this computation

is that *Reviver* computes the minimum spanning tree (MST) of the dual of the *restricted Delaunay triangulation*, which is defined as the dual of the restricted Voronoi diagram.

Each "bare[2] " edge of T_S is maintained in a priority queue. The weight w_e of a bare edge e is calculated in the following manner. Let $t \in T_S$ be a triangle that includes e, and let C_t be its circumcenter. Let t' be the triangle incident on e in T_S^c whose circumcenter $C_{t'}$ minimizes the distance $|C_t C_{t'}|$. We initialize w_e with $|C_t C_{t'}|$. If there are slivers found on a candidate triangle, the weight of the bare edge is increased by a large constant. Hence the bare edges that have slivers incident on them are processed later compared to the other bare edges of T_S. Intuitively, this makes the front explore the areas where it is safest to go.

Similar to the MST approach we include the triangle $t' \in T_S^c$ incident on the bare edge e with the least weight in the priority queue if and only if:

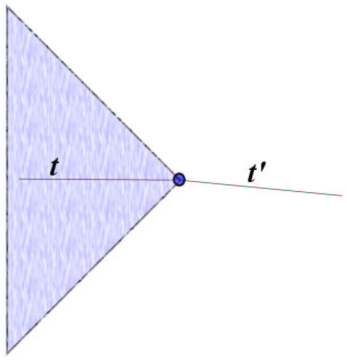


Figure 24: Forbidden region of t'

❶ t' and t do not make a dihedral angle of less than $\frac{\pi}{4}$ [3] (see figure 24).

[2] A bare edge of T_S is an edge on which there is only one triangle from T_S incident.
[3] This is a generalization to the curve reconstruction idea of Dey and Kumar [73]

❷ If in the normal direction of t, t' has a sliver incident on it, select the next triangle in the following manner.

- Go to the next triangle towards the normal to t which makes a sliver angle with t'. Let this triangle be t' now.
- Repeat the previous step if t' still has a sliver incident on it towards the normal to t.

❸ If conditions ❶ and ❷ are satisfied, check that t' does not create a non-manifold at each of its three vertices with triangles in T_S

Once an edge is popped off the priority queue and a triangle is included in T_S on that edge, all triangles incident on that edge in T_S^c are deleted. This manifold enforcement step also makes T_S^c smaller as the front grows and hence makes it faster to search for triangles that might be candidates for inclusion in the front. *Reviver* enforces that all edges in T_S have only two triangles incident on them and every vertex in T_S has no non-manifold triangle incident on it. This also reduces the chances of a triangle getting selected that is incident on a vertex through which a manifold is already passing in T_S. For instance in the *Lyria*[4] dataset, the vertex manifold condition helps reconstruct the border triangles perfectly (see figure 20).

One non-trivial task has been ignored in the previous description: How do we select the first (seed) triangle? We could devise a complicated approach using poles to select the first triangle. Alternatively, we could do the following: Select a triangle that has one vertex on the convex hull. For doing this, the topmost vertex could be chosen in the point set and then, since the shortest edge incident on it has to be in the reconstruction, we could

[4] *Courtesy of Claudio Rocchini*

reduce the problem to choosing a triangle on the *fan* of this edge (i.e., among all triangles passing the shrunk equatorial sphere test incident on this edge). The smallest circumradius triangle could be selected if there were no slivers incident on it; otherwise, we could pick the triangle that was on top of this sliver system.

We did not implement either of these more complicated methods in *Reviver* . Instead, we simply pick the triangle with the maximum sum of z-coordinates among those triangles that pass the shrunk equatorial sphere test. This choice worked for all point sets we considered. For a non-smooth manifold this is not guaranteed to work, although for a smooth manifold, it seems that this will almost always work. Here is the intuition of why: Let t be the first triangle we picked and let it be a wrong triangle. Let v be the top most vertex of this triangle. Since the surface passing through v is smooth, it has to be a maximum since the surface is not allowed to go above this vertex in all directions. Assuming our sampled surface is a smooth surface, consider its projection on a plane that passes through v, the centroid of t, and the normal of t. Since the surface is smooth, it will have to go inside the triangle's circumsphere in this direction. If an adversary makes it small enough to elude it from the constraining factor ε, it will be necessary to reduce the feature size at v.

4.3 How does MST Help?

Let us now turn our attention to the very heart of *Reviver* , the minimum spanning tree (MST). The MST has been used by reconstruction researchers for quite some time, in various ways [143, 113]. In this section we explain why we used the MST of the dual of the triangles that are in the reconstruction.

The answer depends on two other questions:

⋆ How does one handle slivers?

Algorithm 7 Reviver's Main Algorithm: A Simple Advancing Front Adaptation

Require: T_S^c of P and a seed triangle t_g

Priority Queue $Q < edge, float >$
$T_S^c = T_S^c \setminus \{t_g\}$
Advancing Front $\mathscr{F} = t_g$
Calculate costs c_i for all edges of t_g
Push the edges $e_i \in t_g$ into Q with cost c_i
while Q.notempty() **do**
    Pop the minimum cost edge e from Q
    if $\exists t'$ on e that can be added to $\mathscr{F}$ **then**
        Run conditions ❶, ❷ and ❸ on t'
        Correct Orientation of t'
        $\mathscr{F} = \mathscr{F} \cup t'$
        Enforce e to be a Manifold
        if t' completes an umbrella on v **then**
            Remove all $t \in T_S^c$ incident on v
        end if
        Let E be the new set of exposed edges
        if $e' \in E$ has a candidate triangle **then**
            Q.push(e',Cost(e'))
        end if
    end if
end while
Return $\mathscr{F}$

- How does one do reconstruction when the input is not theoretically what it should be?

We try to give the intuition behind why MST helps for both the questions described above.

In Algorithm 6 the first loop (Lines 2-6) filters all triangles based on encroachment of their *shrunk equatorial spheres*. What happens when a triangle is part of a circle lying in 3D that has its vertices slightly perturbed and can afford to pass through itself lots of slivers? This can also be visualized as intersection of a small sphere centered on a smooth closed manifold and the manifold itself and the points sampled on the periphery of the intersecting "wavy" disc. In this case, the shrunk equatorial sphere is not encroached for any of the triangles since the sampling is assumed to be good and the other side of the surface is far away. Remember that if the front encounters a sliver sitting on it, then it does not process them until it has no other triangles to process (Because the weights of the triangles that have slivers incident on them is chosen to be very high). That means the front avoids slivers, as long as it can. What happens when it cannot avoid it? It picks an edge, and goes on the topmost triangle of the slivers sitting at one place. Let us see what happens in this situation.

Let t be the new triangle selected and added to the front at the common edge e. The manifold enforcement at vertices and edges of the front means that e will only have two triangles incident on it now. Look at the other triangles of the sliver system: Their circumcenters almost coincide with the circumcenter of t. Since we use the MST to pick new triangles, the triangle that can be chosen are the only ones that are incident on t's exposed edges. In other words: until an upper umbrella of triangles is constructed by the algorithm to cover a sliver, it can not pick a triangle from the boundary of the sliver to inside and

hence forming a non-manifold that needs backtracking to form a manifold as was done in Amenta and Bern [18].

If there are multiple slivers overlapping on the surface in such a way that their circumcenters are different but their circumspheres overlap, MST still helps to form an umbrella over the union of these slivers in a consistent fashion.

As far as the second question is concerned, there are many datasets in this chapter in which an algorithm designed for only smooth surfaces would never have a chance to work. Experimentally, we found out that the MST helps in getting a better reconstruction compared to without using it, especially for sharp edges. When the algorithm does not know where to go next, it tries to apply a greedy strategy, and picks the one that is nearest to the front since proximity is always the prime criterion for any reconstruction algorithm. The MST favors the front propagation to be in the direction of low curvature in the data if the sampling is somewhat uniform. It also helps to avoid ambiguous situations encountered when trying to propagate the front across sharp edges (as for example in the dataset Lyria).

4.4 Provability

Theorem 4.4.1 *If there are no slivers in the Delaunay tetrahedralization of P, Reviver outputs the correct reconstruction.*

Lemma 4.4.1 (Walk) *If e is a common edge of two triangles t_1, t_2 in the reconstruction, that t_2 has no slivers incident on them, then t_2 has the smallest circumradius among triangles incident on $e \in T_S^c$ that are not in the forbidden region of t_1 on e.*

Conjecture: *Let t_g be a good triangle given to us by an oracle. Then $\mathscr{F}$ always adds correct triangles, even in case of slivers.*

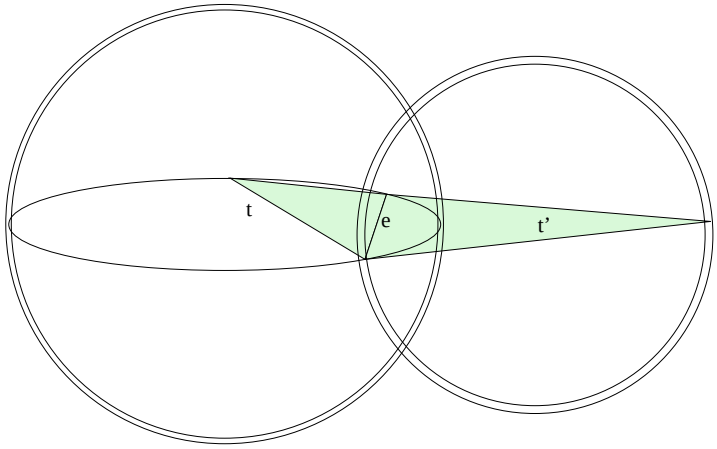


Figure 25: Either t' lies on a sliver, or there is no ambiguity in choosing t' when t is known

Intuition: We use induction here. Since the base case is correct, we assume that the current $\mathscr{F}$ is correct. Let t be the new triangle added on edge e. If it is not a part of a sliver system, then it is correct by the Walk lemma. Otherwise, it is the topmost triangle of a sliver system on the edge e. Once one triangle from a sliver system is placed into the set T_S, the circumcenters of the other triangles left in T_S^c of the sliver system are very close to the picked triangle (in terms of the circumcenter distance). Hence, the front covers the umbrella on the sliver system before it picks any contradicting triangle to the umbrella for the sliver system.

Theoretically, the only thing we can guarantee about reviver is that it can reconstruct point sets where the advancing front $\mathscr{F}$ never encounters a sliver (this can be proved using lemmas from [18]). Practically we have engineered *Reviver* to work on slivers.

Also note that not even one dataset that is reported to be reconstructed by *Reviver* here, follows the sampling criterion of Amenta and Bern [18] although it forms the backbone of how *Reviver* was designed.

4.5 Implementation

Reviver takes as input, unorganized points in $\mathbb{R}^3$ reading the input as an array of `floats`. It works internally with double precision. *Reviver* can also filter triangles based upon an α value [85]. This is convenient when the input sampling is not good enough. The triangles can be filtered using an α value until good triangles are there in the set of output triangles and then the usual algorithm can be applied. *Reviver* provides options to stop the reconstruction after the α-`Hull` computation or to continue the algorithm after the hull computation. Sometimes the *shrunk equatorial sphere* test with a good α value is a good approximation to the mesh desired and no further computation is required. This is especially true for uniform dense samplings.

Figure 26: Torso: Almost entirely made up of slivers!

Making Qhull work for reconstruction took quite a bit of time in our implementation. A few times, Qhull gave problems when there

were many cocircularities in the input. For instance, the *Torso* dataset is almost totally

made up of cocircular points (see figure 26).. This makes Qhull take a long time to compute the Delaunay triangulation. We used Qhull with an implementation of our own joggle function that converts every float to double by joggling an extra 8 bits in the mantissa of the double after moving the points to their centroid and scaling them. Qhull was also instructed to check the convexity at each vertex using the *Tv* option. This takes considerably more time in some cases than using (usual) Qhull on these point sets, especially for the *Torso*.

The implementation also shifts the entire point set to the mean of the coordinates before processing and scales them into a unit cube. This is done so that the value of ε for the shrunk equatorial sphere test can be chosen as a constant, as opposed to choosing a different value for different scales. The movement to mean also frees up valuable bits from the input most of the time so that computations can be performed more accurately.

Currently, *Reviver* supports output in DXF, VRML, POV and OFF formats and works on Linux, Windows NT and Irix Platforms. Its source code is a mixture of C++ and C. Until the shrunk equatorial sphere test, almost all steps that are time-intensive are written in C, except for those cases in which we use LEDA to perform exact arithmetic. After that most of the code is in C++ . We did this because the major bottleneck in speed was getting the triangles after the shrunk equatorial sphere test. The advancing front algorithm works very fast and uses LEDA's fast implementation of priority queues.

4.5.1 Equatorial Sphere Encroachment

The problem of deciding whether a point is inside a equatorial sphere of a triangle t is quite a common problem in mesh generation and computer graphics. There are codes available from various sources on the web [89, 83] for calculating the center of such a Sphere. Both

of these codes do not solve the encroachment problem exactly[5] . This problem can be solved trivially by computing the center first and then checking the distance of the vertex that needs to be checked for encroachment from the computed center. The shrunk equatorial sphere test just needs to multiply the radius of an equatorial sphere by $1-\varepsilon$ and run the usual equatorial sphere test on this sphere.

Reviver uses a floating point filter system to do this test. In the worst case it uses LEDA's exact arithmetic to do the test.

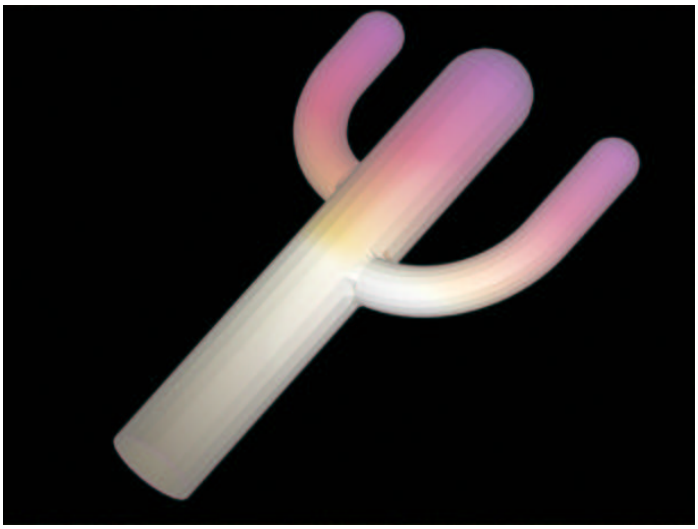


Figure 27: Cactus: This point set has sharp turns.

4.6 Future Work

The main bottleneck in reconstruction is using the Delaunay triangulation. The Qhull implementation may be slower than Jonathan Shewchuk's pyramid code, which is not yet published. We wrote *Reviver* in such a way that the Delaunay triangulator and the surface

[5] Recently Jonathan Shewchuk suggested that adding one line to his code could have removed the division by zero that we were getting when we tried to use his code for circumcenter computation.

Figure 28: A reconstruction of a Head point set.

Figure 29: A reconstruction of a Pear point set.

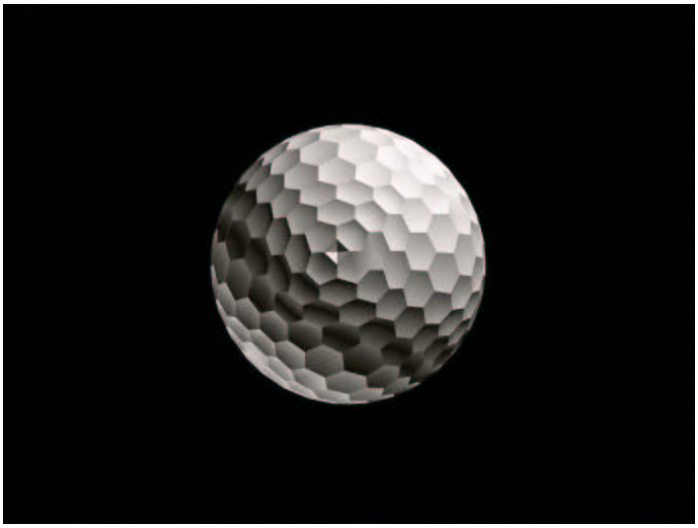


Figure 30: A reconstruction of a Epcot point set.

Figure 31: A reconstruction of a Fist point set.

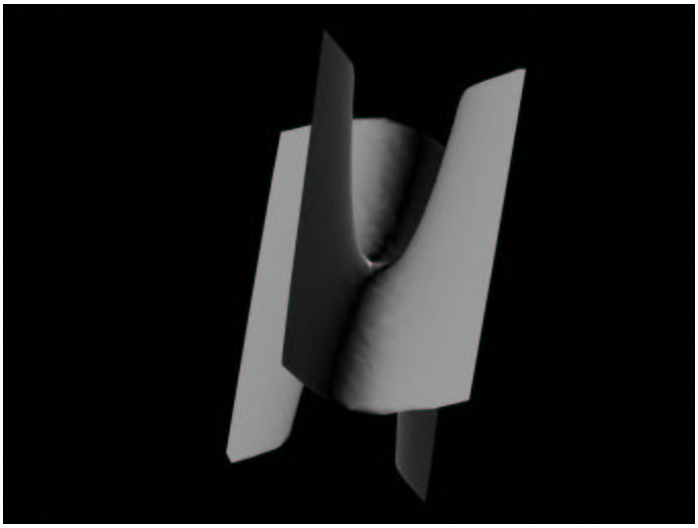


Figure 32: A reconstruction of a Hyper dataset.

Figure 33: The Stanford Bunny reconstructed.

Figure 34: Cat Dataset, *Reviver* cannot reconstruct this dataset properly because it has sharp edges that are in a 'V' shape forming an angle of less than $\frac{\pi}{4}$. This happens especially near the eyes and tail portions.

Point Set	Points	Time in Sec
Hands	54698	58
Cactus	3337	25
Head	12770	35
Lyria	29970	45
Pear	891	3
Epcot	770	1
Fist	16384	30
Torso	12947	392
Hyper	8525	20
Bunny	35947	35
Cat	7340	12

Table 1: Timings for reconstructions. The $\alpha-$filter was switched off when taking the timings. The value of ε that was used to construct the shrunk equatorial sphere was set to 2×10^{-5} in all these reconstuctions. The value for the sliver angle used in all these reconstructions was $\frac{\pi}{12}$.

extractor are two completely separate modules. Can the Delaunay triangulation be avoided totally by a practical algorithm, yet keeping it provable in some sense?

In *Reviver* we implemented a small perturbation module that perturbs all points after reading them. This helps breaking up degeneracies in the input point set although it of course does not guarantee the removal of gratuitous slivers. As mentioned before, *slivers* pose a major challenge to surface reconstructors. If there were no slivers in a 3D Delaunay tetrahedralization, surface reconstruction would much easier, at least for smooth closed surfaces sampled densely. Is there a practical algorithm that can remove slivers from a Delaunay tetrahedralization by giving them a small perturbation while not taking too much time to do this?

Another important question is whether recent theoretical results can be turned into practical solutions for reconstruction without compromising speed and output quality.

CHAPTER V

Cache Oblivious Algorithms

"Memory is like an orgasm.
It's a lot better if you don't have to fake it."

SEYMORE CRAY

A dream machine would be fast and would never run out of memory. Since an infinite sized memory was never built, one has to settle for various trade-offs in speed, size, and cost. In both the past and the present, hardware suppliers seem to have agreed on the fact that these parameters are well optimized by building what is called a memory hierarchy. Memory hierarchies optimize the three factors mentioned above by being cheap to build, trying to be as fast as the fastest memory present in the hierarchy and being almost as cheap as the slowest level of memory. The hierarchy inherently makes use of the assumption that the access pattern of the memory has *locality* in it and can be exploited to speed up the accesses.

The locality in memory access is often categorized into two different types, code reusing recently accessed locations (*temporal*) and code referencing data items that are close to recently accessed data items (*spatial*) [111]. Caches use both temporal and spatial locality to

improve speed. Surprisingly many things can be categorized as caches, for example, registers, L1, L2, TLB, Memory, Disk, Tape etc. The whole memory hierarchy can be viewed as *levels* of caches, each transferring data to its adjacent levels in atomic units called *blocks*. When data that is needed by a process is in the cache, a *cache hit* occurs. A *cache miss* occurs when data can not be supplied. Cache misses can be very costly in terms of speed. Cache misses can be reduced by designing algorithms that use locality of memory access.

The *Random Access Model* (RAM) in which we do analysis of algorithms today does not take into account differences in speeds of random access of memory depending upon the locality of access [67]. Although there exist models which can deal with multi-level memory hierarchies, they are quite complicated to use [5, 7, 12, 11, 6, 112, 166, 167]. It seems there is a trade-off between the accuracy of the model and the ease of use. Most algorithmic work has been done in the RAM model which models a 'flat' memory with uniform access times. The external memory model is a two level memory model, in the context of memory and disk. The 'ideal' cache oblivious model is a step towards simplifying the analysis of algorithms in light of the fact that local accesses in memory are cheaper than non-local ones in the whole memory hierarchy (and not just two-levels of memory). It helps take into account the whole memory hierarchy and the speed differences hidden therein.

In the external memory model, algorithm design is focused on a particular level of memory (usually the disk) which is the bottleneck for the running time of the algorithm in practice. Recall that in the external memory model, processing works almost as in the RAM model, except that there are only M words of internal memory that can be accessed quickly. The remaining memory can only be accessed using I/Os that move B contiguous words between external and internal memory. The I/O complexity of an algorithm amounts

to counting the number of I/Os needed.

The cache oblivious model was proposed in [97] and since then has been used in more than 30 papers already. It is becoming popular among researchers in external memory algorithms, parallel algorithms, data structures, and other related fields. This model was born out of the necessity to capture the hierarchical nature of memory organization. (For instance the Intel Itanium has 7 levels in its memory hierarchy). Although there have been other attempts to capture this hierarchical information the cache oblivious model seems to be one of the most simple and elegant ones. The cache oblivious model is a two level model (like the [8] model that has been used in the other chapters so far) but with the assumption that the parameters M,B are unknown to the algorithm (see figure 35). It can work efficiently on most machines with multi-level cache hierarchies. Note that in the present volume, all external memory algorithms that have been dealt with yet, need the programmer/user to specify M,B.

This chapter is intended as an introduction to the design and analysis of cache oblivious algorithms, both in theory and practice.

Chapter Outline: We introduce the cache oblivious model in section 5.1. In section 5.2 we elaborate some commonly used design tools that are used to design cache oblivious algorithms. In section 5.3 we choose matrix transposition as an example to learn the practical issues in cache oblivious algorithm design. We study the cache oblivious analysis of Strassen's algorithm in section 5.4. Section 5.5 discusses a method to speed up searching in balanced binary search trees both in theory and practice. In section 5.6, a theoretically optimal, randomized cache oblivious sorting algorithm along with the running times of an implementation is presented. In section 5.7 we enumerate some practicalities not caught by the model. Section 5.8 presents some of the best known bounds of other

cache oblivious algorithms. Finally we conclude the chapter by presenting some related open problems in section 5.9.

5.1 The Model

The ideal cache oblivious memory model is a two level memory model. We will assume that the faster level has M size and the slower level always transfers B words of data together to the faster level. These two levels could represent the memory and the disk, memory and the cache, or any two consecutive levels of the memory hierarchy (see figure 35). In this chapter, M and B can be assumed to be the sizes of any two consecutive levels of the memory hierarchy subject to some assumptions about them (For instance the inclusion property which we will see soon). We will assume that the processor can access the faster level of memory which has size M. If the processor references something from the second level of memory, an I/O fault occurs and B words are fetched into the faster level of the memory. We will refer to a *block* as the minimum unit that can be present or absent from a level in the two level memory hierarchy. We will use B to denote the size of a *block* as in the external memory model. If the faster level of the memory is full (i.e. M is full), a block gets evicted to make space.

The ideal cache oblivious memory model enables us to reason about a two level memory model like the external memory model but prove results about a multilevel memory model. Compared with the external memory model it seems surprising that without any memory specific parametrization, or in other words, without specifying the parameters M, B, an algorithm can be efficient for the whole memory hierarchy, nevertheless it is possible. The model is built upon some basic assumptions which we enumerate next.

Assumptions: The following four assumptions are key to the model.

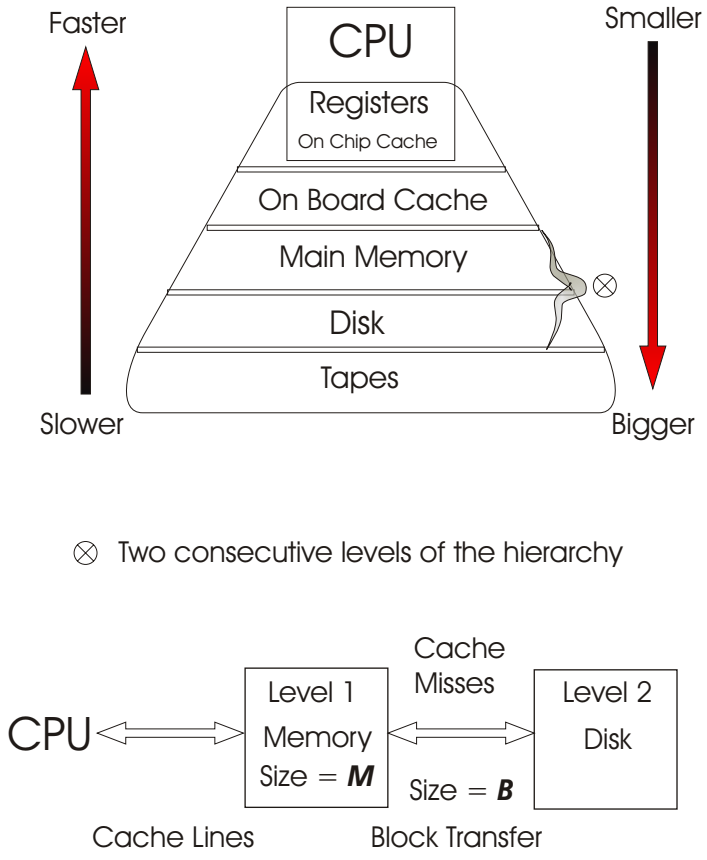


Figure 35: The ideal cache oblivious model.

Optimal replacement The *replacement policy* refers to the policy chosen to replace a block when a cache miss occurs and the cache is full. In most hardware, this is implemented as FIFO, LRU or Random. The model assumes that the cache line chosen for replacement is the one that is accessed furthest in the future. The strategy is called *optimal off-line replacement* strategy.

2 levels of memory There are certain assumptions in the model regarding the two levels of memory chosen. They should follow the *inclusion property* which says that data cannot be present at level i unless it is present at level $i+1$. In most systems, the inclusion property holds. Another assumption is that the size of level i of the memory hierarchy is strictly smaller than level $i+1$.

Full associativity When a block of data is fetched from the slower level of the memory, it can reside in any part of the faster level.

Automatic replacement When a block is to be brought in the faster level of the memory, it is automatically done by the OS/hardware and the algorithm designer does not have to care about it while designing the algorithm. Note that we could access single blocks for reading and writing in the external memory model, which is not allowed in the cache oblivious model.

We will now examine each of the assumptions individually. First we consider the optimal replacement policy. The most commonly used replacement policy is LRU (*least recently used*). In [97] the following lemma, whose proof is omitted here, was proved using a result of [172].

Lemma 5.1.1 ([97]) *An algorithm that causes $Q^*(n,M,B)$ cache misses on a problem of size n using a (M,B)-ideal cache incurs $Q(n,M,B) \leq 2Q^*(n,\frac{M}{2},B)$ cache misses on a*

(M,B) *cache that uses LRU, FIFO replacement. This is only true for algorithms which follow a regularity condition.*

An algorithm whose cache complexity satisfies the condition $Q(n,M,B) \leq O(Q(n,2M,B))$ is called *regular* (All algorithms presented in this chapter are regular). Intuitively, algorithms that slow down by a constant factor when memory (M) is reduced to half, are called regular. It immediately follows from the above lemma that if an algorithm whose number of cache misses satisfies the regularity condition does $Q(n,M,B)$ cache misses with optimal replacement then this algorithm would make $\Theta(Q(n,M,B))$ cache misses on a cache with LRU or FIFO replacement.

The automatic replacement and full associativity assumption can be implemented in software by using LRU implementation based on hashing. It was shown in [97] that a fully associative LRU replacement policy can be implemented in O(1) expected time using O($\frac{M}{B}$) records of size O(B) in ordinary memory. Note that, the above description about the cache oblivious model also proves that any optimal cache oblivious algorithm can also be optimally implemented in the external memory model.

We now turn our attention to multi-level ideal caches. We assume that all the levels of this cache hierarchy follow the inclusion property and are managed by an optimal replacement strategy. Thus on each level, an optimal cache oblivious algorithm will incur an asymptotically optimal number of cache misses. From Lemma 5.1.1, this becomes true for cache hierarchies maintained by LRU and FIFO replacement strategies.

Apart from not knowing the values of M,B explicitly, some cache oblivious algorithms (for example optimal sorting algorithms) require a *tall cache* assumption. The tall cache assumption states that $M = \Omega(B^2)$ which is usually true in practice. It is notable that regular optimal cache oblivious algorithms are also optimal in SUMH [11] and HMM [5]

models. Recently, compiler support for cache oblivious type algorithms have also been looked into [189, 196].

5.2 Algorithm design tools

In cache oblivious algorithm design some algorithm design techniques are used ubiquitously. One of them is a *scan* of an array which is laid out in contiguous memory. Irrespective of B, a scan takes at most $1 + \lceil \frac{N}{B} \rceil$ I/Os. The argument is trivial and very similar to the external memory scan algorithm. The difference is that in the cache oblivious setting the buffer of size B is not explicitly maintained in memory. In the assumptions of the model, B is the size of the data that is always fetched from level 2 memory to level 1 memory. The scan does not touch the level 2 memory until its ready to evict the last loaded buffer of size B already in level 1. Hence, the total number of times the scan algorithm will force the CPU to bring buffers from the level 2 memory to level 1 memory is upper bounded by $1 + \lceil \frac{N}{B} \rceil$. It is easy to show that the scan of an array will at most load $1 + \lceil \frac{N}{B} \rceil$ buffers from level 2.

5.2.1 Main Tool: Divide and conquer

Chances are that the reader is already an expert in divide and conquer algorithms. This paradigm of algorithm design is used heavily in both parallel and external memory algorithms. It should not come as a surprise to the reader that cache oblivious algorithms make heavy use of this paradigm and lot of seemingly simple algorithms that were based on this paradigm are already cache oblivious! For instance, Strassen's matrix multiplication, quicksort, mergesort, closest pair [67], convex hulls [15], median selection [67] are all algorithms that are cache oblivious, though not all of them are optimal in this model. This

means that they might be cache oblivious but can be modified to make fewer cache misses than they do in the current form. In the section on matrix multiplication we will see that Strassen's matrix multiplication algorithm is already optimal in the cache oblivious sense.

Why does divide and conquer help in general for cache oblivious algorithms? Divide and conquer algorithms split the instance of the problem to be solved into several sub problems such that each of the sub problems can be solved independently. Since the algorithm recurses on the sub problems, at some point of time, the sub problems fit inside M and subsequent recursion, fits the sub problems into B. For instance let us analyze the average number of cache misses in a randomized quicksort algorithm. This algorithm is quite cache friendly if not cache optimal and is described in Section 8.3 of [67].

Lemma 5.2.1 *Randomized version of quicksort incurs an expected number of* $\mathsf{O}\left(\frac{N}{B}\log_2\left(\frac{N}{B}\right)\right)$ *cache misses.*

Proof. Choosing a random pivot makes at most one cache miss. Splitting the input set into two output sets, such that the elements of one are all less than the pivot and the other greater than or equal to the pivot makes at most $\mathsf{O}(1+\frac{N}{B})$ cache misses by Exercise **??**.

As soon as the size of the recursion fits into B, there are no more cache misses for that subproblem ($Q(B) = O(1)$). Hence the average cache complexity of a randomized quicksort algorithm can be given by the following recurrence (which is very similar to the average case analysis presented in [67]):

$$Q(N) = \frac{1}{N}\left[\sum_{i=1..\frac{N-1}{B}} (Q(i)+Q(N-i)) + \left(1+\left\lceil\frac{N}{B}\right\rceil\right)\right] \tag{59}$$

which can be simplified to

$$Q(N) = \frac{2}{N}\left[\sum_{i=1..\frac{N-1}{B}} Q(i) + \Theta\left(1+\frac{N}{B}\right)\right] \tag{60}$$

which solves to $O(\frac{N}{B}\log_2 \frac{N}{B})$ [108]. A similar analysis also shows that mergesort does $O\left(\frac{N}{B}\log_2 \frac{N}{B}\right)$ cache misses. □

As we said earlier, the number of cache misses randomized quicksort makes is not optimal. The sorting lower bound in the cache oblivious model is also the same as the external memory model. We will see later in the chapter a sorting algorithm that does sorting in $O\left(\frac{N}{B}\log_{\frac{M}{B}} \frac{N}{B}\right)$ cache misses.

We now will analyze another cache oblivious algorithm that is also optimal, the *median finding* as given in [67] (Section 10.3). The recurrence for finding the median of N entities is given by:

$$Q(N) = Q\left(\frac{N}{5}\right) + Q\left(\frac{7N}{10}+6\right) + O\left(1+\frac{N}{B}\right)$$

Solving which we get, $Q(N) = O\left(1+\frac{N}{B}\right)$. Note that again $Q(B) = O(1)$ so the recursion stops adding cost to cache misses when the instance of the problem becomes smaller than B.

5.2.2 Randomization

Two basic tools that are almost always used in randomized algorithms are selection and permutation. Permuting a set or an array randomly can be done exactly the way distribution sort (see section 5.6) is done, except, that when comparing two objects, the outcome is defined by a random coin. We don't know of a simple way to do randomized shuffling cache obliviously yet. The same sorting lower bounds hold for shuffling too.

Random subset selection is a more interesting problem. In this section we describe how one could pick a random subset of size n^{ε} from an array of size n without incurring too many cache misses. This could be done in work complexity $O(n^{\varepsilon})$ and cache complexity $O\left(min\left(1+n^{\varepsilon}, \frac{N}{B}\right)\right)$. Use algorithm S of Knuth [130], Sec 3.4.2 with $N = n, n = n^{\varepsilon}$.

By using the Hyper-Geometric distribution, a method mentioned as an exercise in Knuth's book is cache oblivious for sampling without replacement. We can sample with or without replacement k objects from n objects by incurring almost optimal number of cache faults.

This seems to be faster than linearly scanning the whole input array if the array is very big and in practice, this method looks quite promising. For instance we could generate 2^{15} random numbers in sorted order for $n = 2^{30}$ in .2 seconds on a 1GHz Laptop without any effort to optimize the code.

Another method good for practical implementations is to generate n^{ε} numbers in an array ranging from $1..n$, sort the array and then pick the unique elements using a linear scan. Another very simple method to choose random samples is the use of Bernoulli trials.

In section 5.6 we use sampling with replacement. This kind of sampling appears wasteful, but we use it here to keep the analysis clean. Sampling without replacement would result in a marginally sharper analysis, at the cost of complicating the analysis.

5.2.3 Amortization

This technique of analysis can be used to show that the average cost of an operation is small, even though a single operation might take a long time to complete. It guarantees average performance of an operation in the worst case. Amortized analysis is helpful in gaining an insight of the design of a particular data structure or algorithm [67].

In this section we take an example to illustrate how amortization could help in design and analysis of a cache oblivious data structure called the packed memory structure [32]. This structure can help one maintain a dynamic van Emde Boas layout (section 5.5) of a strongly weight balanced search tree [32]. This structure can also be used as a cache oblivious linked list data structure and has been used to design dynamic dictionaries [33]. Our description mostly follows [32].

In the packed memory problem, also known as ordered file maintenance problem, we want to store N elements in an array of size $|A| = O(N)$. Note that $|A|$ is not N but cN for some $c > 1$.

We are supposed to design a data structure that supports inserts and deletes without doing too many cache misses and without taking too much time. The inserts in this structure are similar to linked list inserts, they also come with a pointer to the element after which the element needs to be inserted. Another constraint of the structure is the density constraint, any set of k contiguous elements are stored in a contiguous array of $O(k)$. It is not hard to imagine that to create such a structure, one should cleverly maintain gaps between elements such that inserts and deletes don't do too much damage in running time or locality of operation. Roughly speaking, when a sub-array becomes too full or too empty, one could evenly redistribute the elements in a larger sub-array. The sub-array sizes range from $O(\log N)$ to N and are powers of two.

Let us first define some terms that we will need in the design. The density of a sub-array u, denoted by $d(u)$ is equal to the number of elements stored in the sub-array divided by the size of the sub-array. Associated with each sub-array is a density threshold. This thresholding is very similar to [115] except that [32] also uses a lower bound threshold. So this means that $d(u)$ for each sub-array is bounded between an interval depending on the

height of the node u in the tree. We will describe the data structure as a conceptual tree on the array A. Let the array A be split into $\log|A|$ size sub arrays. These sub arrays will form the leaves of a tree on whose nodes we will maintain density thresholds. Let the root node be at height zero, then leaf nodes are at height $\log\log|A|$. Each node maintains a upper and lower bound thresholds on the density. As soon as a node is discovered whose density $d(u)$ violates the thresholds, it is "fixed" so that the density lies again between the thresholds. The bounds for node densities $d(u)$ are (τ_k, ρ_k) where k is the height of the node u. τ_k and ρ_k are arithmetic progressions, $\tau_k = \tau_0 + k\delta$ and $\rho_k = \rho_0 - k\delta'$ where

$$0 < \rho_{\log\log N} < \rho_0 < \tau_0 < \tau_{\log\log N} = 1$$

$$\delta = \frac{\tau_{\log\log N} - \tau_0}{\log\log N}$$

$$\delta' = \frac{\rho_{\log\log N} - \rho_0}{\log\log N}$$

We will only describe insertions here, deletions are very similar and in fact easier than insertions. When a element x needs to be inserted, first the leaf that needs to contain it is located. If inserting x in the leaf does not interfere with the density threshold upper and lower bounds we are done. If not, then we walk up the tress from the leaf and locate the first node whose density threshold bounds are not violated. At this point, this node is *rebalanced*. Re-balancing means redistributing all the elements in the sub-array corresponding to the node, evenly in the entire space available to the node. This also makes sure that the densities of the children of the rebalanced node respect their upper and lower bounds (Because thresholds become more relaxed down the tree).

We will now see how amortized analysis will help us prove that each insertion can be done in $O(\log^2 N)$ amortized time and $O\left(1 + \frac{\log^2 N}{B}\right)$ amortized memory transfers. Suppose node u has depth l and was rebalanced, i.e. $\rho_l \leq d(u) \leq \tau_l$. Some child of u, say

v has violated its bounds for the density thresholds. Note that $\rho_l \leq d(v) \leq \tau_l$. For u to overflow again, we need at least $size(u)(\tau_{l+1} - \tau l)$ inserts, where $size(u)$ denotes the total number of elements that can be stored in node u. The number of elements touched while re-balancing u is $size(u)$. Thus the average work done per insertion in the node v is

$$\frac{size(u)}{size(v)(\tau_{l+1} - \tau_l)} = \frac{2}{(\tau_{l+1} - \tau_l)} = \frac{2\log\log N}{\tau_{\log\log N} - \tau_0} = O(|A|) = O(\log N)$$

This shows the amortized number of array positions touched per insertion of an element x into each node u of the tree. Each time we insert x into A, we insert it into $\log|A|$ different nodes, hence total number of array positions touched is $O(\log^2 N)$. Each time we perform an insertion, we only scan A to the left and to the right, to decide how big a sub-array we should re-balance (Note that this means we always touch contiguous memory locations). Thus the amortized number of memory locations touched is $O\left(1 + \frac{\log^2 N}{B}\right)$. A similar analysis shows that in the same time bound, we could do deletions.

5.3 Matrix transposition

Matrix transposition is a fundamental operation in linear algebra and in fast Fourier transforms and has applications in numerical analysis, image processing and graphics. The simplest hack for transposing a $N \times N$ square matrix in C++ could be:

```
for (i = 0; i < N; i++)
   for (j = i+1; j < N; j++)
       swap(A[i][j], A[j][i])
```

In C++ matrices are stored in "row-major" storage, i.e. the rightmost dimension varies the fastest. In the above case, the number of cache misses the code could do is $O(N^2)$.

The optimal cache oblivious matrix transposition makes $O\left(1+\frac{N^2}{B}\right)$ cache misses. Before we go into the divide and conquer based algorithm for matrix transposition that is cache oblivious, let us see some experimental results (see figure 36, 37, 38, 40 , 41 and 42).

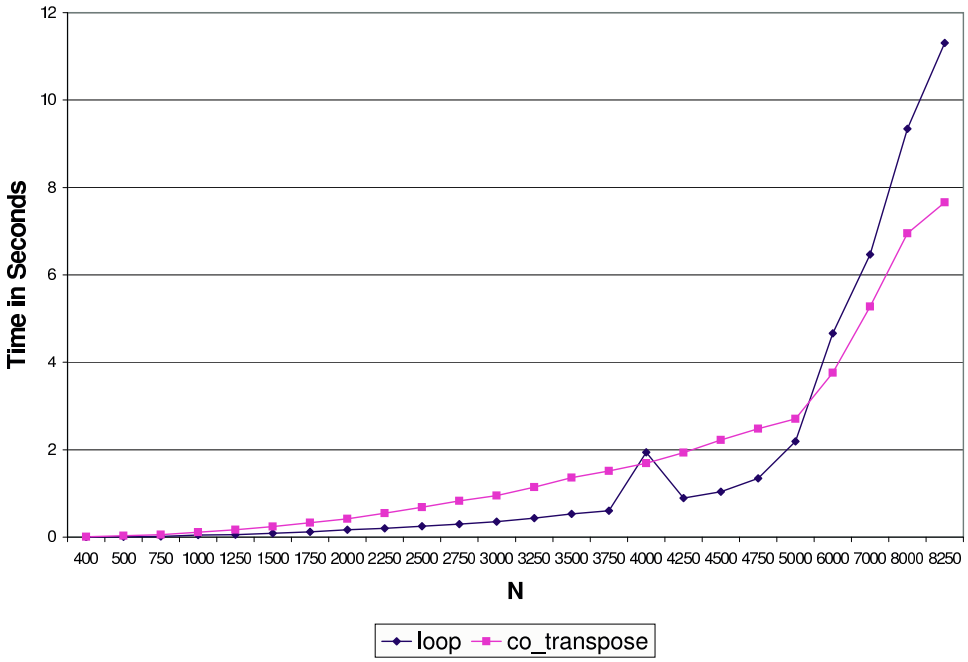


Figure 36: Experiments with simple for loop and a cache oblivious matrix transposition subroutine on a Windows NT running on 1GHz/512MB RAM notebook, compiled with g++.

Figure 39 shows the C/C++ code for cache oblivious matrix transposition. The code takes as input a sub-matrix given by $(x,y)-(x+delx,y+dely)$ in the input matrix I and

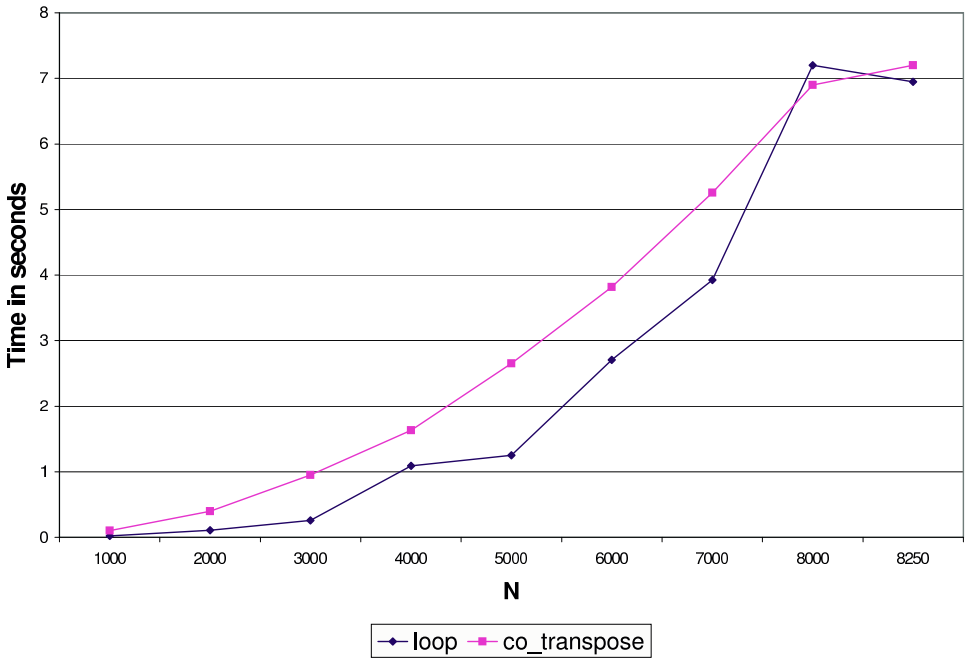


Figure 37: Same experiment as in figure 36 but now compiler has `-O3` flag set.

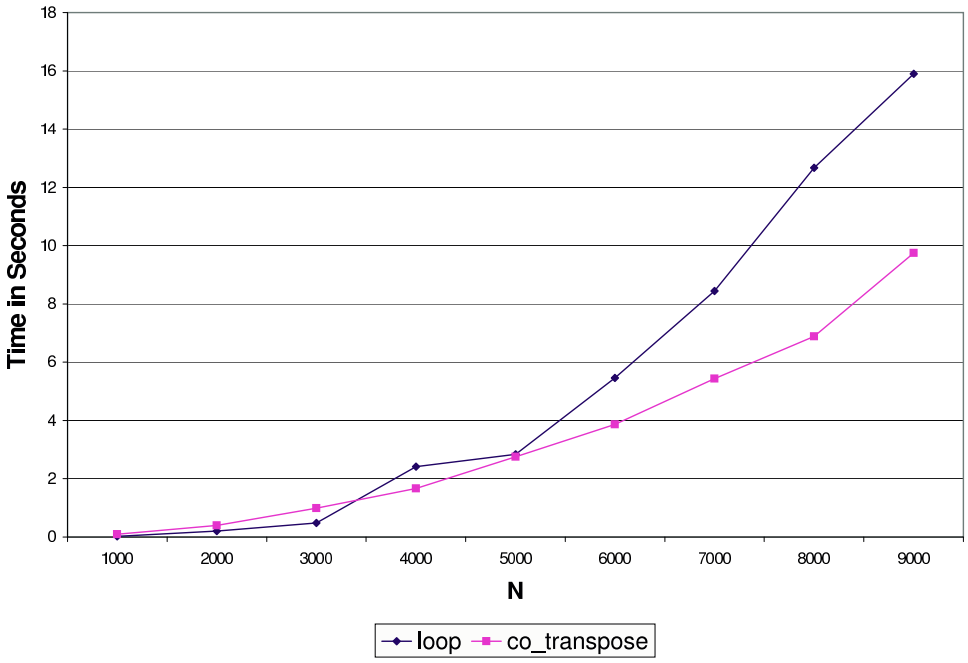


Figure 38: Same experiment as in figure 36 but now on a Linux machine, with Athlon 1GHz processor and 512MB RAM.

```
void transpose(int x, int delx, int y, int dely,
               ElementType I[N][P], ElementType O[P][N]){
        // Base Case of recursion
        // Should be tailored for specific machines if one wants
        // this code to perform better.
        if((delx == 1) && (dely == 1)) {
                O[y][x] = I[x][y];
                return;
        }
        // Divide the transposition into two sub transposition
        // problems, depending upon which side of the matrix is
        // bigger.
        if(delx >= dely){
                int xmid = delx / 2;
                transpose(x,xmid,y,dely,I,O);
                transpose(x+xmid,delx-xmid,y,dely,I,O);
                return;
        }
        // Similarly cut from ymid into two subproblems
        ...
}
```

Figure 39: Function for cache oblivious matrix transposition.

transposes it to the output matrix O. `ElementType`[1] can be any element type , for instance `long`.

The code works by divide and conquer, dividing the bigger side of the matrix in the middle and recursing. We also compared matrix transposition code for a general $N \times P$ matrix (using two for loops) with the cache oblivious code (see figure 40 and 41).

Lemma 5.3.1 *For an $N \times N$ input matrix, the above code causes at most $O\left(1+\frac{N^2}{B}\right)$ cache misses.*

[1] In all our experiments, `ElementType` was set to `long`.

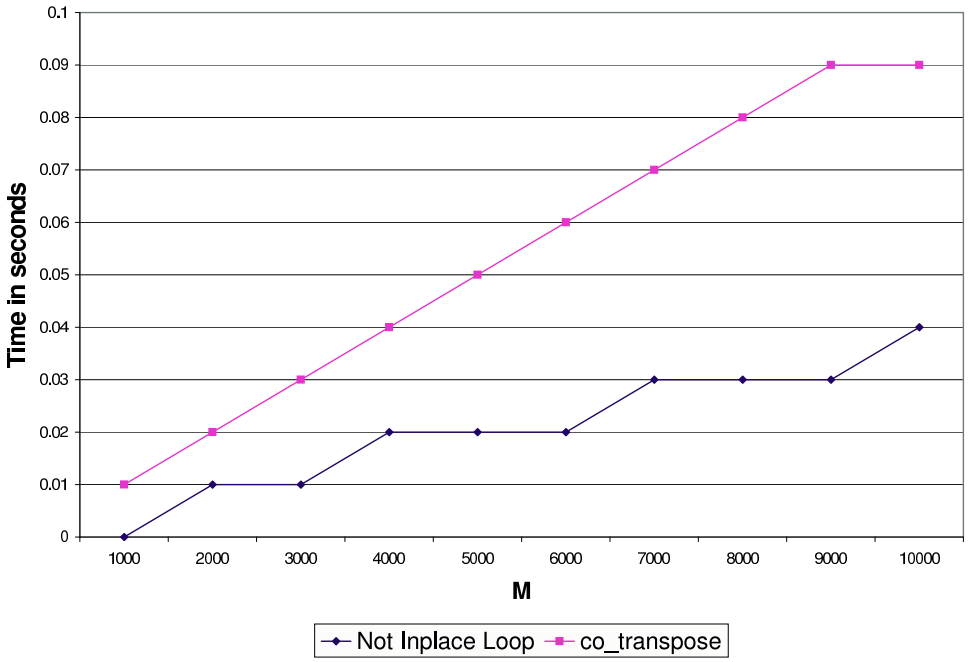


Figure 40: A transpose for an $N \times P$ matrix where $P = 100$. Experiments on Linux machine as in figure 38.

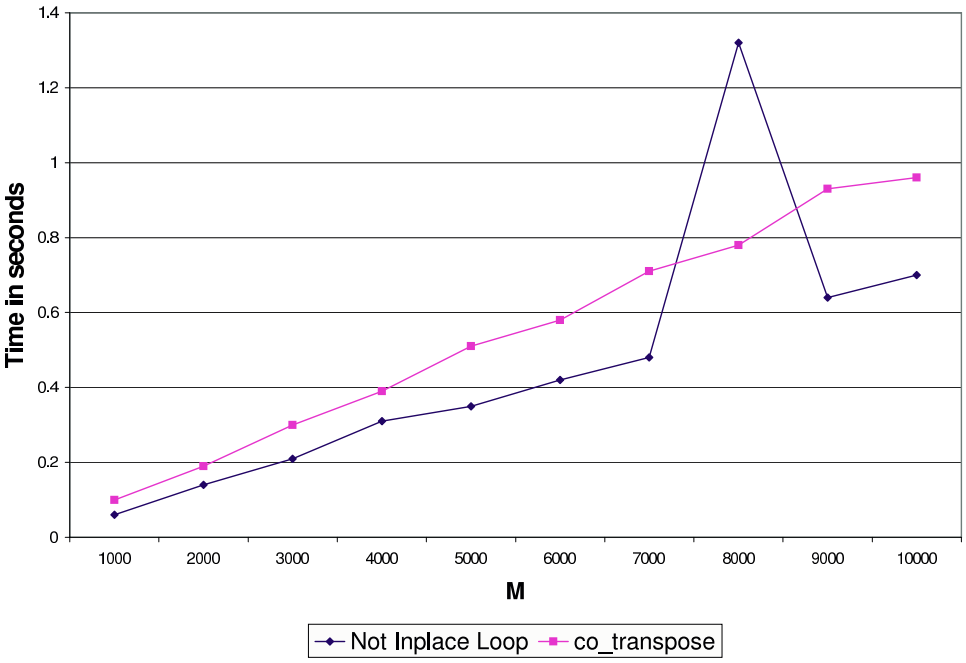


Figure 41: Same experiment as in figure 40 but now $P = 1000$.

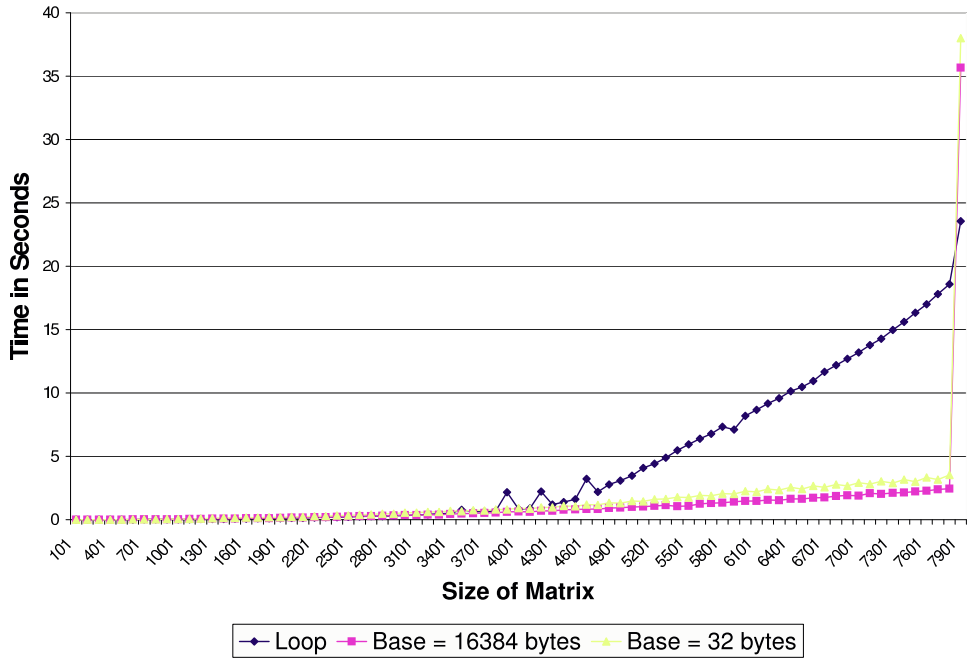


Figure 42: The graph compares a simple for loop implementation with a blocked cache oblivious implementation. In the blocked cache oblivious implementation, we stop the recursion when the problem size becomes less than a certain block size and then use the simple for loop implementation inside the block. Note that using different block sizes has little effect on the running time. This experiment was done on the cygwin platform used earlier.

Proof. Let the input be a matrix of $N \times P$ size. There are four cases:

Case I $max\{N,P\} \leq \alpha B$ In this case,

$$Q(N,P) \leq \frac{NP}{B} + O(1)$$

Case II $N \leq \alpha B < P$ In this case,

$$Q(N,P) \leq \begin{cases} O(1+N) & \text{if } \frac{\alpha B}{2} \leq P \leq \alpha B \\ 2Q(N,P/2) + O(1) & N \leq \alpha B < P \end{cases} \tag{61}$$

Case III $P \leq \alpha B < N$ Analogous to Case III.

Case IV $min\{N,P\} \geq \alpha B$

$$Q(N,P) \leq \begin{cases} O\left(N+P+\frac{NP}{B}\right) & \text{if } \frac{\alpha B}{2} \leq N,P \leq \alpha B \\ 2Q(N,P/2) + O(1) & P \geq N \\ 2Q(N/2,P) + O(1) & N \geq P \end{cases} \tag{62}$$

The above recurrence solves to $Q(N,P) = O\left(1+\frac{NP}{B}\right)$. □

There is a simpler way to visualize the above mess. Once the recursion makes the matrix small enough such that $max(N,P) \leq \alpha B \leq \beta\sqrt{M}$ (here β is a suitable constant), or such that the sub-matrix (or the block) we need to transpose fits in memory, the number of I/O faults is equal to the scan of the elements in the sub-matrix. A packing argument of these not so small sub-matrices (blocks) in the large input matrix shows that we do not do too many I/O faults compared to a linear scan of all the elements.

Remark: Note that the timings of this algorithm are not so impressive except figure 42. The algorithm given here is not the best for matrix transposition. If the reader is interested

in practicality of matrix transposition, excellent references are [53, 189]. Also note that if one expands the base case (instead of moving single elements around, one moves multiple elements) the performance improves. Changing the environment (OS, Processor etc.) has a significant impact on performance of cache oblivious algorithms (implemented without any kind of blocking). The experimental results reported here do not match the results of [53]. Their implementation of cache oblivious matrix transposition always ran faster than their naive implementation, which was not the case for us. This could occur because of more than one reasons, one of them being use of loop unrolling directives to the compiler while optimization, function call overheads because we recurse down to the base case of size 1, whereas one could stop the recursion at some other constant size to speed the code up.

Figure 42 shows the effect of using blocked cache oblivious algorithm for matrix transposition. Note that in this case, the simple for loop algorithm is almost always outperformed. This comparison is not really fair. The cache oblivious algorithm gets to use blocking whereas the naive for loop moves one element at a time. A careful implementation of a blocked version of the simple for loop might beat the blocked cache oblivious transposition algorithm in practice (see the timings of Algorithm 2 and 5 in [53]). The same remark also applies to matrix multiplication (figure 43).

5.4 Matrix multiplication

Matrix multiplication is one of the most studied computational problems: We are given two matrices of $m \times n$ and $n \times p$ and we want to compute the product matrix of $m \times p$ size. In this section we will use $n = m = N$ although the results can easily be extended to the case when they are not equal. Thus, we are given two $N \times N$ matrices $x = (x_{i,j})$, $y = (y_{i,j})$, and

we wish to compute their product z, i.e. there are N^2 outputs where the (i,j)'th output is

$$z_{i,j} = \sum_{k=1}^{N} x_{i,k} \cdot y_{k,j}$$

In 1969, Strassen surprised the world by showing an upper bound of $O(N^{\log_2 7})$ [174] using a divide and conquer algorithm. This bound was later improved and the best upper bound today is $O(N^{2.376})$ [66]. We will use Strassen's algorithm as given in [67] for the cache oblivious analysis.

The algorithm breaks the three matrices x,y,z into four sub-matrices of size $\frac{N}{2} \times \frac{N}{2}$, rewriting the equation $z = xy$ as :

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & c \\ b & d \end{pmatrix} \times \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

Now we have 8 subproblems (matrix multiplications) of size $\frac{N}{2} \times \frac{N}{2}$ and four additions (which will lead us to a $O(N^3)$ algorithm). This can be done more cleverly by only solving 7 subproblems and doing more additions and subtractions but still keeping the number of additions and subtraction a constant (see [67] for more details). The recurrence for the internal work now becomes :

$$T(N) = 7T\left(\frac{N}{2}\right) + \Theta(N^2)$$

which solves to $T(N) = O\left(N^{\lg 7}\right) = O(N^{2.81})$. The recurrence for the cache faults is:

$$Q(N) \leq \begin{cases} O\left(1 + N + \frac{N^2}{B}\right) & \text{if } N^2 \leq \alpha M \\ 7Q\left(\frac{N}{2}\right) + O\left(1 + \frac{N^2}{B}\right) & \text{otherwise.} \end{cases} \tag{63}$$

which solves to $Q(N) \leq O\left(N + \frac{N^2}{B} + \frac{N^{\lg 7}}{B\sqrt{M}}\right)$. We also implemented a blocked cache oblivious matrix transposition routine that works very similar to the transposition routine in the previous section. It breaks the largest of the three dimensions of the matrices to be multiplied (divides it by 2) and recurses till the block size is reached. Note that this algorithm is not optimal and does $O\left(1 + \frac{N^2}{B} + \frac{N^3}{B\sqrt{M}}\right)$ cache misses.

We also present here experiments of a simpler divide and conquer matrix multiplication algorithm. The worst case of the implementation is $O\left(N + N^2 + \frac{N^3}{B\sqrt{M}}\right)$. The experimental results are shown in figure 43. An excellent reference for practical matrix multiplication results is [87].

5.5 Searching using Van Emde Boas layout

In this section we report a method to speed up simple binary searches on a balanced binary tree. This method could be used to optimize or speed up any kind of search on a tree as long as the tree is static and balanced. It is easy to code, uses the fact that the memory consists of a cache hierarchy, and could be exploited to speed up tree based search structures on most current machines. Experimental results show that this method could speed up searches by a factor of 5 or more in certain cases!

It turns out that a balanced binary tree has a very simple layout that is cache-oblivious. By layout here, we mean the mapping of the nodes of a binary tree to the indices of an array where the nodes are actually stored. For example figure 44 shows a layout of the tree in the figure. The nodes should be stored in the bottom array in the order shown for searches to be fast and use the cache hierarchy.

Given a complete binary tree, we describe a mapping from the nodes of the tree to positions of an array in memory. Suppose the tree has N items and has height $h = \log N + 1$.

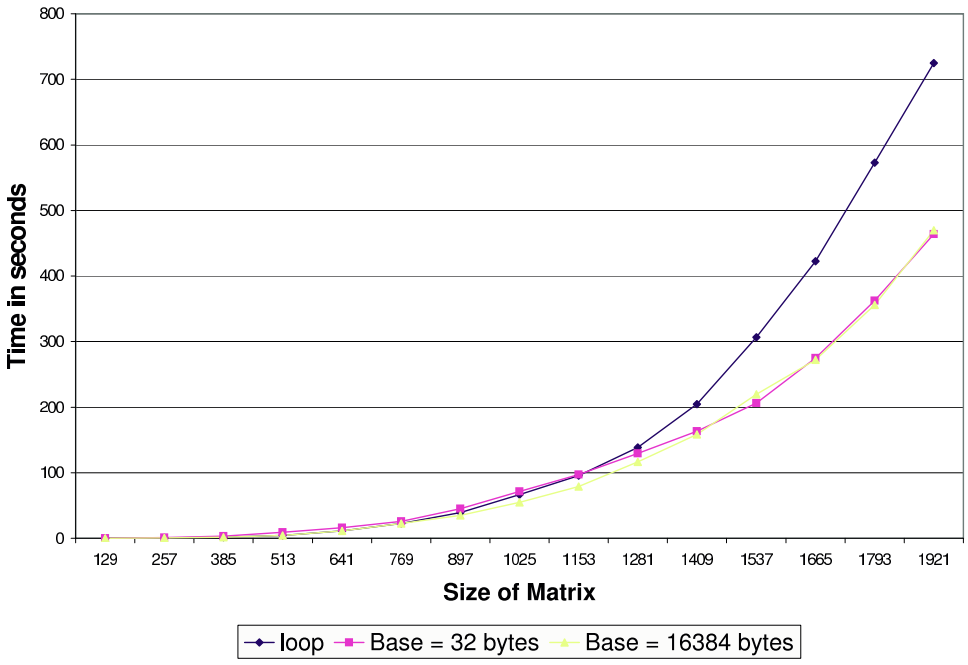
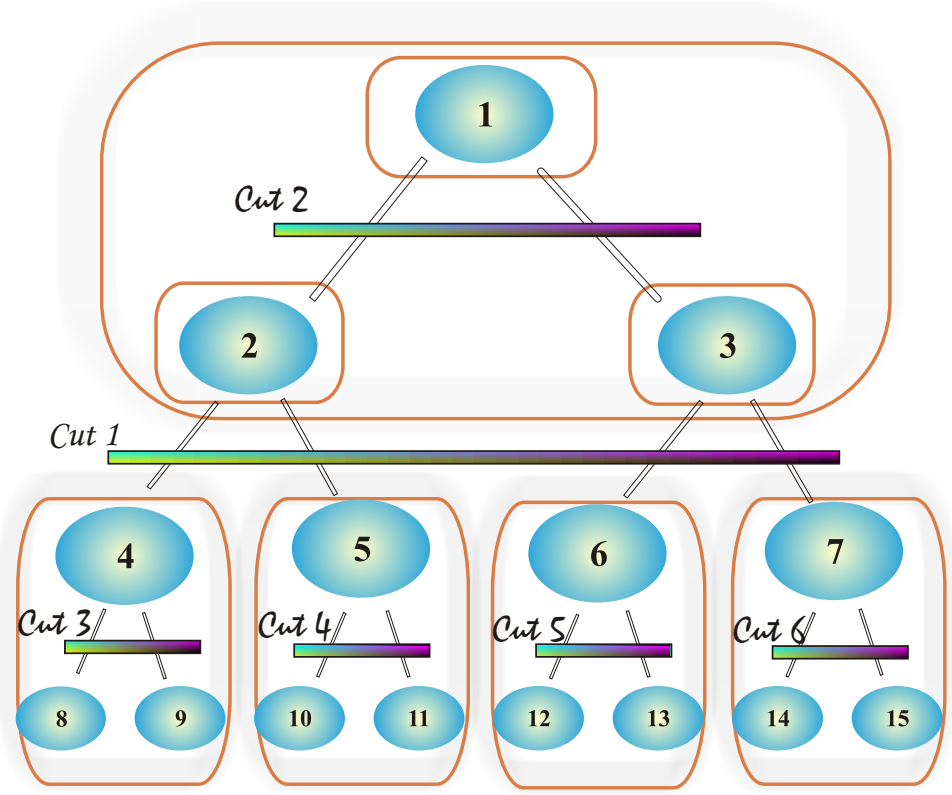


Figure 43: Blocked cache oblivious matrix multiplication compared with simple for loop based matrix multiplication. This experiment was done on an dual processor Intel Itanium with 2Gb RAM. Only one processor was being used. Note that on this platform, the blocked implementation consistently outperforms the simple loop code. The base case has little effect for large size matrices.



ACTUAL LAYOUT OF TREE IN MEMORY:

1, 2, 3, 4, 8, 9, 5, 10, 11, 6, 12, 13, 7, 14, 15

Figure 44: An example of the Van Emde Boas layout using a balanced binary tree.

Split the tree in the middle, at height $h/2$. This breaks the tree into a top recursive subtree of height $\lfloor h/2 \rfloor$ and several bottom subtrees $B_1, B_2, ..., B_k$ of height $\lceil h/2 \rceil$. There are $\sqrt{N}$ bottom recursive subtrees, each of size $\sqrt{N}$. The top subtree occupies the top part in the array of allocated nodes, and then the B_i's are laid out. Every subtree is recursively laid out.

Another way to see the algorithm is to run a breadth first search on the top node of the tree and run it till $\sqrt{N}$ nodes are in the BFS, see figure 45. The figure shows the run of the algorithm for the first BFS when the tree size is $\sqrt{N}$. Then the tree consists of the part that is covered by the BFS and trees hanging out. BFS can now be recursively run on each tree, including the covered part. Note that in the second level of recursion, the tree size is $\sqrt{N}$ and the BFS will cover only $N^{\frac{1}{4}}$ nodes since the same algorithm is run on each subtree of $\sqrt{N}$. The main idea behind the algorithm is to store recursive sub-trees in contiguous blocks of memory.

Lets now try to analyze the number of cache misses when a search is performed. We can conceptually stop the recursion at the level of detail where the size of the subtrees has size $\leq B$. Since these subtrees are stored contiguously, they at most fit in two blocks. (A block can not span three blocks of memory when stored) The height of these subtrees is $\log B$. A search path from root to leaf crosses $O\left(\frac{\log N}{\log B}\right) = O(\log_B N)$ subtrees. So the total number of cache misses is bounded by $O(\log_B N)$. It is not very hard to show that the Van Emde Boas layout can be at most a constant factor of 4 away from an optimal layout (which knows the parameter B).

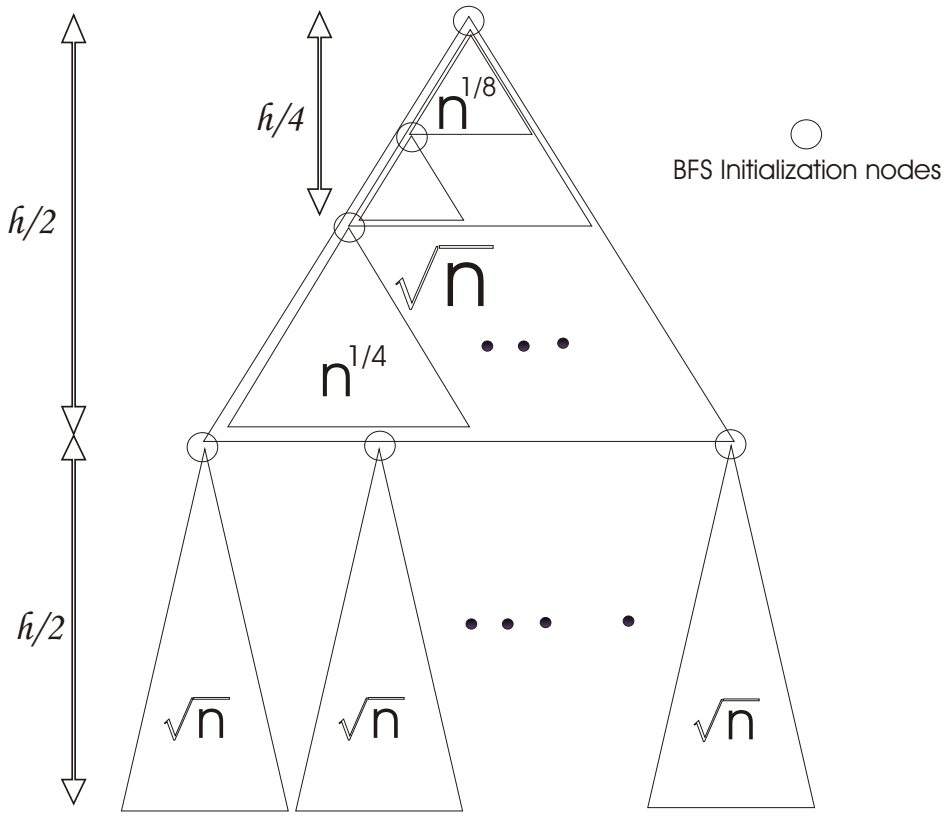


Figure 45: Another view of the algorithm in action.

5.5.1 Experiments

We did a very simple experiment to see how in real life, this kind of layout would help. A vector was sorted and a binary search tree was built on it. A query vector was generated with random numbers and searched on this BST which was laid out in pre-order. Why we chose pre-order compared to random layout was because most people code a BST in either pre/post/in-order compared to randomly laying it which incidentally is very bad for cache health.

Once this query was done, we laid the BST using Van Emde Boas Layout and gave it the same query vector. Before timing both trees, we made sure that both had enough queries to begin with otherwise, the order of timing could also effect the search times. (Because the one that is searched last, has some help from the cache). The code written for this experimentation is below 300 lines. The results reported in figure 46 and 47 used cygwin[2] g++ version 2.95 with compiler optimization `-O3` and was running on our laptop with 1GHz processor, 512MB RAM and Windows 2000 as the Operating System. The experiment reported in figure 48 were done on a Itanium dual processor system with 2Gb RAM. (Only one processor was being used)

Currently the code copies the entire array in which the tree exists into another array when it makes the tree cache oblivious. This could also be done in the same array though that would have complicated the implementation a bit. One way to do this is to maintain pointers to and from the array of nodes to the tree structure, and swap nodes instead of copying them into a new array. Another way could be to use a permutation table. We chose to copy the whole tree into a new array just because this seemed to be the simplest

[2]Cygwin is a linux like environment which runs on windows. It's available from `http://www.cygwin.org`

way to test the speed up given by cache oblivious layouts. For more detailed experimental results on comparing searching in cache aware and cache oblivious search trees, the reader is referred to [133]. There is a big difference between the graphs reported here for searching and in [133]. One of the reasons might be that the size of the nodes were fixed to be 4 bytes in [133] whereas the experiments reported here use bigger size nodes.

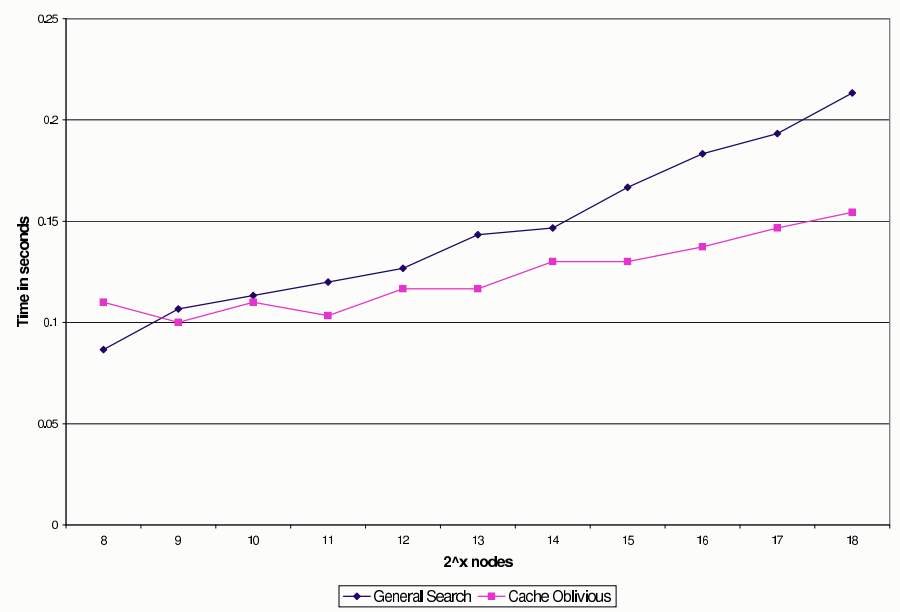


Figure 46: 32 byte tree nodes. All Timings are from internal memory. These experiments were done on our PIII notebook.

5.6 Sorting

Sorting is a fundamental problem in computing. Sorting very large data sets is a key routine in many external memory applications. We have already seen how to do optimal sorting in

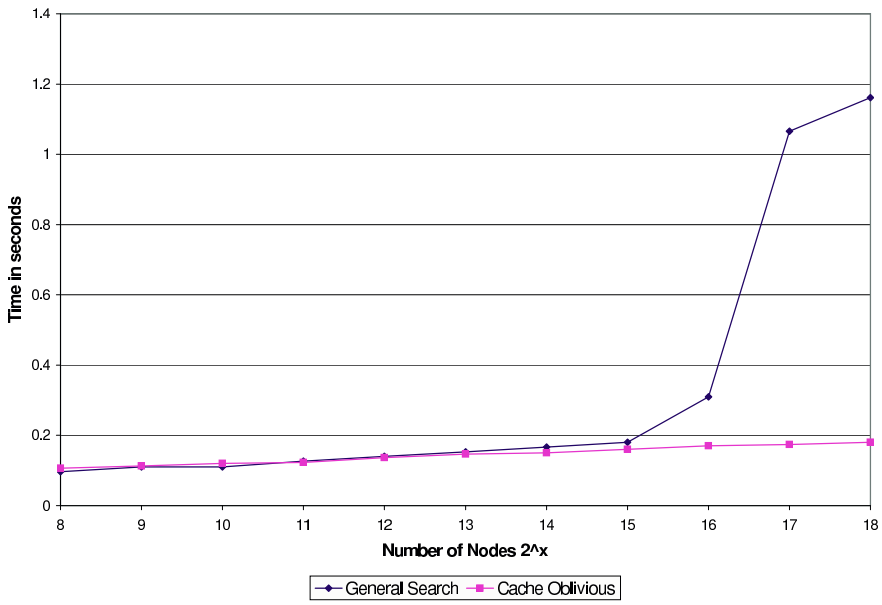


Figure 47: 256 byte tree nodes. All Timings are from internal memory. For bigger node sizes, and trees that do not fit into internal memory, we expect the speed up to be more than internal memory. These experiments were done on our PIII notebook.

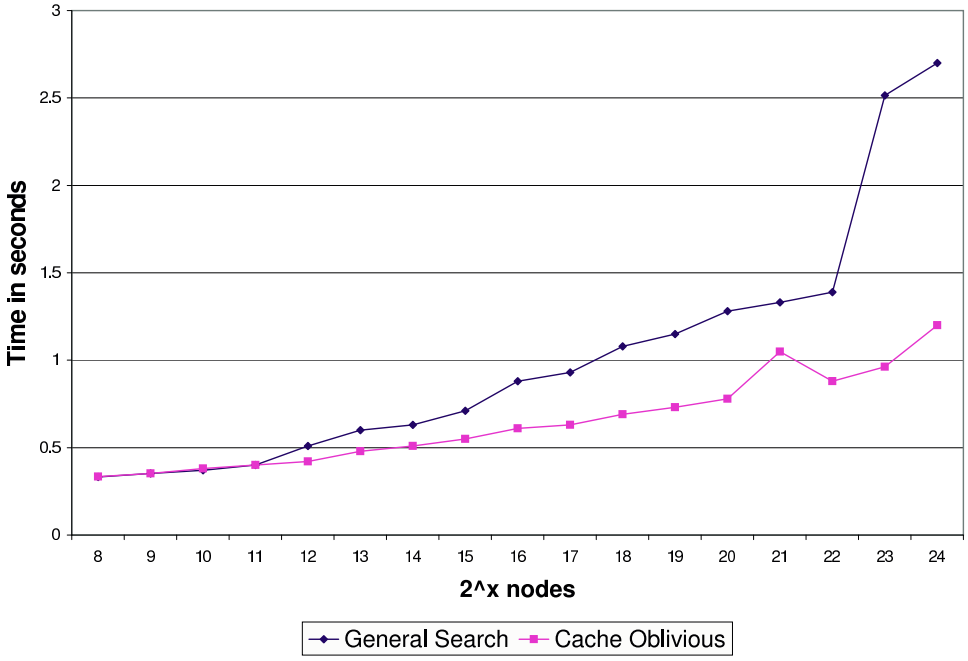


Figure 48: Similar to the last experiment, this experiment was performed on a Itanium with 48 byte node sizes.

the external memory model. In this section we outline some theory and experimentation related to sorting in the cache oblivious model. Some excellent references for reading more on the influence of caches on the performance of sorting are [134, 150].

5.6.1 Randomized distribution sorting

There are two optimal sorting algorithms known, funnel sort and distribution sort. Funnel sort is derived from merge sort and distribution sort is a generalization of quick sort. We will see the analysis and implementation of a variant of distribution sort that is optimal and easier to implement than the original distribution sort of [97] (The original algorithm uses median finding and is deterministic). The algorithm for randomized distribution sort is presented in Algorithm 8.

Algorithm 8 procedure `Sort`(A)

Require: An input array A of Size N

1: Partition A into $\sqrt{N}$ sub arrays of size $\sqrt{N}$. Recursively sort each sub-array.
2: $R \leftarrow$ `Sample`$(A, \sqrt{N})$
3: Sort(R) recursively. $R = \{r_0, r_1, ..., r_{\sqrt{N}}\}$.
4: Calculate counts c_i such that $c_i = |\{x \mid x \in A \text{ and } r_i \leq x < r_{i+1}\}|$
5: $c_{1+\sqrt{N}} = |A| - \Sigma_i c_i$.
6: Distribute A into buckets $B_0, B_1, ...B_{1+\sqrt{N}}$ where last element of each bucket is r_i except the last bucket. For the last bucket, the last element is maximum element of A. Note that $|B_i| = c_i$.
7: Recursively sort each B_i
8: Copy sorted buckets back to A.

The sorting procedure assumes the presence of three extra functions which are cache oblivious, choosing a random sample in Step 2, the counting in Step 4 and the distribution in Step 6. The random sampling step in the algorithm is used to determine splitters for the buckets, so we are in fact looking for splitters.

The function `Sample`(X,k) in step 2 takes as input an array X and the size of the

sample that it needs to return as output. A very similar sampling is needed in Sample Sort [37] to select splitters. We will use the same algorithm here. What we need is a random sample such that the c_i's do not vary too much from $\sqrt{N}$ with high probability. To do this, draw a random sample of $\beta\sqrt{N}$ from A, sort the sample and then pick each βth element from the sorted sample. In [37], β is referred to as the *oversampling ratio*, the more the β the greater the probability that c_i's concentrate around $\sqrt{N}$.

The distribution and counting phases are quite analogous. Here is how we do distribution:

Algorithm 9 procedure `Distribute`(`int` i, `int` j, `int` m)

```
1: if m = 1 then
2:    CopyElements(i, j)
3: else
4:    Distribute(i, j, m/2)
5:    Distribute(i + m/2, j, m/2)
6:    Distribute(i, j + m/2, m/2)
7:    Distribute(i + m/2, j + m/2, m/2)
8: end if
```

In coding one would have to also take care if m is odd or even and appropriately apply floor ceiling operations for the above code to work. In our code, we treated it by cases, when m was odd and when m was even, the rounding was different. Anyway, this is a minor detail and can be easily figured out. In the base case `CopyElements`(i, j) copies all elements from sub-array i that belong to bucket j. If the function `CopyElements`(i, j) is replaced by `CountElements`(i, j) in `Distibute`$()$, the counting that is needed in Step 4 of Algorithm 8 can be performed. `CountElements`$()$ increments c_j's depending upon how many elements of sub-array i lie between r_j and r_{j+1}. Since these sub-arrays are sorted, this only requires a linear scan.

Before we goto the analysis, lets see how the implementation works in practice. We

report here two experiments where we compare C++ STL sort with our implementation. In our implementation we found out that as we recurse more, the performance of our sorting algorithm deteriorates. In the first graph in figure 49, Series 1 is the STL Sort and Series 2 is our sorting implementation. The recursion is only 1 level, i.e. after the problem size becomes $\sqrt{N}$ STL Sort is used. In figure 50 two levels of recursion are used. Note that doing more levels of recursion degrades performance. At least for 1 level of recursion, the randomized distribution sort is quite competitive and since these experiments are all in-memory tests, it might be possible to beat STL Sort when the data size does not fit in memory but we could not experiment with such large data sizes.

Lets now peek at the analysis. For a good random sample, $Pr(c_i \geq \alpha\sqrt{N}) \leq Ne^{-(1-\frac{1}{\alpha})^2\frac{\beta\alpha}{2}}$ (for some $\alpha > 1, \beta > \log N$) [37]. This follows from Chernoff bound type arguments. Once we know that the subproblems we are going to work on are bounded in size with high probability, the expected cache complexity follows the recurrence:

$$Q(N) \leq \begin{cases} O\left(1+\frac{N}{B}\right) & \text{if } N \leq \alpha M \\ 2\sqrt{N}Q(\sqrt{N}) + Q\left(\sqrt{N}\log N\right) + O\left(1+\frac{N}{B}\right) & \text{otherwise.} \end{cases} \tag{64}$$

which solves to $Q(n) \leq O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$.

Columnsort is a sorting algorithm that has been shown to be practical for large inputs [54]. The cache complexity of columnsort can be defined by the following recurrence.

$$Q(n) = \begin{cases} O\left(1+\frac{N}{B}\right) & \text{if } N \leq \alpha M \\ 8N^{\frac{1}{4}}Q\left(N^{\frac{3}{4}}\right) + O\left(1+\frac{N}{B}\right) & \text{otherwise.} \end{cases} \tag{65}$$

where α is a sufficiently small constant. Note that the recurrence above is very similar

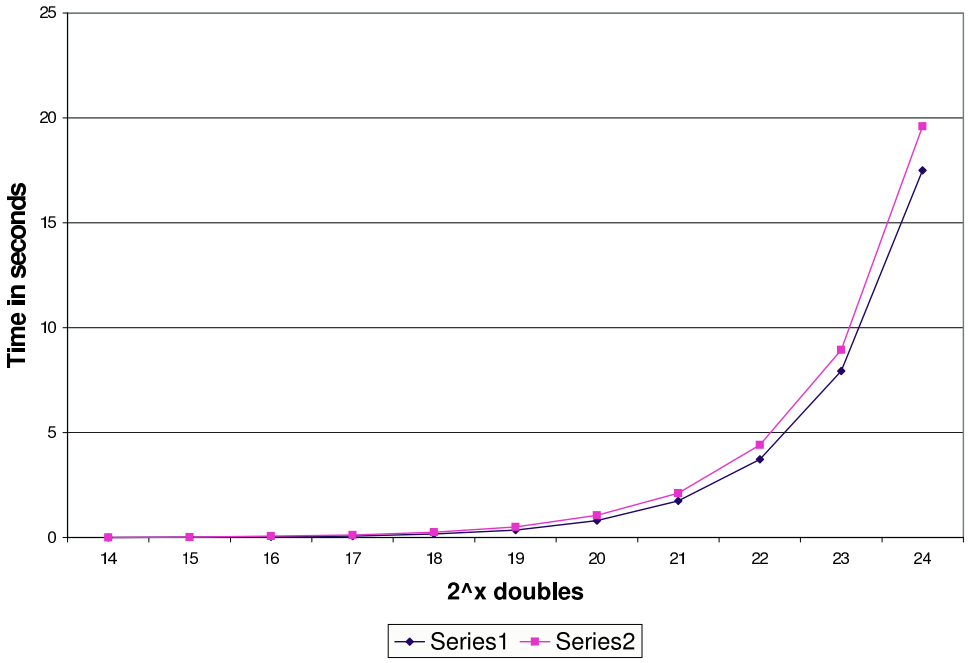


Figure 49: Two Pass cache oblivious distribution sort, one level of recursion.

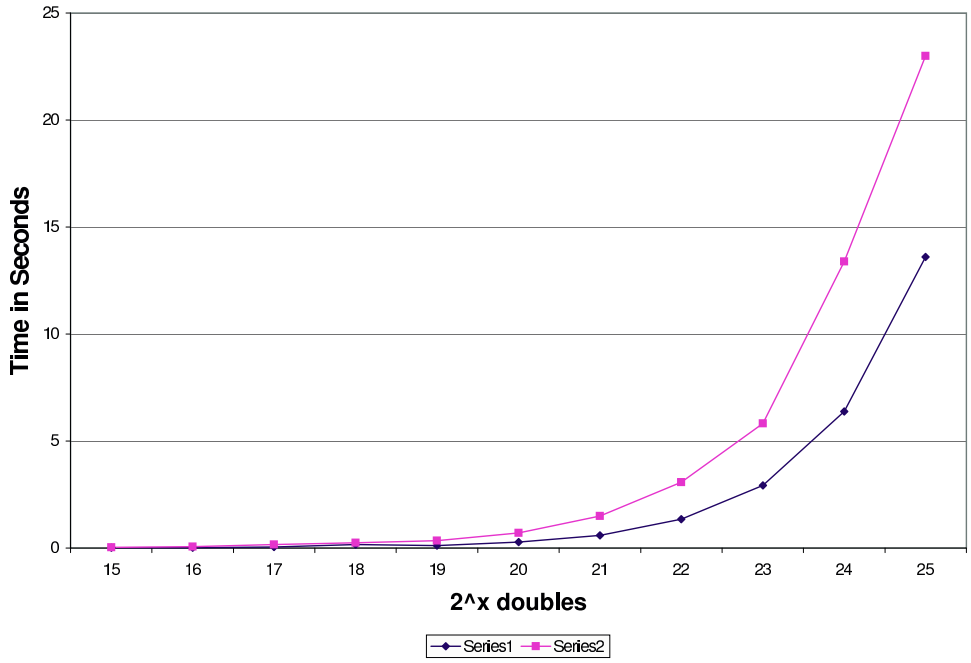


Figure 50: Two Pass cache oblivious distribution sort, two levels of recursion.

to the recurrence of cache oblivious distribution sort [97] except for the fact that the constant is 8 instead of 2. We note that, this recurrence does not solve to the optimal bound for sorting in the cache oblivious model.

5.7 Is the model an oversimplification?

In theory, both the cache oblivious and the external memory models are nice to work with, because of their simplicity. Lot of the work done in the external memory model has been turned into practical results as well. Before one makes his hand "dirty" with implementing an algorithm in the cache oblivious or the external memory model, one should be aware of practical things that might become detrimental to the speed of the code but are not caught in the theoretical setup.

Here we list a few practical glitches that are shared by both the cache oblivious and the external memory model. The ones that are not shared are marked[3] accordingly. A reader that wants to use these models to design practical algorithms and especially one who wants to write code, should keep these issues in mind. Code written and algorithms designed keeping the following things in mind, could be a lot faster than just directly coding an algorithm that is optimal in either the cache oblivious or the external memory model.

TLB^o TLBs are caches on page tables, are usually small with 128-256 entries and are like just any other cache. They can be implemented as fully associative. The model does not take into account TLB misses.

Concurrency The model does not talk about I/O and CPU concurrency, which automatically looses it a 2x factor in terms of constants. The need for speed might drive

[3] A superscript 'o' means this issue only applies to the cache oblivious model.

future *uniprocessor* systems to diversify and look for alternative solutions in terms of concurrency on a single chip, for instance the hyper-threading[4] introduced by Intel in its latest Xeons is a glaring example. On these kind of systems and other multiprocessor systems, *coherence misses* might become an issue. This is hard to capture in the cache oblivious model and for most algorithms that have been devised in this model already, concurrency is still an open problem. A parallel cache oblivious model would be really welcome for practitioners who would like to apply cache oblivious algorithms to multiprocessor systems.

Associativity^o The assumption of the fully associative cache is not so nice. In reality caches are either direct mapped or k-way associative (typically $k = 2, 4, 8$). If two objects map to the same location in the cache and are referenced in temporal proximity, the accesses will become costlier than they are assumed in the model (also known as cache interference problem [179]). Also, $k-$way set associative caches are implemented by using more comparators.

Instruction/Unified Caches Does not deal with the issue of instruction caches. Rarely executed, special case code disrupts locality. Loops with few iterations that call other routines make loop locality hard to exploit and plenty of loopless code hampers temporal locality. Issues related to instruction caches are not modeled in the cache oblivious model. *Unified caches* (e.g. the latest Intel Itanium chips L2 and L3 caches) are used in some machines where instruction and data caches are merged(e.g. Intel PIII, Itaniums). These are another challenge to handle in the model.

[4]One physical processor Intel Xeon MP forms two logical processors which share CPU computational resources The software sees two CPUs and can distribute work load between them as a normal dual processor system.

Replacement Policy^o Current operating systems do not page more than 4Gb of memory because of address space limitations. That means one would have to use legacy code on these systems for paging. This problem makes portability of cache oblivious code for big problems a myth! In the experiments reported in this chapter, we could not do external memory experimentation because the OS did not allow us to allocate array sizes of more than a GB or so. One can overcome this problem by writing one's own paging system over the OS to do experimentation of cache oblivious algorithms on huge data sizes. But then its not so clear if writing a paging system is easier or handling disks explicitly in an application. This problem does not exist on 64-bit operating systems and should go away with time.

Multiple Disks^o For "most" applications where data is huge and external memory algorithms are required, using Multiple disks is an option to increase I/O efficiency. As of now, the cache oblivious model does not handle this case, though it might not be tough to introduce multiple disks in the model.

Write-through caches^o L1 caches in many new CPUs is write through, i.e. it transmits a written value to L2 cache immediately [94, 111]. Write through caches are simpler to manage and can always discard cache data without any bookkeeping (Read misses can not result in writes). With write through caches (e.g. DECStation 3100, Intel Itanium), one can no longer argue that there are no misses once the problem size fits into cache! *Victim Caches* implemented in HP and Alpha machines are caches that are implemented as small buffers to reduce the effect of conflicts in set-associative caches. There also should be kept in mind when designing code for these machines.

Complicated Algorithms^o **and Asymptotics** For non-trivial problems the algorithms can

> become quite complicated and impractical, a glaring instance of which is sorting. The speed by which different levels of memory differ in data transfer are constants! For instance the speed difference between L1 and L2 caches on a typical Intel pentium can be around 10. Using an $O()$ notation for an algorithm that is trying to beat a constant of 10, and sometimes not even talking about those constants while designing algorithms can show up in practice. For instance there are "constants" involved in simulating a fully associative cache on a k-way associative cache. Not using I/O concurrently with CPU can make an algorithm loose another constant. Can one really afford to hide these constants in the design of a cache oblivious algorithm in real code?

Despite these limitations the model does perform very well for some applications [53, 133, 95], but might be outperformed by more coding effort combined with cache aware algorithms [157, 53, 133, 95]. Here's an intercept from an experimental paper by Chatterjee and Sen [53].

> Our major conclusion are as follows: Limited associativity in the mapping from main memory addresses to cache sets can significantly degrade running time; the limited number of TLB entries can easily lead to thrashing; the fanciest optimal algorithms are not competitive on real machines even at fairly large problem sizes unless cache miss penalties are quite high; low level performance tuning "hacks", such as register tiling and array alignment, can significantly distort the effect of improved algorithms, ...

5.8 Other Results

We present here problems, related bounds and references for more interested readers. Note that in the table, $sort()$ and $scan()$ denote the number of cache misses of scan and sorting functions done by an optimal cache oblivious implementation.

Data Structure/Algorithm	Cache Complexity	Operations
Array Reversal	scan(N)	
List Ranking [60]	sort(N)	
LU Decomposition [184]	$\Theta\left(1+\frac{N^2}{B}+\frac{N^3}{B\sqrt{M}}\right)$	On $N \times N$ matrices
FFT [97]	sort(N)	
B-Trees [33, 42]	Amortized $O(\log_B N)$	Insertions/Deletions
Priority Queues [23, 41]	$O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$	Insertions/Deletions/deletemin

5.9 Future Work

There are a lot of problems that still can be explored in this model both theoretically and practically. Sorting strings in the cache oblivious model is still open. Optimal shortest paths and minimum spanning forests still need to be explored in the model. Optimal simple convex hull algorithms for $d-$dimensions is open.

One of the major open problems is to extensively evaluate how practical cache oblivious algorithms really are. For instance, our sorting experiment results don't look very promising. Is there a way to sort in the cache oblivious model that is at par with cache aware sorting codes already available? (Toy experiments comparing quicksort with a modified funnelsort or distribution sort don't count!) Currently the only impressive code that might back up "practicality" claims of cache oblivious algorithms is FFTW [95]. It looks like that the practicality of FFTW is more about coding tricks (special purpose compiler) that give

it its speed and practicality than the theory of cache obliviousness. Moreover, FFTW only uses cache oblivious FFT for its base cases (Called codelets).

Matrix multiplication and transposition using blocked cache oblivious algorithms do fairly well in comparison with cache aware/external memory algorithms. B-Trees also seem to do well in this model [33, 42]. Are there any other non-trivial optimal cache oblivious algorithms and data structures that are really practical and come close to cache aware algorithms, only time will tell!

For some applications, it might happen that there are more than one cache oblivious algorithms available. And out of the few cache oblivious algorithms available, only one is good in practice and the rest are not. For instance, for matrix transposition, there are at least two cache oblivious algorithms coded in [53]. There is one coded in this chapter. Out of these three algorithms only one really performs well in practice. In this way, cache obliviousness might be a necessary but not a sufficient condition for practical implementations. Another example is FFTW, where a optimal cache oblivious algorithm is outperformed by a slightly suboptimal cache oblivious algorithm. Is there a simple model that captures both necessary and sufficient condition for practical results?

CHAPTER VI

I/O-Efficient Voronoi diagrams

"Nothing is particularly hard if you divide it into small jobs."

HENRY FORD

In this chapter we consider the problems of computing 2- and 3-d Voronoi diagrams or Delaunay triangulations for large data sets efficiently. We describe a cache-oblivious distribution data structure (buffer tree) that is the basis for the cache oblivious implementation of a random incremental construction for geometric problems. We then apply this to the construction of 2- and 3-d Voronoi diagrams. We also describe a very simple variant of the standard random incremental construction based on history dag, which has optimal running time and is likely to be I/O-efficient because the pattern of insertions is also local (but we don't have theoretical bounds). Finally, we describe a practical variant that we implemented and present some experimental results.

Some applications – particularly surface reconstruction – require the computation of Delaunay tetrahedralizations (or their dual 3-d Voronoi diagrams) for up to several million points and more. For this data, even if the input can be stored in main memory, secondary storage will be needed for the computation and final output. Therefore, there is considerable interest in algorithms and actual implementations that are "I/O-efficient" or "cache

efficient." In this context, for the purpose of theoretical analysis, the most widely used model of computation [8] consists of a processing unit, an internal memory of size M and an (unbounded) external memory partitioned into blocks of size B, with $B \leq M$. Each access to the external memory transfers from/to the internal memory one block of B items. The goal is to design algorithms that take advantage of the block transfer and thus exhibit *locality* in the reference to data. The complexity of an algorithm is then evaluated in this model by providing bounds for the total number of disk accesses (I/Os) performed, and for the number of internal operations executed (CPU cost). The 2-d case had been considered previously in [106] and in [68] in the cache-aware context. Both provide solutions with an optimal number $O((N/B)\log_{M/B}(N/B))$ of I/Os, where N is the number of sites. The 3-d case is not explicitly handled in [106] nor [68]; however, the same approaches work (but the latter one has an extra log factor in the I/O cost: $O(N^2/B)\log_{M/B}(N/B))$ (since the size of a 3-d Voronoi diagram can be quadratic in the number of sites, $O(N^2/B)$ is worst-case optimal; aiming for this worst-case bound, rather than an "output sensitive" bound, makes the 3-d problem somewhat unrealistically easy). Until recently, external memory algorithms were *cache-aware*, that is, they made use of the parameters M and B. More recently, the concept of *cache-oblivious* algorithms was introduced [96]. It assumes a model also with two levels of memory that uses an optimal replacement strategy to decide which block is to be evicted from internal memory (that whose next access is furthest in the future). The algorithms then do not need to know the values of M and B. Efficient algorithms have been developed for this model (matrix transpose, FFT and sorting in [96]; B-trees [32]; dynamic dictionaries [33, 43]) including some geometric problems [40]. In this chapter, we investigate the cache-oblivious implementation of 2-d and 3-d Voronoi diagrams and also describe the implementation of a simplified version.

Our Results. For elements with a linear ordering, we describe a cache-oblivious data structure (*k-distributor* or buffer tree) of size $\Theta(k^2)$ that distributes $X = \Omega(k^3)$ elements into the k intervals (*buckets*) determined by k given elements (*k-splitter*) using $O((X/B)\log_M k + k)$ I/Os. We extend this construction to the geometric case in which the buckets are more general (for example, cells in an arrangement), and even further to the case of *conflict list computation*, in which each element goes to many buckets rather than to a single one (for example, a line is distributed to the different cells it intersects). Then, we present a cache-oblivious version of the I/O random incremental construction (RIC) presented in [68]. This unfortunately only leads to algorithms whose I/O cost is from optimal by a factor $\log_M N$. Though presented for Voronoi diagrams, the approach can be used for many other constructions where the usual RIC applies (in most cases with the same extra factor $\log_M N$). We also present a variant of the usual RIC with history graph that has optimal running time and is likely to be I/O-efficient because of the "clearly local" pattern in which the insertions are performed; unfortunately, we do not have theoretical bounds to support this claim, nor have yet performed experiments. To obtain optimal algorithms, we have to either use the divide-and-conquer approach (for both the 2-d and 3-d cases), or a modification of the RIC construction (3-d case). The 2-d case is more difficult and we have to resort to the same "pruning" approach used previously in various parallel algorithms (*e.g.* [161]) and in the cache aware algorithm in [106]. We thus obtain algorithms for constructing Voronoi diagrams in 2- and 3-d space using $O((N/B)\log_M N)$ and $O(N^2/B)$ (expected) I/Os. We have implemented a simplified version of the algorithm using a 2-level sampling.

Sampling: For concreteness we consider a set of N sites (in 2- or 3-d space) and for $R \subset S$ let $\mathrm{Vor}(R)$ be the Voronoi diagram of R. Let $\mathscr{T}(R)$ be a canonical decomposition of $\mathrm{Vor}(R)$. For example, a bottom vertex triangulation for lower dimensional cells and then join these

triangulations to the sites. A *p-sample* R from S is obtained by choosing every $s \in S$ into R independently with probability p. Alternatively, we say an *r-sample*, to mean a p-sample with $p = r/|S|$. Let $F(R)$ denote the expected size of $\mathscr{T}(R)$. The *conflict list* $S_{|\Delta}$ of $\Delta \in \mathscr{T}(R)$ is the set of those sites $s \in S$ that "conflict" with Δ, that is, s is closer to some point in Δ than the site in R that "owns" Δ. (In 2-d, it is convenient to join triangles that share the same Voronoi edge into a *diamond*, since they share the same conflict list; similarly in 3-d.) The main property of this sampling process that is relevant for the design and analysis of our algorithms is the following bound on the expected total size of the conflict lists [61, 62, 146]. For any "well-behaved" function g:

$$\mathbf{E}\left[\sum_{\Delta \in \mathscr{T}(R)} g(|S_{|\Delta}|)\right] = O(g(1/p)) \cdot F(R). \tag{66}$$

Thus, in a strong sense, the average conflict list size is $O(1/p)$, though deviations for individual cells are possible. For 2- and 3-d Voronoi diagrams, $F(R) = \Theta(|R|^D) = \Theta((p|S|)^D)$ where $D = 1, 2$ respectively.

6.1 Cache-Oblivious Distribution

A basic building block in algorithms based on sampling is the distribution of elements between buckets determined by the sample. In the case of sorting in a linear order, the buckets are simply the intervals determined by the sample. In this section, we first describe the basic building block, the *k-distributor* and then use it to implement a simple sampling based sorting algorithm. In geometric constructions, the situation is more complicated and will be described in the next section.

The k-distributor resembles the k-merger described by Frigo *et al.* [96] (which is also nothing else than the van Emde Boas layout of a binary tree [156] with buffers in the

edges). It takes k^3 elements and distributes between the k intervals determined by a k-splitter. It is built recursively of $\sqrt{k}$-distributors as shown in Figure 51. It consists of a *top* $\sqrt{k}$-distributor with its $\sqrt{k}$ outputs connected to $\sqrt{k}$ *bottom* $\sqrt{k}$-distributors through *middle buffers* of size $2k^{\frac{3}{2}}$ (implemented as circular queues). The recursion bottoms up for k smaller than a sufficiently large constant k_0, and in this case it consists of a (*basic*) k_0-ary distributor. For the purpose of cache efficiency, it is important that a k-distributor be layed out in consecutive memory positions (this is achieved using the van Emde Boas layout: recursively lay out the first the top half of the tree followed by the sequence of the lower subtrees). The k-distributor processes the k^3 elements in the input as follows: the top $\sqrt{k}$-distributor is invoked $k^{3/2}$ times. Each time it is invoked, $k^{3/2}$ elements are pushed down the distributor into the middle buffers, and then for each middle buffer that has become more than half full, the respective bottom $\sqrt{k}$-distributor is activated.

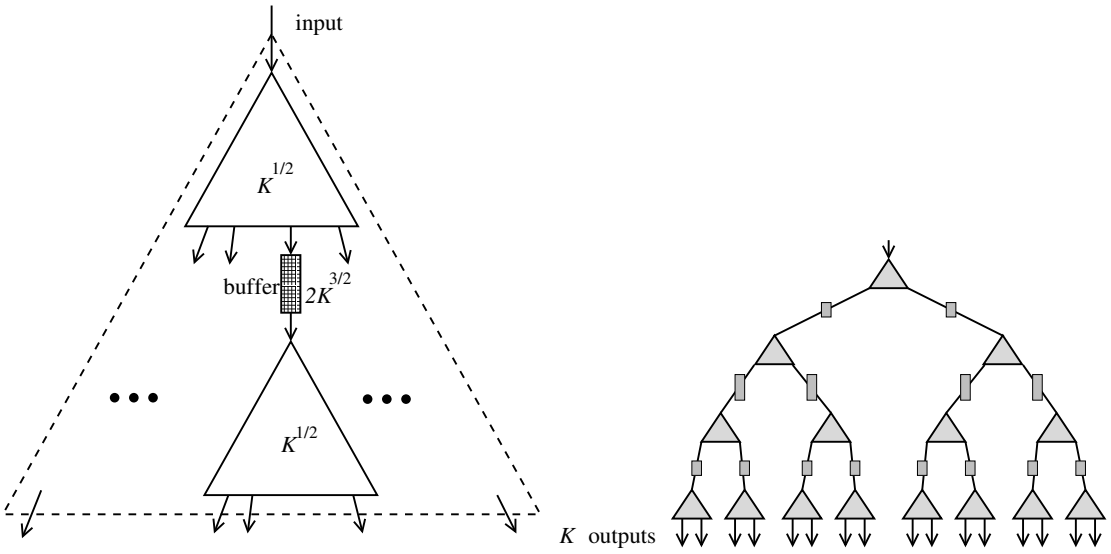


Figure 51: k-distributor: recursive definition and unfolded form.

The analysis of the k-distributor follows closely (in fact, it is essentially equal to) that

of the k-merger in [96]. The following lemma summarizes the construction (the proof paraphrases that in [96] and can be found in the appendix).

Lemma 6.1.1 *A k-distributor has size $O(k^2)$ and, assuming $B^2 \leq M$, the number of I/O's performed during an invocation is $O((k^3/B)\log_M k + k)$.*

Alternatively, following [40] in their modification of the k-merger, one can unfold the recursive construction, obtaining a tree whose nodes contain k_0-ary branchings and whose edges are buffers of size defined by the recursive layout: the edges in the middle level have buffers of size $k^{3/2}$, etc. Furthermore, the flow of data is modified as follows (as a result, a buffer does not need to be a circular queue): an active basic distributor reads an element from the input buffer and puts it in the appropriate output buffer as long as the input buffer is not empty and neither of the output buffers is full; if an output buffer becomes full, then the corresponding basic distributor is activated. The entire distributor is activated by activating the basic distributor at the root node r. Furthermore, the information that r's input buffer (the overall input) has become empty needs to be propagated down, so that buffers that are only partially full get processed.

The analysis then parallels that in [40] for the k-merger. The overall tree is decomposed into *base trees* by removing edges containing *large buffers.* More precisely, remove edges in order of decreasing buffer size (so following the recursive definition of the buffer size) until the resulting connected subtrees are of size at most $M/2c$ for an appropriate constant c. Let $\bar{k}$ be the number of leaves in such subtree; so its size is $c\bar{k}^2 \leq M/2$ (so it fits in main memory). With the tall-cache assumption $B^2 \leq M/2c$, $\bar{k} \cdot B \leq (M/2c)^{1/2} \cdot (M/2c)^{1/2} = M/2c$ and hence a base tree together with a buffer for each of its leaves can be fit in memory. Thus, if a k-distributor is a base tree, the number

of I/Os for distributing the k^3 input elements is $O(k^3/B+k)$. Otherwise, consider an activation of the root v of a base tree T_v. Loading the tree and a block for each output buffer requires $O(\bar{k}^2/B+\bar{k})$ I/Os and it will process $\Omega(\bar{k}^3)$ elements from the input buffer to v. Furthermore, if an output buffer of T_v becomes full and the corresponding node is activated, the cost of possibly evicting T_v out and again reloading it can be charged to the full output buffer. Since $\bar{k}^2/B+\bar{k} = O(\bar{k}^3/B)$, the required I/Os can be accounted by charging $1/B$ to each element inserted into a large buffer. The previous argument fails for the last time a buffer is filled because then it is not filled completely. This is fixed by charging $O(1/B)$ to each location in all the large buffers. Since this is bounded by the k-distributor size, this amounts to $O(k^2/B)$. Since $\bar{k} \geq (M/2c)^{1/4}$, then the number of large buffers on a path from the root to a leaf from the k-distributor is $O(\log_M k)$. Thus, a k-distributor requires $O((k^3/B)\log_M k)$ I/Os.

This unfolded picture will be useful for us in the geometric setting where we have a search tree which can then be easily augmented with buffers to obtain an efficient cache-oblivious search structure. The basic k-distributor leads to a simple sampling based sorting algorithm (an alternative to the previous funnel and distribution sort algorithms described in [96]) using expected $O((N/B)\log_M N)$ I/Os.

6.2 Conflict List Computation

In the context of geometric algorithms that use sampling, the analog of distributing elements among the intervals is the computation of *conflict lists*. We continue to concentrate on the specific case of Voronoi diagrams. We discuss in this section, two ways to compute conflict lists. The first one – roundabout, but effective in some cases where the more direct approach is not – is by a reduction to batch point location among hyperplanes; it can be

handled by the distribution approach discussed in the previous Section. The second one is more direct and extends the distribution approach in that an element does not go to a unique bucket.

6.2.1 Batched Point Location among Hyperplanes

Let S be a set of sites in $\mathbb{R}^{d-1}$ and R be a sample from S. We want to compute the conflict lists of the cells in the triangulation $\mathscr{T}(R)$ of $\mathrm{Vor}(R)$. For this, it is sufficient to compute the conflict lists of the vertices of $\mathrm{Vor}(R)$. Lifting to the paraboloid in $\mathbb{R}^d$ and point-hyperplane duality transforms a site $s \in S$ to a point p and a vertex v of $\mathrm{Vor}(R)$ into a hyperplane h in $\mathbb{R}^d$ so that s conflicts with v iff p is above h. Thus, the conflict list problem translates into a point location problem. There is a systematic way to do this for more complicated structures through *linearization* [195, 4, 140].

Thus, we are interested in the ability to answer efficiently simultaneous point location queries in the arrangement of a set H of r hyperplanes in $\mathbb{R}^d$. A simple way to reduce this problem directly to the one-dimensional batched search problem is to use point location among in a *slab* [77], but it is very space inefficient. (an infinite prism whose section is a convex cell of constant complexity, so that the hyperplanes do not intersect inside the prism) which is a one-dimensional search problem. [1] The space requirement of this data structure is $O(r^{2^d})$ This approach has been used before in parallel algorithms, see *e.g.* [14]. A more space efficient solution is possible using divide-and-conquer based on sampling directly to the hyperplanes. This results in a "hierarchical cutting" [56]. They

[1] For completeness, we recall the inductive construction. For $d = 1$, the problem is trivially the 1-d batched search problem; for $d > 1$, solve the point location problem recursively for the set of all the $\binom{r}{2}$ pairwise hyperplane intersections projected onto $x_d = 0$, then for each resulting cell in $x_d = 0$, the point location problems is now a one-dimensional search in a d-dimensional slab. The total size for this structure is $O(r^{2^d})$.

can be made cache-oblivious using the scheme of the k-distributor in its unfolded form. Let r_0 be a sufficiently large constant (its value will depend on the analysis). A hierarchical cutting is constructed iteratively as follows. In the 0-th iteration we start with an infinite cell Δ_0 and associated set $H_{|\Delta_0}$. For $i > 0$, we maintain a decomposition T_i into cells Δ, each with the associated set $H_{|\Delta} \subseteq H$ of hyperplanes intersecting it, so that $|H_{|\Delta}| \leq N/r_0^i$. T_i is obtained from T_{i-1} by taking in each $\Delta \in T_i$ with $|H_{|\Delta}| > N/r_0^i$ a random sample $R_\Delta \subseteq H_{|\Delta}$ of size $O(r_0 \log r_0)$, decomposing it (in a canonical form) into cells of constant complexity, and then computing the conflict list for each cell (by brute force, checking each $h \in H_{|\Delta}$ against all the cells. This is repeated (an expected constant number of times) until the size reduction by a factor r_0 is achieved. Thus, we obtain a $(\leq k_0)$-ary tree, where $k_0 = c_0(r_0 \log r_0)^d$, of depth $D = \Theta(\log_{k_0} r)$ and with a number of leaves $L = \Theta(r^d)$ (using the "vertex-sensitive" argument in [56]; specifically, that the size of the triangulation is proportional to the number of vertices in the sample plus the size of the sample). [2] To this tree, we can apply the distribution tree construction of Section 6.1: The number of

[2] To verify this, consider $\Delta \in T_{i-1}$, and let $v(H,\Delta)$ denote the number of vertices of the arrangement of H inside Δ. The expected number of vertices the arrangement of R_Δ inside Δ is $(Cr_0 \log r_0/n_\Delta)^d v(H,\Delta)$ where $n_\Delta = |H_{|\Delta}|$, and the expected number of vertices on the boundary of Δ is $D(r_0 \log r_0)^{d-1}$. So we have, for sufficiently large constant r_0,

$$\begin{aligned}
|T_i| &\leq \sum_{\Delta \in T_{i-1}} \left\{ \left(\frac{Cr_0 \log r_0}{n_\Delta} \right)^d v(H,\Delta) + D(r_0 \log r_0)^{d-1} \right\} \\
&\leq \left(\frac{Cr_0 \log r_0}{N/r_0^i} \right)^d \sum_{\Delta \in T_{i-1}} v(H,\Delta) + \sum_{\Delta \in T_{i-1}} D(r_0 \log r_0)^{d-1} \\
&\leq \left(\frac{Cr_0 \log r_0}{N/r_0^i} \right)^d \cdot O(N^d) + \sum_{\Delta \in T_{i-1}} D(r_0 \log r_0)^{d-1} \\
&\leq C' r_0^{(i+1)d} \log^d r_0 + D(r_0 \log r_0)^{d-1} |T_{i-1}| \\
&\leq C'' r_0^{(i+1)d} \log^d r_0,
\end{aligned}$$

where we have used the fact that $\sum_{\Delta \in T_{i-1}} v(H,\Delta) = O(N^d)$.

input elements is $N = \Theta(L^3) = \Theta(r^{3d})$; in the edges of the middle level, we insert buffers of size $\Theta(L^{3/2}) = \Theta(r^{3d/2})$, and so recursively with the top tree and the bottom trees. The total size of the distribution tree is then $\Theta(r^{2d})$. Assuming $M \geq B^2$, the total number of I/Os needed for the batched point location is $O((N/B)\log_M r + r^d)$. For the conflict list computation problem, the point location data structure must be complemented by a list $L(v)$ of the conflicts for each leaf v. Since the size of the tree is $\Theta(r^{2d})$, this does not increase the space requirement of the data structure. The complete conflict list computation consists of the batched search and then writing out the conflicts: if p ends in the leaf v then p conflicts with each $h \in L_v$. Thus, to obtain the lists of conflicts, the pairs (p,h) for $h \in L_v$ need to be sorted according to h (to bring together all conflicts of the same h). So, the total I/O cost is then

$$O((r^{2d}/B) + (N/B)\log_M r + r^d + (C/B)\log_M r) = O((N/B)\log_M r + r^d + (C/B)\log_M r)$$

where C is the total number of conflicts determined.

6.2.2 Direct Approach

Let R be a p-sample from a set of N sites S with $p = r/N$. The idea is also to use a hierarchical cutting for $\mathrm{Vor}(R)$. We concentrate on the 3-d case (the procedure also applies to the 2-d case, but then the result is not good for use in an optimal algorithm). As before, let r_0 be an appropriate constant, and let $T_0, T_1, \ldots, T_l$ be the hierarchy of cuttings. The resulting depth and number of leaves are $D = \Theta(\log_{k_0} r)$ and $L = \Theta(r^2)$ (here again a "vertex-sensitive" analysis is needed). [3] We want to apply the distribution tree construction but taking

[3] Though it is not essential, one can prove with a bit more effort this tight $\Theta(r^2)$ bound rather than a weaker $\Theta(r^{2+\varepsilon})$ (and a similar analysis is needed later for the 3-d Voronoi diagram algorithm anyway). So, consider $\Delta \in T_{i-1}$, and let $v(S,\Delta)$ denote the number of Voronoi vertices determined by four sites in $S_{|\Delta}$. The expected number of vertices of $\mathrm{Vor}(R_\Delta)$ inside Δ is $(Cr_0 \log r_0/n_\Delta)^4 v(S,\Delta)$ where $n_\Delta = |S_{|\Delta}|$, and the expected

into account that an element in the input propagates into multiple buckets rather than into a unique one. We use the same (corresponding) buffers sizes as before: For the middle level, we insert buffers of size $\Theta(L^{3/2}) = \Theta(r^3)$. The size of the tree is then $O(r^4)$ and can be constructed using $O(r^4/B)$ I/Os. The analysis (similar to that for the unfolded k-distributor) then shows that the number of I/Os performed can be accounted by charging $O(1/B)$ to every element stored in large buffers. The latter is bounded by $O(N \log_M r)$ plus the total size of conflict lists corresponding to large buffers, which is dominated by those in the last level: each of the $L = \Theta(r^2)$ leaf cells can have a conflict list size up to $\Theta((N/r)r^\delta)$, where δ is a small positive fraction, for a total of $\Theta(Nr^{1+\delta})$. Thus, for $N = \Theta(r^6)$, the total number of I/Os is

$$O((r^4/B) + (N/B)\log_M r + (N/B)r^{1+\delta} + r^2).$$

6.3 I/O-Efficient RIC

A variant of the random incremental construction performs the insertions in a small number of batches [146]. More precisely, it considers a *gradation* of the set S by sampling

number of vertices on the boundary of Δ is $D(r_0 \log r_0)$. So we have, for sufficiently large constant r_0,

$$\begin{aligned}
|T_i| &\leq \sum_{\Delta \in T_{i-1}} \left\{ \left(\frac{Cr_0 \log r_0}{n_\Delta} \right)^4 v(S,\Delta) + D(r_0 \log r_0) \right\} \\
&\leq \left(\frac{Cr_0 \log r_0}{N/r_0^i} \right)^4 \sum_{\Delta \in T_{i-1}} v(S,\Delta) + \sum_{\Delta \in T_{i-1}} D(r_0 \log r_0) \\
&\leq \left(\frac{Cr_0 \log r_0}{N/r_0^i} \right)^4 \cdot O(N^2 (N/r_0^{i-1})^2) + \sum_{\Delta \in T_{i-1}} D(r_0 \log r_0) \\
&\leq C' r_0^{2(i+1)} \log^4 r_0 + D(r_0 \log r_0)|T_{i-1}| \\
&\leq C'' r_0^{2(i+1)} \log^4 r_0,
\end{aligned}$$

where we have used the fact that $\sum_{\Delta \in T_{i-1}} v(S,\Delta) = O(N^2 (N/r_0^{i-1})^2)$ (this follows by the usual lifting to the paraboloid in $\mathbb{R}^4$, because the number of vertices in the $(\leq k)$-levels of an arrange of N hyperplanes is $O(N^2 k^2)$ [61]).

successively:

$$\emptyset \subseteq S_0 \subseteq S_1 \subseteq \ldots \subseteq S_{l-1} \subseteq S_l = S$$

(for convenience in the algorithm description, the resulting sets are numbered backwards). Such gradation can be extracted from a random permutation by writing from left to right the subsets determined by prefixes of the permutation. The construction then proceeds in l rounds, adding $R_i = S_i \setminus S_{i-1}$ to $\mathscr{T}(S_{i-1})$ to obtain $\mathscr{T}(S_i)$. The case in which S_{i-1} is a $(1/2)$-sample from S_i, and so $l = \log n$, is convenient in the usual sequential random access model. An I/O-efficient and cache-aware variant was described in [68]; there S_{i-1} is a $(1/\mu)$-sample from S_i where $\mu = \Theta(m^{1/D})$ (for $i = l$, a different probability is used and the last round is also handled differently).

6.3.1 A Simple Variant: Local RIC

We describe a simple variant that is easily implementable and is "likely" to be I/O-efficient, though we do not have a proof.

We consider the now well-know RIC construction with history dag for Delaunay triangulations in 2-d or tetrahedralizations in 3-d. The random permutation that determines the insertion order is divided in blocks of size $\mu, \mu^2, \ldots, \mu^i, \ldots$ that determine a gradation. As above, R_i is the i-th block, S_i consists of the blocks $R_1, \ldots, R_i$. The insertions follow the random permutation overall but are localized in each level of the gradation. Given $\mathscr{T}(S_{i-1})$, first we propagate all the points in R_i down the dag to the leaves (locate each $p \in R_i$ in $\mathscr{T}(S_{i-1})$). This is I/O-efficient if all the point in a node are propagated one level down in a batch. Second, we consider the triangles (or tetrahedra in 3-d) in some arbitrary order (following spacial locality seems to be appropriate though), and for each one we insert the points of R_i that it contains in the random order determined by the overall permutation (note

that this requires more searches down the dag, but we may do these one by one in the order they appear).

This approach turns out to be a refinement of the "blocked RIC" proposed recently by Amenta and Choi [17]. From their analysis, it follows that our "local RIC" has optimal running time. Our approach seems the "right way" to implement their blocked RIC, since the locality is provided adaptively for each level of the gradation by the same algorithm, rather than relying on a separate data structure (kd-tress) which provide a fixed blocking through the algorithm.

On the other hand, this algorithm is "almost" the basic one used in [68], with the important difference that here (unlike what it is usual in divide and conquer algorithms based on sampling), when taking care of the conflict list of a triangle, we allow the insertions to affect the global triangulation. Unfortunately, we cannot claim that the simple variant is also I/O-efficient, and have not performed experiments yet. However, the similarity to the algorithm in [68] indicates that some good theoretical bound might be possible.

6.3.2 Cache-Oblivious RIC

To obtain an efficient (provable) cache-oblivious version, we use a shorter gradation obtained by growing S_i faster. Specifically, we set S_i to be a q_i-sample from S with

$$q_i = \frac{1}{N^{(1-\nu)^i}},$$

for a small positive fraction ν to be determined in the analysis. Note that $q_0 = 1/N$, $q_1 = 1/N^{1-\nu}$, and it approaches 1 as i grows. This gradation is generated by taking S_{i-1} as a γ_i-sample from S_i with

$$\gamma_i = \frac{q_{i-1}}{q_i} = \frac{1}{N^{(1-\nu)^{i-1}\nu}}.$$

Furthermore, with a *forward view*, R_i is a p_i-sample from $S - S_{i-1}$, where $p_i = (q_i - q_{i-1})/(1 - q_{i-1}) \approx q_i$ (see [68]) which is useful in the analysis. In the i-th round, given $T_{i-1} = \mathscr{T}(S_i)$ and its conflict lists, the algorithm proceeds as follows:

1. For each $\Delta \in T_{i-1}$ do

   (a) Collect $R_{i|\Delta}$ and construct $\mathrm{Vor}(R_{i|\Delta})$ restricted to Δ.

   (b) Construct a data structure for conflict list computation (Section 6.2.1) and use it to determine the conflict lists for the vertices of $\mathrm{Vor}(R_{i|\Delta})$ inside Δ.

2. Use sorting to obtain $\mathrm{Vor}(S_i)$ out of the pieces $\mathrm{Vor}(R_{i|\Delta})$, its decomposition $T_i = \mathscr{T}(S_i)$ and the conflict lists of T_i with respect to S.

In Step 1(a), we can afford to construct $\mathrm{Vor}(R_{i|\Delta})$ with a non-efficient algorithm: even one that checks for every tuple (3 or 4) of sites whether it defines a Voronoi vertex.

In the 2-d case, in $\Delta \in T_{i-1}$, the expected size of the conflict lists computed is $O((q_i|S_{|\Delta}|)/q_i) = O(|S_{|\Delta}|)$. So, the I/O-cost of Step 1 for $\Delta \in T_{i-1}$ is:

$$\frac{|R_{i|\Delta}|^6}{B} + \frac{|S_{|\Delta}|}{B}\log_M |R_{i|\Delta}| + |R_{i|\Delta}|^3.$$

Sorting in Step 2 dominates the cost: it involves the movement of all the conflict lists, whose total size is $O(|S|)$; so it requires $O((|S|/B)\log_M |S_i|)$ expected I/Os. Thus, the I/O-cost of the i-th round is

$$\sum_{\Delta \in T_{i-1}} \left(\frac{|R_{i|\Delta}|^6}{B} + \frac{|S_{|\Delta}|}{B}\log_M |R_{i|\Delta}| + |R_{i|\Delta}|^3 \right) + \frac{|S|}{B}\log_M |S_i|.$$

Choosing ν appropriately small (in particular, so that $6\nu \leq 1/2$) and using the sampling bounds developed in [68], we obtain that the I/O cost over all the rounds is $O((N/B)\log_M^2 N)$.

In the 3-d case, in $\Delta \in T_{i-1}$, the expected size of the conflict lists computed is $O((1/q_i)(q_i|S_{|\Delta})^2) = O(q_i|S_{|\Delta}|^2)$ and the overall size of the lists sorted in Step 2 is $O(p_i|S|^2)$. The I/O cost over all the rounds is then $O((N^2/B)\log_M N)$.

In both cases the I/O cost is off by a factor $\log_M N$ from the worst case optimal. In the next section we discuss variants that achieve the optimal bounds.

6.4 Optimal Algorithms

6.4.1 2-D Case

The difficulty is that every time a sample R is taken from S, the expected size of the conflict lists is $O(|S|)$, and the hidden constant accumulates when this sampling process is iterated. Several alternatives have been explored to get around this problem and ensure that the overall conflict list size is $O(N)$ where N is the size of the original set, at all the levels of the recursion. We are aware of three approaches:

Pruning ([161]): For a subproblem triangle Δ, one computes the Voronoi diagram restricted to the boundary and use this to decide which sites that are outside Δ contribute vertices in the final Voronoi diagram inside Δ, and so must be kept in the conflict list. Then a global accounting shows that the overall conflict list size is linear.

Two step sampling ([158]): Take a sample of size $O(N/\log^c N)$ for appropriate c, use a non-optimal algorithm to find the Voronoi diagram of the sample, compute then the corresponding conflict lists, and then finish with the resulting small problems using again a non optimal algorithm. Here, the major difficulty is to compute the conflict lists.

Graph separators ([72]): Instead of recursing on each subproblem, planar graph separators are used to group subproblems into *clusters* that have "small" boundaries so that the total conflict list size is still linear over all levels of the recursion.

We describe now some details of the cache-oblivious implementation of the pruning approach (though it might be possible to implement the others as well). Note that we are only interested in the expected running time, so we avoid some complications needed in the parallel algorithm in [161] and also in [106] (they use a sampling technique called *polling* to achieve bounds with high probability). The outline is the same as in [161] (and also [106]), but for simplicity, we restrict it to 2-d Voronoi diagrams (those references handle 3-d half-space intersection) and adopt the primal-dual view used in [51]. In this view, the Voronoi diagram picture is used to determine the subproblems (via its decomposition), but what we actually construct is the dual Delaunay triangulation. Pruning is more easily visualized and analyzed in the Delaunay diagram.

With the primal-dual view, the input to the problem is a set of sites S and a closed polygonal chain P of Delaunay edges whose vertices are in S and enclosing the rest of the sites in S. The problem is to "fill" the Delaunay triangulation inside P, denoted by $\mathrm{Del}_P(S)$. The initial problem is the overall point set together with the convex hull as bounding Delaunay chain. For this, and also inside the algorithm, we need a cache-oblivious 2-d convex hull algorithm with I/O-cost $O((N/B)\log_M N)$; this can be obtained with a minor modification of the sorting algorithm describes earlier or alternatively with the cache-oblivious RIC approach (they are essentially the same because no clean-up is needed in this case). Let $N = |S|$. The outline is as follows:

1. Obtain an N^ε-sample R from S.

2. Compute $\mathrm{Vor}(R)$ and its decomposition $\mathscr{T}(R)$ into diamonds.

3. Construct the conflict list computation data structure described in Sec. 6.2 and use it to compute the conflict lists of $\mathscr{T}(R)$ with respect to S. If any conflict list has size $\Omega(N^{1-\varepsilon}/\log N)$

then repeat steps 1-3.

4. For each diamond $\Delta \in \mathscr{T}(R)$ do the following:

   (a) Compute $\mathrm{Del}(S)$ restricted to each of the 4 edges forming its boundary (this is a 2-d convex hull computation). This results in 4 Delaunay chains χ_i, $i = 1,2,3,4$.

   (b) For each site $s \in S$ not in any χ_i, determine for each i the side of χ_i where s lies (using sorting of the points projected on the corresponding edge of Δ, we can identify for each s the edge in χ_i on which s projects; then a simple sidedness test suffices).

   (c) Using sorting, bring together edges that might be repeated in the χ_i's and use this to eliminate edges in the χ_i's that do not bound a triangle in $\mathrm{Del}_P(S)$. Then scan the remaining edges in the lists ξ_i and P to determine closed polygonal chains P_i (that enclose Delaunay triangles still to be computed).

   (d) For each $s \in S$ not in any χ_i, determine which of the P_i encloses s (using the results of the sideness queries above). This results in a set S_i for each P_i.

   (e) For each pair (S_i, P_i), recursively compute $\mathrm{Del}_{P_i}(S_i)$.

In Step 2, $\mathrm{Vor}(R)$ could be computed using recursion, but choosing ε appropriately small, this can also be done with a trivial (brute force) algorithm. It is clear that the total conflict list size for subproblems at the same level of the recursion is $O(N)$ because a site is included in a subproblem either (i) if it is inside the bounding polygon (so it belongs to only one subproblem), or (ii) if it is on the bounding polygon (so it can be charged to a triangle inside the polygon). The expected running time follows the recurrence

$$T(N) = O((N/B)\log_M N + N^{6\varepsilon}/B + N^{3\varepsilon}) + \sum_i T(|S_i|)$$

which, choosing ε appropriately small and taking into account the tall-cache assumption, has a solution $O((N/B)\log_M N)$.

6.4.2 3-D Case

As already pointed out, this problem is (artificially) simpler because we are interested only in worst case optimality. Here a divide-and-conquer approach works without resorting to pruning (if we are careful to use a vertex-sensitive analysis [56]). We prefer to modify slightly the oblivious RIC algorithm to obtain the worst-case optimal bound $O(N^2/B)$ (but the divide-and-conquer approach would the advantage of producing a hierarchical search structure). The outline is as follows:

1. Run the cache-oblivious RIC algorithm until the expected size of the subproblems is in the interval $[\Omega(\log N), O(\log^2 N)]$.
2. For each resulting subproblem Δ, compute $\mathrm{Vor}(S_{|\Delta})$ using the cache-oblivious RIC algorithm (run to the end).
3. Put together (clean-up) the diagrams $\mathrm{Vor}(S_{|\Delta})$ into $\mathrm{Vor}(S)$.

In the last step, we again use sorting. However, we note that only Voronoi cells that intersect the boundaries of the $\Delta \in \mathscr{T}(R)$ need to participate in a sorting, in order to be able to put together the cleaned-up $\mathrm{Vor}(S)$. The expected number of these cells is $O((N/\log N)^2 \cdot \log N) = O(N^2/\log N)$ and so the algorithm can afford to sort without exceeding the bound $O(N^2/B)$. We claim that the overall expected I/O-cost is $O(N^2/B)$. A detailed calculation will be given in the complete version.

6.5 Implementation Results

We are not aware of any theoretically provable I/O efficient Delaunay triangulation codes that have been developed till date. The closest effort that we know of was an implementation of parallel Delaunay done by Blelloch et.al [38]. This implementation was not theoretically optimal and was not designed keeping in mind I/O issues for very large point sets.

Borrowing from the ideas we used in the development of the cache oblivious Voronoi diagram and Delaunay triangulation algorithms, we implemented a Delaunay triangulation algorithm based on the divide and conquer paradigm to do very large size Delaunay triangulations in 2D. The algorithm implemented is a very simplified version of the theoretical algorithms developed the chapter. We made the following additional assumptions for the algorithm we implemented:

Memory Size There exists an ε such that the memory can store N^{ε} and $N^{1-\varepsilon}$ size Delaunay triangulations. This assumption lets us restrict the gradation to just two levels in practice.

Base Case We are given an efficient and robust implementation that creates Delaunay triangulations in internal memory.

These not elegant theoretical assumptions actually apply in practice. For instance, even if one wants to do a triangulation of a billion points, one only needs to work with 10,000 points in internal memory($\varepsilon = .5$). Most internal memory Delaunay triangulators available can easily deal with these many points. We chose to experiment with LEDA [135] and Triangle [168] as our internal memory triangulators. The simple algorithm that we chose for implementation is as follows:

1. Obtain an N^{ε}-sample R from S.

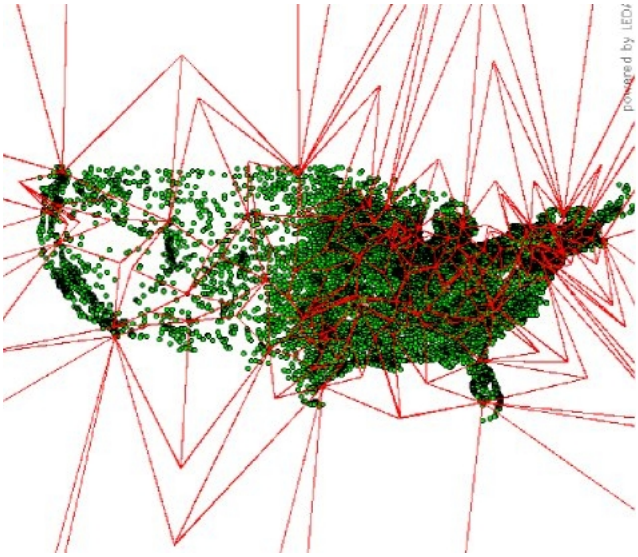


Figure 52: An example of cuttings for a sample of cities in the US.

2. Compute Vor(R) and its decomposition $\mathscr{T}(R)$ into diamonds (see figure 52).

3. Construct the conflict lists of $\mathscr{T}(R)$ with respect to S. Once this is done, external memory sorting (using TPIE) is used to bring together all the conflicting points of each $\Delta \in \mathscr{T}(R)$. For fast point location, we used approximate nearest neighbor searching using ANN [144] to determine a good starting point to walk. In most cases ANN already gave us a face of the Delaunay that conflicted with the query point.

4. For each diamond $\Delta \in \mathscr{T}(R)$ do the following:

   (a) Collect the conflict list of each diamond by a linear scan from the file containing the conflict list (this only does the required number of I/Os).

   (b) Triangulate the conflict list using the internal memory triangulator which does the Delaunay for us.

   (c) Report all triangles whose circumcenters lie in or on Δ.

Figure 53 shows a plot with some timings.

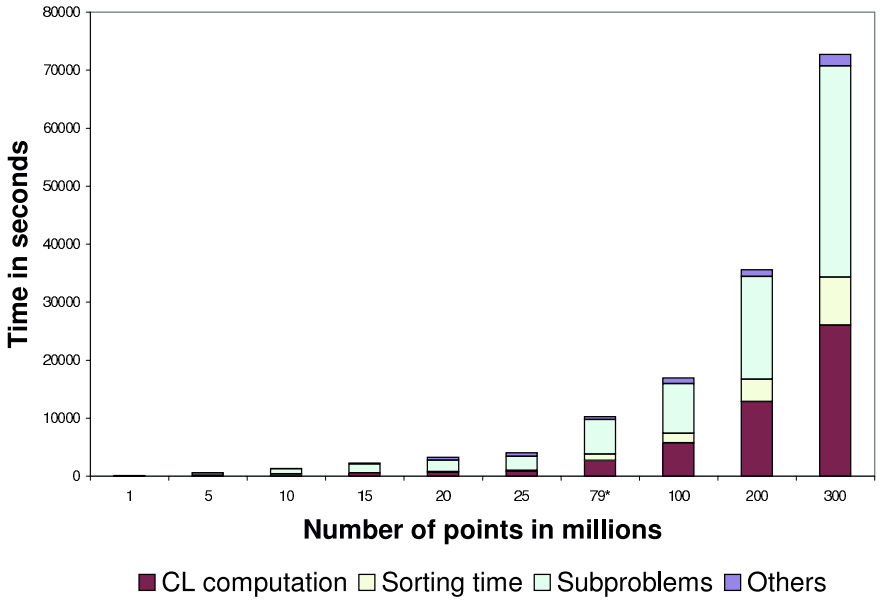


Figure 53: Timings in 2D using Triangle to solve the sub problems. Platform: Dual Athlon MP 1800+ with 1GB RAM, 60GB IBM Hard disk drive, Only 1 processor being used.

The 79 million data set is the standard TIGER data set used to experiment with external memory geometric algorithms [155]. More details of the implementation along with code fragments, examples and work in progress are available at:

`http://www.compgeom.com/ramos/`.

We also tried some hacks like sorting the points spatially as well as in x-coordinate and then doing the Delaunay using LEDA, but the biggest bottleneck then was the 4GB limit imposed by the OS. We use 64-bit file operations in our implementation to get around the

4GB limit existing on most systems. We hope to be able to do the Delaunay triangulation of a billion points in in the near future. We ran out of disk space for doing .5 billion points (The sorting phase of TPIE crashed saying 'Insufficient Disk Space' on the 60GB hard disk drive we were using for this experiment). Please look at the above web page for more details of the experimentation.

We implemented the algorithm in C++. The total length of the code is less than 2000 lines. We plan to implement the 3D extension of this algorithm in the near future.

6.6 Conclusions

Brodal and Fagerberg [41] have shown how to weaken the tall cache condition for algorithms based on the k-merger of [96]. Since our k-distributor follows essentially the same structure, such improvement also applies in our case.

In the case of 3-d Voronoi diagram, we have obtained a worst-case optimal algorithm. However, most data does not exhibit this behavior, and so a cache-oblivious output sensitive algorithm would be interesting.

6.7 Future Work

In the near future we plan to implement the algorithm that we present here in 3D.

CHAPTER VII

Hand Recognition

"Art is the imposing of a pattern on experience, and our aesthetic enjoyment is recognition of the pattern."

ALFRED NORTH WHITEHEAD

We discuss the issues and challenges in the design of a hand outline based recognition system. Our system is easier to use, cheaper to build and more accurate than previous systems. Extensive tests on more than 700 images collected from 70 people are reported. Classification, verification and identification of the input images were done using two simple geometric classifiers. We describe a novel minimum enclosing ball classifier which performs well for hand recognition and could be of interest for other applications.

7.1 Introduction

Biometric recognition systems find applications in security systems of varying requirements. While finger printing and iris based systems work well for high security applications, they are not as suitable for medium and low security applications because of privacy concerns. Most users are not comfortable providing an identifying biometric signature for access to low risk facilities. Hand Geometry based verification systems find more acceptance because hand geometry is not considered distinctive enough to establish a positive

identity. Yet hand geometry differs enough from person to person that it is sufficient for medium security applications.

Hand geometry recognition systems may provide three kinds of services. Verification, classification and identification. For verification the user provides his identity along with the hand geometry and the system verifies his identity. Verification can be used alone or in conjunction with other security systems where the user can enter his identity. For example, secure web-access can use hand geometry to verify a user [163]. For classification the user does not provide any identity information but is known to be legitimate. The system classifies him assuming his presence in the user database. This can be used in applications where the users can be trusted. For example, use of photocopying facilities in a university can be regulated using a hand based recognizer instead of a password. There is minimal risk of intruders because most such facilities are secured by other means. For identification the user does not provide any identity information other than the hand geometry and may be an intruder. The system tries to identify the individual or deny access. This is required in systems where a user will not be willing to go through the hassle of entering his identity. For example, a restricted dining facility or health club may install a hand geometry based identification system to keep track of its users.

Chapter Outline First we summarize previous work in section 7.2. Then we describe data collection and feature extraction steps of our experiments in section 7.3 and 7.4. The nearest box classifier is described in section 7.5 followed by the minimum enclosing ball classifier in section 7.6. Experimental results for both classifiers are presented in section 7.7. Finally we present a summary and conclusions of the chapter in section 7.9.

7.2 Previous Work

The idea of using hand geometry for verification is not new. Most of the early work is in the form of patents. Recently there has been some documented research in this area. Arun Ross developed a hand geometry based verification system for his master's thesis and used it for a prototype web security system [163] (see also [117, 118]). They developed a verification system using hand images obtained from a digital camera. A set up is used for acquiring these images that uses pegs to guide the user to place the hand in more or less the same way every time. The set up also obtains the side view of the palm by placing a mirror at 45 degrees to the camera. They obtained 10 images each from 50 people and ran experiments on them. However 140 of these 500 images had to be discarded because users didn't do what was expected of them. From the remaining images they report a false acceptance rate of 2% and a false rejection rate of 15%. They extract several features like length and width of fingers for each hand image. Five images are taken for each person's hand to train the system. A feature vector containing the average for each feature is stored for each hand. A new hand image from a known person is then verified by calculating a distance function between the feature vector of the new hand image with the feature vector stored in the system for that person. The distance should be less than a tuned threshold. They report results using four different distance functions.

Jain and Duta [116] report on experiments with another way of doing verification. They try to align finger contours and measure the mean alignment error between them. They use the same set up as in [163]. They experimented with 353 images from 53 persons and report a false acceptance rate of 2% and a false rejection rate of 3.5%. Their method is an improvement over [163] in that it can work even if the user places his hand in different positions during different data acquisition events. Also their method can be used as a first

step before the feature based approach of [163].

Raul Sanchez-Reillo et al. [165] report experiments with another system similar to the one in [116]. However they implemented verification as well as classification. They tested using a relatively smaller database of 200 hand images from 20 persons. They use various distance metrics to measure the distance between feature vectors of hand images and use them for classification and verification. The best among their methods achieve 97 percent success in classification and error rates below 10 percent in verification.

Öden et al. [151] report on a system for classification and verification using implicit polynomials. They fit fourth degree implicit polynomials in each finger and compare them using algebraic invariants. Using this method they achieve 85% success in identification and 90% success in verification. However when they combine this method with geometric features similar to that in [163], they achieve 95% success in identification and 99% success in verification. They worked on 30 images from 28 people.

Recognition Systems Inc's HandKey II is a commercial biometric hand recognition system [160]. It is based on a patent by Sidlauskas [169] which describes a hand verification system. A 3D image of the hand is obtained using a digital camera and a pair of orthogonal reflecting surfaces. The side view is used to obtain the thickness of the hand which is used as an additional feature for classification. The user has to input an identifying code. Note that this system only performs verification. To the best of our knowledge no experimental study for this system has been published anywhere including the company website [159].

Another patent by Faulkner [91] describes a biometric measuring apparatus for verifying a user based on measurements performed on the user's hand using a 3D view. It uses finger guides to assist in hand positioning to obtain reproducible finger orientations. An outline of the hand is drawn within which the user has to keep his hand. A calibration

image is formed with just the outline. The features of the hand are computed with respect to the calibration image to nullify the movement of the lighting system with respect to the base on which the hand is kept. Features such as thickness of fingers, size of fingers, size of palm, thickness of hand etc are computed. The user has to input an identifying code. We could not find any experimental data about the performance of this system.

Also related is work by Duta et al. [82] and Li et al. [136] on palm print based verification systems.

7.3 Data Collection

All previous experimental work on hand recognition has been on hand images obtained from a set up that includes a digital camera. Most of them also have pegs or a similar mechanism to guide the hand in a somewhat consistent position. We however collected data using a document scanner. Thus no special set up was required. Also the users were free to keep their hands anywhere on the scanner. The only instruction given to the users was to keep their fingers separated and have their hand roughly vertical on the scanner surface. Users were free to stretch their hands to whatever level they felt comfortable. In fact some of the users were encouraged to stretch their hands to different extents for different scans so as to generate difficult data. Our system is thus more tolerant of user inconsistencies. In all we took around 10 scans of the right hand of 70 people to obtain a total of 714 images. The images were scanned at 90 dpi using a HP Deskscan scanner. There were a number of images where the hand was not placed flat on the scanner or one of the fingers got cut off by the scanner edge. However we kept all such images as part of our experiments.

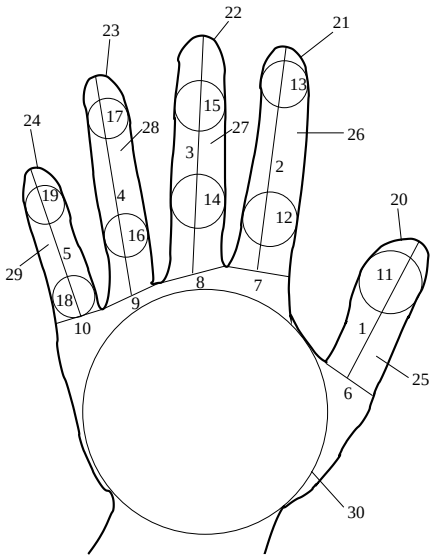
7.4 Feature Extraction

In this section we describe our feature extraction algorithms. We extract 30 different features form each image. The first step is to convert the colored image into a binary image. This is done as follows. The image is first converted to CIELAB color model and thresholded using a fixed value in the B channel. The threshold value was chosen by looking at many hand images and picking a value that gets rid of most of the background noise. Edge areas of hands were quite faint and hence were indistinguishable from the background noise if we look at the intensity values only. However hand pixels and noise pixels had different color signatures. Noise pixels had cooler (i.e. bluish) colors. Hence we could remove them effectively by thresholding in the color channel B. This is not possible to do using RGB colors because they do not have a color channel.

From these binary images the hand contours were extracted using the following algorithm. First the boundary pixels were identified as those which had less than four neighbors. Then we applied hit and miss transform to remove dead ends and cycles in the boundary. After that we found the right most point and traverse the resulting boundary in counterclockwise order.

Thirty different features were then extracted from each hand. The features are illustrated in figure 54.

Our feature extraction algorithms relied on detecting tips of the fingers and centers of the valleys between the fingers correctly. Here's how this was accomplished. For each point q in the hand outline consider the points p and r at a certain distance (about half the typical finger length) along the outline clockwise and counterclockwise. Consider the angle pqr. This angle is the lowest at the tips and highest at the valleys. We plotted a graph



1. Thumb length
2. Index finger length
3. Middle finger length
4. Ring finger length
5. Pinkie length
6. Thumb width
7. Index finger width
8. Middle finger width
9. Ring finger width
10. Pinkie width
11. Thumb circle radius
12. Index circle radius lower
13. Index circle radius upper
14. Middle circle radius lower
15. Middle circle radius upper
16. Ring circle radius lower
17. Ring circle radius upper
18. Pinkie circle radius lower
19. Pinkie circle radius upper
20. Thumb perimeter
21. Index finger perimeter
22. Middle finger perimeter
23. Ring finger perimeter
24. Pinkie perimeter
25. Thumb area
26. Index finger area
27. Middle finger area
28. Ring finger area
29. Pinkie area
30. Largest inscribed circle radius

Figure 54: Feature extraction.

of these values, one for every point on the outline, smoothed it to get rid of some noise and detected peaks and valleys. Peaks represented finger tips, and valleys represented dips between fingers. There were five well defined peaks roughly equally spaced and some other smaller peaks created by the noise at the bottom of the hand. We pick the five highest peaks and mark the first one to be the thumb, second to be the index finger, and so on.

Once we had positions of finger tips and valleys between fingers, computing features was simple. Length was measured to be the distance from the tip to the midpoint between two valleys around that finger. Width was measured as the distance between two valleys.

Area was measured as the area of the finger after being cut off by the line connecting two surrounding valleys. Perimeter was the length traversed along the finger.

We used Euclidean Distance Transform (EDT) [69, 122] to find greatest inscribed circles. The greatest inscribed circle of the whole hand was found by computing EDT of the binary hand image and finding the maximum value. The greatest inscribed circles for the

fingers were found by cutting the finger off, and running EDT on that finger separately. We cut four of the fingers further into two pieces (top half and bottom half), and fit the circle into each of those parts.

7.5 Nearest box classifier

In this section we describe a simple classifier that we use to implement hand geometry verification, classification, and identification. For each person we pick a small number (3 to 5) of hand images as training set. We find the bounding box of each training set in the 30 dimensional feature space. For a query vector, the distance to these bounding boxes in L_∞ metric is used as a measure similarity. The distance along each feature axis is normalized using the maximum difference variability observed for that feature in the training set. Thus a feature that varies a lot gets its effect diminished. For verification, we use a experimentally determined threshold ε to decide whether a query feature vector is close enough to a given training set. For classification we simply classify the query point to the set with the nearest bounding box. Identification is implemented by classifying the query point assuming it to be legitimate followed by verification.

7.6 Minimum enclosing ball classifier

In this section we describe a novel classifier which performs better than the nearest box classifier and is also of general interest for other classification problems. The classifier is based upon ideas from mathematical programming [148], core sets [47, 132], support vector clustering [30, 178] and pattern classification [79]. For our application, it also beats the standard Support Vector Machine implementation with Gaussian kernel in some cases [79].

Suppose we are given N d-dimensional features $x_1, x_2, \ldots x_N$ from c classes $D_1, D_2, \ldots D_c$.

Given a new feature vector v we want to classify it in one of the classes.

Our classifier first maps all x_i to a higher dimensional space using a Gaussian kernel, employs a novel way to approximate minimum enclosing balls in the mapped space of each D_i and then uses the Voronoi diagram of the centers for classification. Note that instead of using the convex hulls of each D_i for separation (as is done in standard SVMs) we use the voronoi cell of the centers of minimum enclosing balls for classification purposes. This of course might be worse for some instances and some applications than using SVMs but at least for our application this classifier looks more promising to use than directly using SVMs.

In the absence of information about the probability distribution of data, there is little theoretical justification of one window width over another for the Gaussian kernel. Nevertheless the classifier seems to work well in practice (see figure 55) for a large range of window widths. The approximate method of computing minimum enclosing balls used in the classifier can also be used for support vector clustering, is much easier to code than the current Lagrange multiplier based methods [30, 178] and is conceptually simpler (see algorithm 10). One drawback of this method is that we don't know how to take care of outliers efficiently in practice, although theoretically this is known [81]. In practice, we observed that the classifier works for a few outliers.

The first step in the classification is the use of the Gaussian kernel. All points in the feature space are implicitly mapped to a high dimensional space by using the Gaussian kernel

$$\begin{aligned} K(x_i, x_j) &= <\phi(x_i), \phi(x_j)> \\ &= e^{-\frac{1}{\theta}\|x_i - x_j\|^2} \end{aligned}$$

where $\phi : \mathbb{R}^d \rightarrow \mathscr{F}$ is a nonlinear map that takes the data into a higher dimensional space $\mathscr{F}$, that is also a dot product space. Here $<,>$ denotes the usual dot product of two vectors. In the description of the whole algorithm we will not need to compute either ϕ or $\mathscr{F}$ explicitly. All we would need is the definition $K(x_i, x_j) = e^{-\frac{1}{\theta}\|x_i - x_j\|^2}$ for our classifier which can be computed in $O(d)$ time in $\mathbb{R}^d$ [30].

In $\mathscr{F}$, we compute an implicit representation of the approximate center and the radius of the minimum enclosing ball for each class D_i. We now classify a point p to a class if $\phi(p)$ lies in the Voronoi cell of the approximate center calculated in the space $\mathscr{F}$.

We first look at the new approximation algorithm we designed for computing the minimum enclosing ball (MEB) of a set of points in $\mathscr{F}$ (see algorithm 11). Algorithm 11 is an extension of algorithm 10 for kernel spaces. Conceptually the algorithm is very simple. It maintains a vector $\mathbf{vec}\lambda = [\lambda_1, \lambda_2, ...]$, in which it implicitly maintains the center $c_i = \sum_{j=1}^{\chi} \lambda_j \phi(p_j)$. Here χ is the core set size [47]. In each iteration it finds the farthest point from the current center and updates the $\lambda_i's$ to reflect the movement of the center. The center in algorithm 11 moves exactly the way it moves in algorithm 10.

Algorithm 10 Approximate MEB Algorithm

Require: A point set $P = \{p[1], p[2], ..., p[n]\} \in \mathbb{R}^d$

1: $i \leftarrow \text{random}(1, n)$
2: Choose $p[j] \in P$ farthest from $p[i]$
3: Choose $p[k] \in P$ farthest from $p[j]$
4: $c_3 = \frac{1}{2}(p[j] + p[k])$
5: **for** $i = 3..iter$ **do**
6: Find farthest point $p \in P$ from c_i
7: $c_{i+1} \leftarrow (1 - \frac{1}{i+2})c_i + \frac{1}{i+2}p$
8: **end for**
9: Return c_{iter+1}

It was proven in [46] that for Euclidean spaces, if $iter = \frac{1}{\varepsilon^2}$ then algorithm 10 gives

a $(1+\varepsilon)$ approximate MEB. Since we need the MEB algorithm for kernel spaces we extended the algorithm to take into account the new mapping function ϕ. The approximate MEB algorithm needs to find the farthest point $\phi(p_j)$ from $\phi(p_i)$. This can be easily done by looking at

$$\begin{aligned}
&\max_{j=1..n}^{j\neq i} \|\phi(p_i)-\phi(p_j)\| \\
&\max_{j=1..n}^{j\neq i} <\phi(p_i)-\phi(p_j),\phi(p_i)-\phi(p_j)> \\
&\max_{j=1..n}^{j\neq i} <\phi(p_j),\phi(p_j)>-2<\phi(p_i),\phi(p_j)> \\
&\min_{j=1..n}^{j\neq i} K(p_i,p_j)
\end{aligned}$$

Similarly the function $\texttt{FarthestPoint}(\mathbf{vec}\lambda, P)$ that calculates the farthest point from the current center $c=\sum_{j=1}^{n}\lambda_j\phi(p_j)$ to the mapped points of P can be implemented without explicitly using $\phi()$. This can be done by calculating:

$$\begin{aligned}
&\max_{j=1..n} \|c-\phi(p_j)\| \\
&\max_{j=1..n} \|\sum_{i=1}^{n}\lambda_i\phi(p_i)-\phi(p_j)\| \\
&\max_{j=1..n} \sum_i\sum_k \lambda_i\lambda_k<\phi(p_i),\phi(p_j)> \\
&\qquad -2\sum_i \lambda_i<\phi(p_i),\phi(p_j)>+<\phi(p_j),\phi(p_j)> \qquad (1)\\
&\min_{j=1..n} \sum_i \lambda_i K(p_i,p_j)
\end{aligned}$$

If the dimension of the mapped space is finite, one can prove that algorithm 11 with $iter=\frac{1}{\varepsilon^2}$ gives a $1+\varepsilon$ approximate minimum enclosing ball [46]. But even for a bad

Algorithm 11 Approximate MEB Algorithm for Kernel Spaces

Require: A point set $P = \{p[1]1, p[2], ..., p[n]\} \in \mathbb{R}^d$

$i \leftarrow \text{random}(1, n)$
Choose $p[j] \in P$ farthest from $p[i]$
Choose $p[k] \in P$ farthest from $p[j]$
Swap$(p[1], p[j])$
Swap$(p[2], p[k])$
$\lambda_i = 0$ for $i = 3..n$
$\lambda_1 = \lambda_2 = \frac{1}{2}$
$\chi \leftarrow 2$
for $i = 3..iter$ **do**
  $ip \leftarrow$ `FarthestPoint`$(\mathbf{vec}\lambda, P)$
  if $ip > \chi$ **then**
    Swap$(p[\chi+1], p[ip])$
    $\lambda_{\chi+1} \leftarrow \frac{1}{i+2}$
  end if
  for $l = 1..\chi$ **do**
    $\lambda_l = (1 - \frac{1}{i+2})\lambda_l$;
  end for
  if $ip \leq \chi$ **then**
    $\chi \leftarrow \chi - 1$
    $\lambda_{ip} = \lambda_{ip} + \frac{1}{i+2}$;
  end if
  $\chi \leftarrow \chi + 1$
end for
Return $P, \mathbf{vec}\lambda$

approximation (say $\varepsilon = 0.05$), and finite dimensions, the iterations required are so huge for the algorithm that it becomes impractical to set $iter = \frac{1}{\varepsilon^2}$. For the purposes of classification, we found that setting $iter = 100$ in practice works quite well for Gaussian kernels.

Once the centers and radius of the mapped points are computed, we use a Voronoi diagram in the mapped space to do the classification. A point is classified according to

$$\min_{i=1..n} \|c_i - p\|$$

which can be again evaluated using kernel functions without mapping the points explicitly. This gives a decomposition of the mapped space that is used for classification. We do not compute this decomposition because the computation and storage of Voronoi diagrams in such higher dimensions is prohibitive. Instead we compute the nearest distant point using a linear scan of the centers in the mapped space. One could also think of using weighted Voronoi diagrams for classification using the radii of the centers computed. There is a slight difference when one needs the radius compared to only the center. The running time of the radius computation is cubic in terms of the number of iterations ($iter$) whereas the center can be computed in quadratic time in the number of iterations (see Equation 1).

The value of θ we used for the whole algorithm was 125. We first scaled all the feature vectors so that their range becomes comparable. This was done by first normalizing each feature to mean zero and variance 1 and then scaling the feature so that it lies between 0 and 100. This is very important for the classifier otherwise there might not exist a value for θ for which the classifier works. For scaled features, there is a wide choice of θ for which the classifier performs very well (see figure 56).

For verification purposes in our application we also needed to answer the question if a person is in the set of people that the database contains. The classifier classifies any person to a class, irrespective of whether the person is in the group of people who are in the

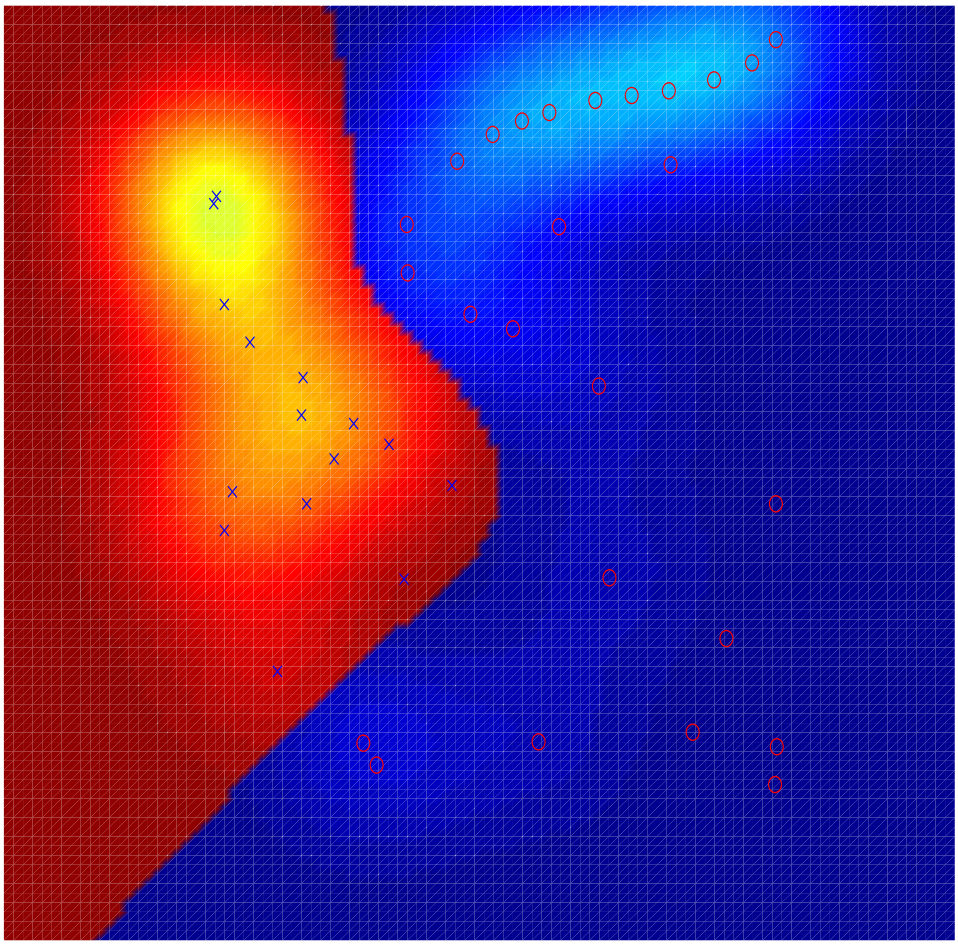


Figure 55: An example classification of a point set in $\mathbb{R}^2$ The figure color codes the distance from the respective centers.The lighter the color the nearer it is to the center.

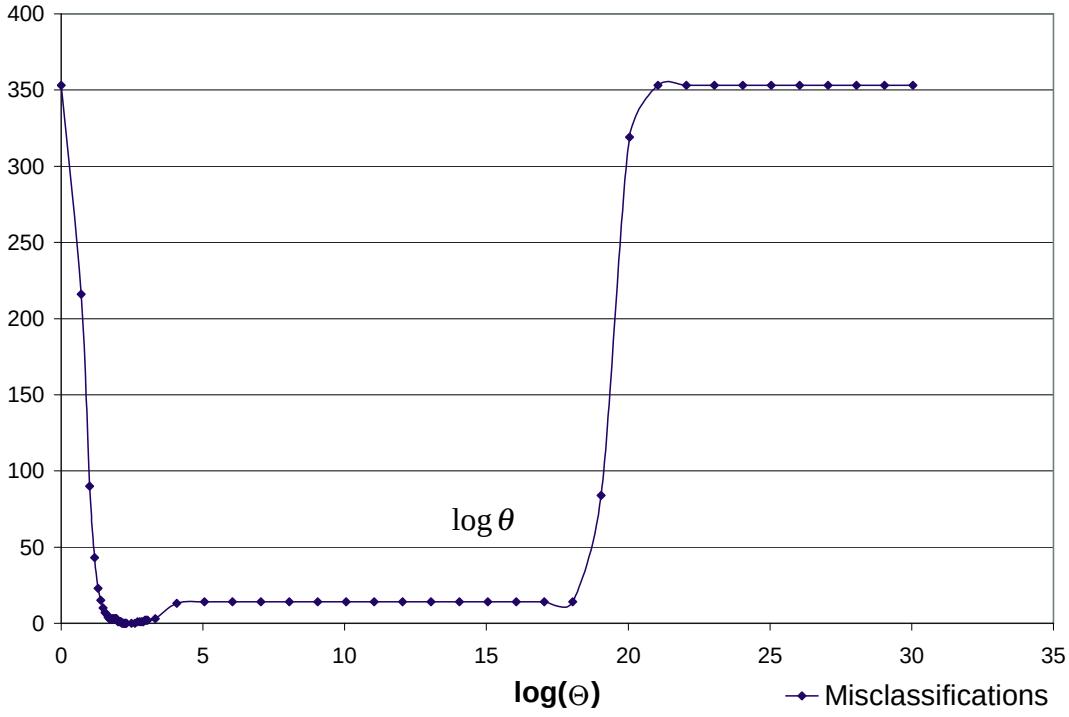


Figure 56: A graph of $\log\theta$ vs the number of misclassifications for a randomly chosen training set of 5 samples from each class(Averaged over 4 runs). For these four runs and for the range of theta between 140 and 400 the misclassification value was on average 0.75. Note that there is a wide choice of θ for which the number of misclassifications is small.

database or not. For verification, we used a verifier that answers the question if a feature vector v belongs to one of the classes D_i where i is output of the classifier when v is fed into it. This can be turned into an outliers detection question. Is v an outlier for D_i? We ran into numerical problems in trying to use off the shelf outlier detection methods [81]. For this we tried to use a standard outlier detection method, the *Mahalanobis distance*. A point is a γ^2 outlier if its Mahalanobis distance is greater than γ. If μ_{D_i} and Σ_{D_i} are the mean vector and covariance matrix for the class D_i, then v is considered an outlier if

$$\|v-\mu_{D_i}\|_M^2 = (v-\mu_{D_i})'\Sigma_i^{-1}(v-\mu_{D_i}) \geq \gamma^2$$

where

$$\Sigma_i = \frac{1}{2}(\Sigma_{D_i}+\Sigma^{pooled})$$
$$\Sigma^{pooled} = \frac{1}{N-c}\sum_{i=1}^{c}(|D_i|-1)\Sigma_{D_i}$$

Note that Σ_i are positive semi-definite matrices like covariance matrices since $x'\Sigma_{D_i}x \geq 0$ and $x'\Sigma^{pooled}x \geq 0$ imply

$$x'(\frac{1}{2}(\Sigma_{D_i}+\Sigma^{pooled}))x = \frac{1}{2}x'\Sigma_{D_i}x + \frac{1}{2}x'\Sigma^{pooled}x \geq 0$$

We found out that perturbing Σ^{pooled} matrix with Σ_{D_i} gave us better results than just using Σ^{pooled} for our verifier. The reason we did this was because the covariance matrix does not seem to be the same for all classes in our data.

7.7 Experimental results

We tested the nearest box and the MEB classifier for training set sizes of 3, 4, and 5. Experiments were repeated for randomly chosen training sets and results averaged. For

Training set size	Nearest Box	MEB	SVM
3	2.81	1.53	4.77
4	1.74	0.94	0.93
5	1.72	0.39	1.93

Table 2: Misclassification rates for classification. The SVM used is taken from PRtools Matlab toolbox [80] with its default settings except the kernel. The kernel used for SVM and MEB is Gaussian kernel with $\theta = 125$.

verification each set was queried by all the images. Figures 57 and 58 show the average plots of false acceptance rate and false rejection rate for different values of threshold. Figures 59 and 60 show the corresponding plots for identification. False acceptance rate (FAR) is defined as the percentage of images that were accepted incorrectly. In case of identification, acceptance with wrong identification is considered false acceptance. False rejection rate (FRR) is defined as the percentage of images that were rejected incorrectly.

Using training set size of 5, we achieve the following results keeping FAR below 1%. For verification we achieve an FRR below 3% using the nearest box classifier and below 2% using the MEB classifier. For identification we achieve an FRR below 6% using the nearest box classifier and below 5.5% using the MEB classifier.

Similarly we tested classification using random training sets. We also compared our classifiers to SVM using the same kernel function and found out that we were competitive to standard SVM at least for this application. Table 2 reports the average misclassification rates.

The average timings for both our MEB and Nearest Box classifiers were negligible (Less than 1 millisecond for verification/classification/identification on our test PC with 1.3 GHz Pentium IV and 512MB RAM).

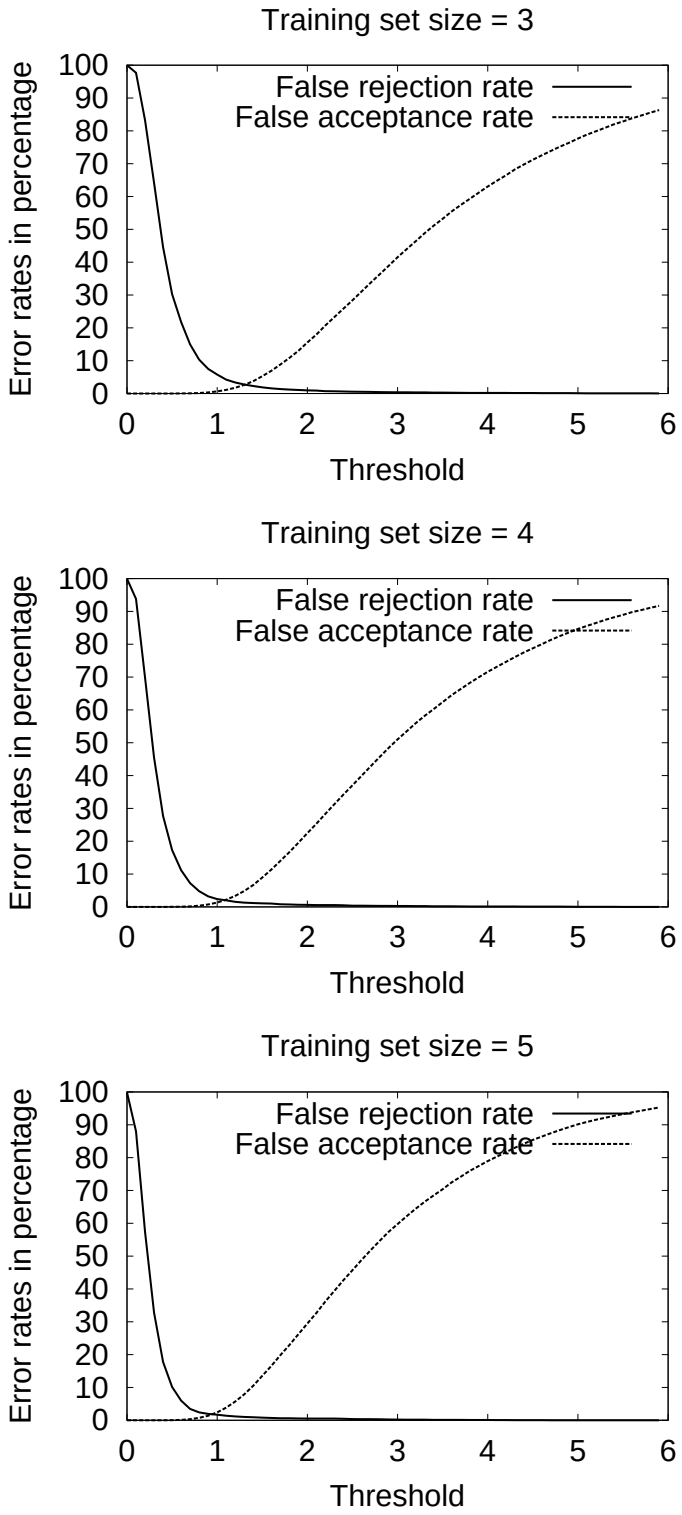


Figure 57: Verification error rates for nearest box classifier.

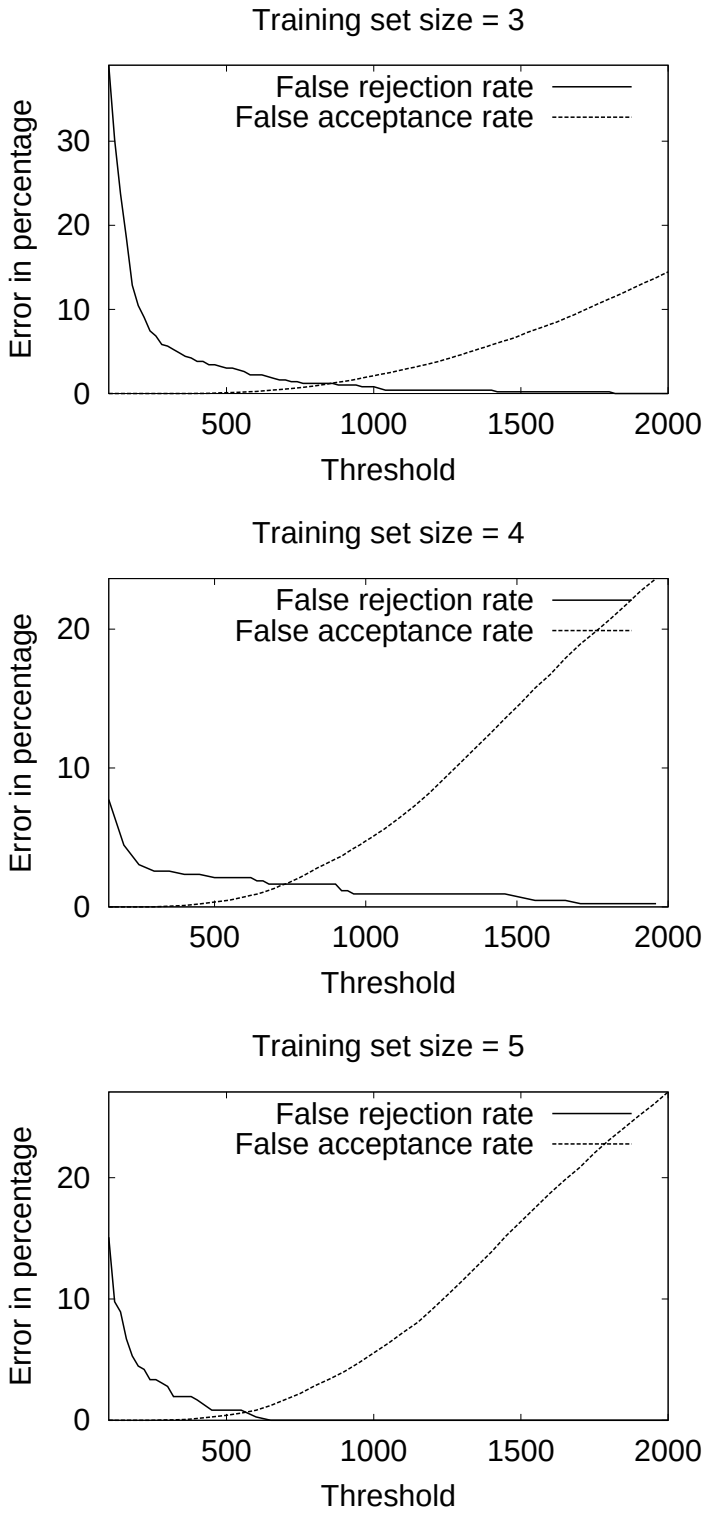


Figure 58: Verification error rates for MEB classifier.

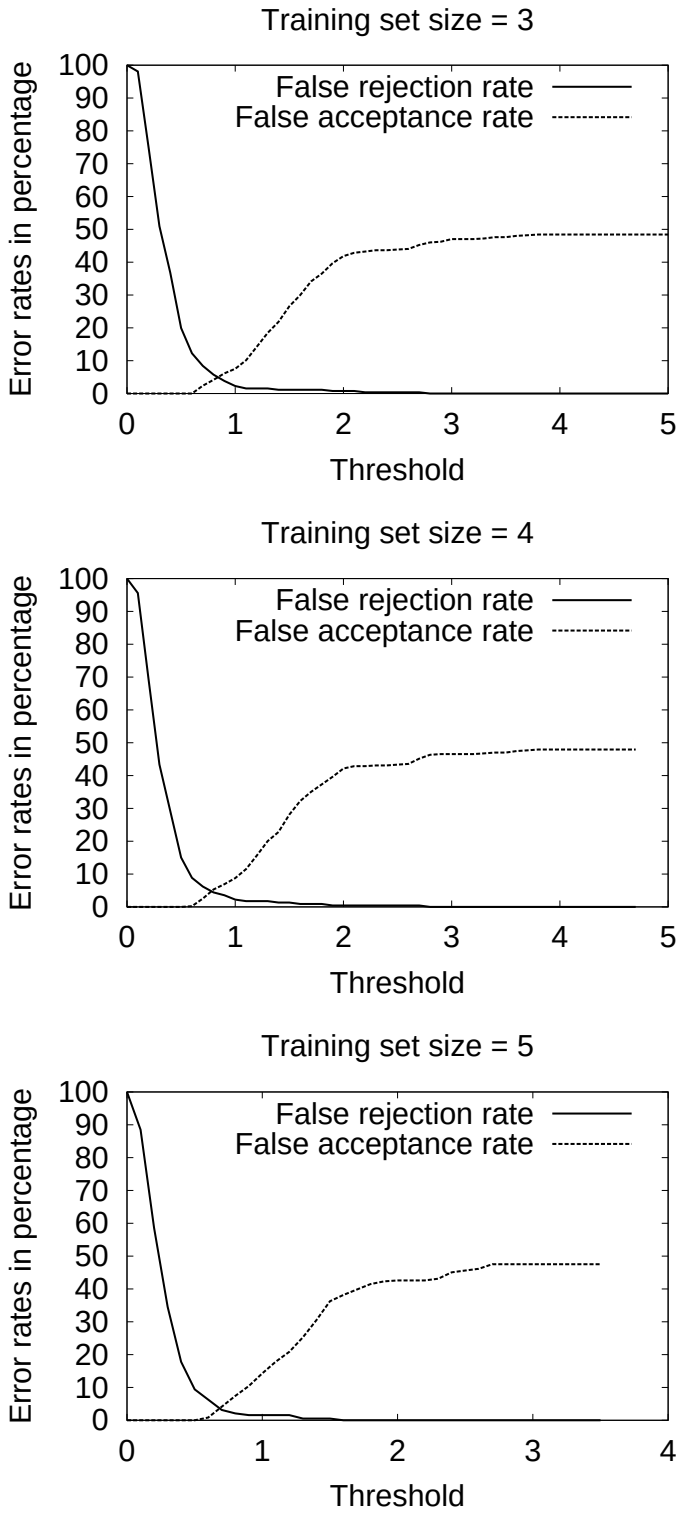


Figure 59: Identification error rates for nearest box classifier.

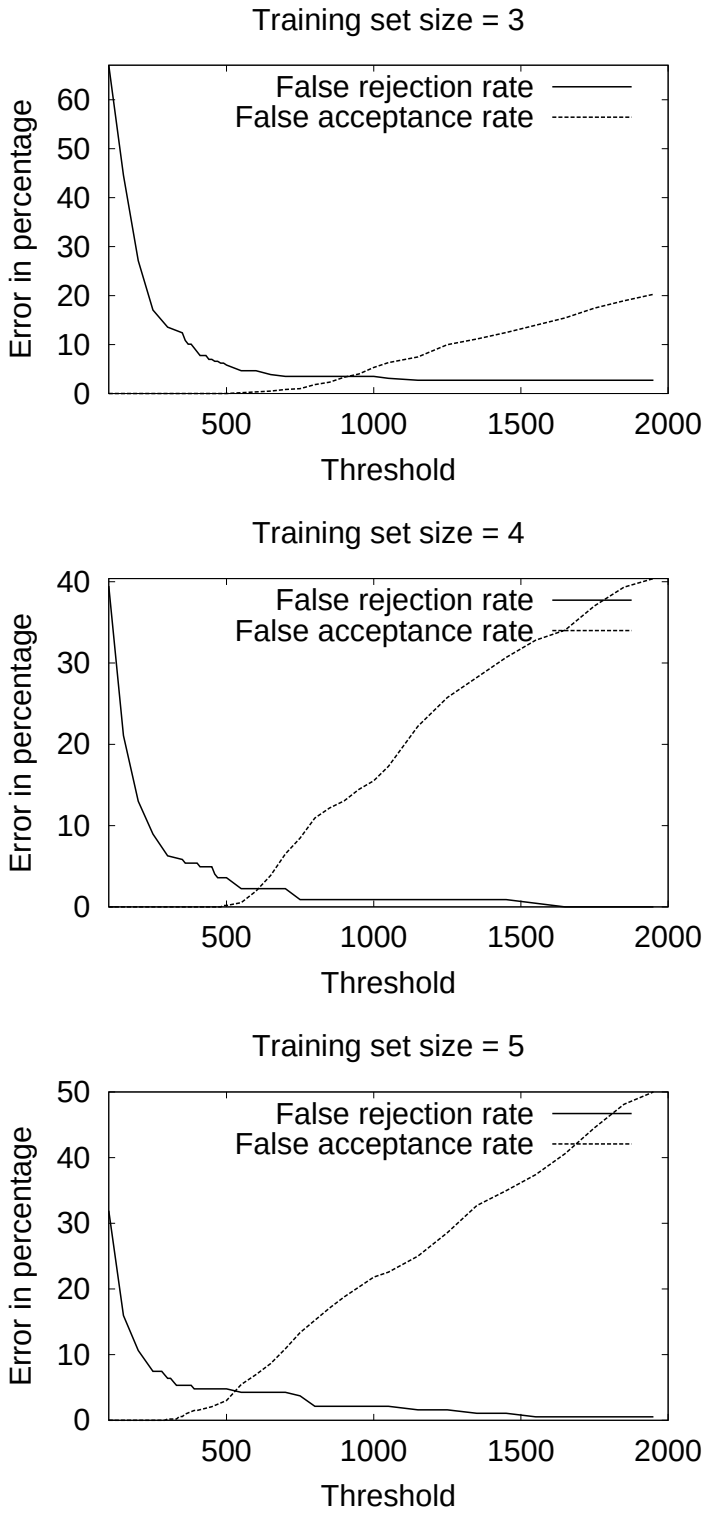


Figure 60: Identification error rates for MEB classifier.

7.8 Breast Cancer Detection

Breast cancer diagnosis is a challenging task. It can be treated as a pattern recognition problem [191, 192, 190]. Given a set of features and classes associated with those features, one can train the classifier to recognize characteristics about the features corresponding to each class. Recently, we have tried to compare the classification results of Minimum enclosing ball classifier (MEB) with existing training machines.

7.8.1 Materials and Methods

The Wisconsin breast cancer dataset was used [192]. The features were computed from a digitized image of a fine needle aspirate (FNA) of a breast mass and the characteristics of cell nuclei present in image were described. 30 real valued features such as mean area, radius etc was obtained from images. The dataset contained 569 (357 benign + 212 malignant) cases. The 30 features were normalized using Z-score normalization. It was classified using a variety of different classifiers like support vector machines (SVM), k-nearest neighbor, multilayer feed-forward neural network, linear discriminant, Fischer coefficient, Parzen window based classifier and MEB classifier. Each classifier was trained using 50% of the data randomly selected and the remaining 50% was used for testing. Mean accuracy, sensitivity and specificity for each classifier were obtained over 10 runs.

7.8.2 Results

The results showed in Table 3 were obtained by averaging the accuracy, sensitivity and specificity of each classifier over 10 runs.

	MEB	SVC	Neural	Parzen	Fischer	k-nn	ldc
Accuracy	95.96	91.75	96.81	95.79	95.82	94.60	95.72
Sensitivity	97.76	99.55	98.55	98.44	99.27	96.31	99.33
Specificity	92.92	78.58	93.87	91.32	90.00	91.70	89.62

Table 3: Classification Results using various classifiers.

7.9 Conclusions

We have developed a hand recognition system that uses hand geometry for verification, classification, and identification of individuals. We tested the system on a database of 714 hand images from 70 people. Our experiments clearly show that hand based recognition systems can be used for medium security applications. Stronger claims about such a system can only be made after conducting experiments at a larger scale. Our novel minimum enclosing ball classifier works well for hand recognition. Recently, we also got encouraging results for breast cancer detection using MEB classifiers.

7.10 Future Work

It remains to be seen where MEB classifiers could be applied. Outliers are a problem with any MEB clustering approach. MEB with outliers seems to be a natural problem to address in the future. There are many other interesting problems that remain unanswered in this research. Is there a way to design a classifier based upon MEBs that is PAC? Does the size of the core-set lead to better generalization results?

CHAPTER VIII

Finding large empty convex bodies

"There is no "royal road" to geometry."

EUCLID

Recent advances in data acquisition technology generate very large datasets that have millions of primitives. Rendering these datasets at interactive rates is a challenge in itself. Many methods have been presented to improve rendering speeds, including simple octree-based constructions, mesh simplification techniques, and use of currently available programmable graphics hardware [164, 138, 152]. Other methods include level-of-detail techniques and occlusion culling. Occlusion culling techniques utilize bounding volume hierarchies, hardware acceleration, and other methods [131, 65, 28, 107].

This chapter involves finding convex inner approximations of three-dimensional environments. We start by exploring similar problems in two dimensions before moving on to three dimensions. Convex shapes inside an object can serve as efficient occluders, as they may reduce the complexity of the shape being approximated by a considerable amount. These occluders can be used prior to rendering to remove large amounts of data that will not be visible in the final display, thus increasing rendering speeds.

8.1 Prior Work

Finding empty convex shapes is a well-studied problem in computational geometry. In two dimensions, algorithms have been studied for computing empty rectangles [55, 57, 71, 70], empty convex polygons [78], and the related problem of finding a maximum-length line segment ("biggest stick") within a polygon [3]. In three dimensions, algorithms have been developed to compute empty ellipsoids [125, 21, 198, 22] and empty cylinders [93]. Most of the results are on the theoretical front. Implementations have been done in graphics for finding good occluders in visibility preprocessing [131, 34].

Recent work by Daniel Cohen-Or *et al.* [63] finds the inner cover of non-convex polygons that function as occluders. Packing ellipsoids [36] in three-dimensional shapes has been done by Stephan Bischoff *et al.* [63]. Their algorithm proceeds by starting with a sphere inside the model and then gradually inflating and translating it until it gets pinned. Non-convex occluders [44], called *hoops*, have been studied by Brunet *et al.* [63]; they compute polylines that have convex silhouettes when seen from certain view points.

8.2 Overview

We present three main algorithms in this chapter. These algorithms are aimed at finding approximately largest empty triangles, ellipses and k-dops in closed objects under various assumptions.

The algorithm to compute a large empty triangle applies to a general simple polygon P in $\mathbb{R}^2$. The algorithm for finding a large empty ellipse applies to well-sampled smooth curves $\mathscr{S}$ in $\mathbb{R}^2$. Both of these algorithms come with some theoretical guarantees.

The algorithm for finding large empty (convex) k-dops is engineered to be practical and has no theoretical guarantees. It can find large empty k-dops inside polyhedral objects in

$\mathbb{R}^3$. The main idea of the algorithm is to "grow" *k*-dops within the environment, inflating them like balloons, until they get *pinned* and cannot grow any further. To check for intersections of the object with the environment during the inflation, we use QuickCD [126], a collision detection library developed at Stony Brook. Once we have an initial configuration of a pinned *k*-dop, we try to increase its volume by shifting outwards each of its facets, if possible.

Our choice of a convex object is a *k*-dop, a convex object whose facet's outward normals are taken from a fixed set of *k*-directions. This choice is partly because of the simplicity in handling *k*-dops for computation and partly because QuickCD uses *k*-dops internally for its collision detection routines.

The input to the algorithm is a set of triangles (the *environment*), specified as a list of vertices and triples of indices. Models with small cracks and holes can also be given as input. In particular, holes with maximum diameter less than the diameter of the largest inscribed sphere are not a problem. Similarly models with cracks can also be input as the cracks are generally small compared with the inscribed spheres. The output is a list of *k*-dops whose union covers most of the interior of the environment. These *k*-dops form the set of occluders that are used for accelerated rendering.

A number of variations and heuristics have been investigated. These include using different *k*-dops (varying *k*, and the normal vectors defining the *k*-dop), various seed point strategies, and various inflation techniques. Since a single *k*-dop will not, in general, completely fill the inside of the model, we have also developed an algorithm designed to *pack* the environment with non-intersecting *k*-dops.

After packing an environment with *k*-dops, in order to reduce the effect of local maxima, we apply an algorithm designed to merge neighboring *k*-dops. This reduces the number of total *k*-dops and also finds a better convex inner approximation.

8.3 Maximum-Area Triangles

In this section we will develop a constant factor approximation algorithm for finding a maximum-area empty triangle in a simple polygon, assuming a restriction on the size of the optimal answer.

Given a simple polygon P, the problem of finding a maximum-area empty triangle inside the polygon can be solved in polynomial time, but the known algorithms do not seem to be either easy to code or practical.

For this problem we will assume, without loss of generality, that the area of the polygon P is 1. We make the following additional assumption about P: We assume that the area of a maximum-area triangle is at least ε, for some fixed $\varepsilon > 0$. In this case, a constant factor approximation algorithm is easy to obtain, as follows. Pick a uniform sample, s, of $O(\frac{1}{\varepsilon^2}\log(\frac{1}{\varepsilon}))$ points within the polygon P and then output a maximum-area triangle in the visibility graph of s. (Such a triangle necessarily lies within P, since P is simple.) We now sketch an argument that the resulting triangle is already a constant factor approximation for a maximum-area empty triangle, τ^*, with some positive probability. By VC-dimension arguments, the random sample is an ε-net. With a large enough constant in the big-Oh estimate of the sample size, we can make the probability that there are four or more sample points within τ^* be at least some fixed positive probability. Now one can show that if we pick four points uniformly at random from τ^*, then, with positive probability, the area defined by the largest triangle defined by three of the four points is at least one fourth of

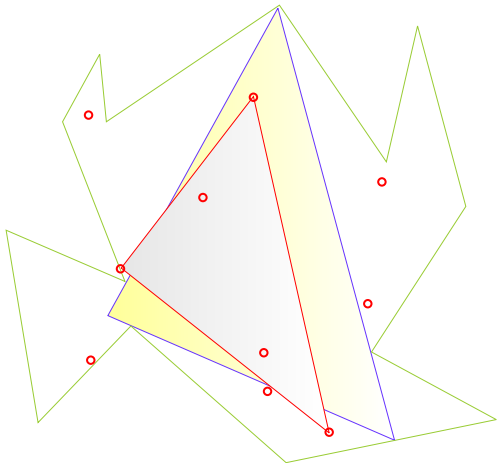


Figure 61: An illustration of our algorithm at work for computing an approximate maximum-area triangle.

the area of τ^* (see figure 61).

Now, a uniform sampling of P can be done by first triangulating P, then choosing a triangle of the triangulation with probability proportional to its area, and then uniformly sampling a point from inside the triangle. This can be done in time $O(n+s)$, where n is the number of vertices of P. The total number of triangles defined by the s samples is $O(s^3)$. An emptiness query can be implemented in $O(\log n)$ time using ray-shooting data structures [90], since we only have to check that each of the three sides of a triangle does not intersect the boundary of P. Hence, the total running time of the algorithm is $O(n+s^3 \log n)$.

We can get a slight improvement in the running time, at the expense of complicating the algorithm some. We can compute the visibility graph of all of the sample points inside the polygon in time $O(n+s\log s\log ns+k)$, where k is the size of the resulting visibility graph [31]. (Note that $k \in O(s^2)$ in the worst case, but k could be much smaller.) Computing a maximum-area triangle given the visibility graph can be done in $O(s^3)$ time by

checking each candidate triangle to see if all three edges lie in the visibility graph. This reduces the total running time to $O(n+s^3+s\log s\log ns)$.

8.4 Maximum-Area Ellipses

In this section we examine the problem of finding a maximum-area inscribed ellipse in a well-sampled smooth closed curve. We show that, as the sampling rate increases, our solution converges to an exact solution. We also show implementation results of a simplified version of the algorithm.

Let $\mathscr{E}_{opt}$ denote a maximum-area empty ellipse contained in the curve $\mathscr{S}$, and let a (resp., b) denote the length of the major (resp., minor) axis of $\mathscr{E}_{opt}$. The *feature size* of a point on the curve is defined to be the distance from the point to the closest point of the medial axis. Let $\varkappa$ be the minimum feature size of $\mathscr{S}$. Without loss of generality, we will assume that $\mathscr{S}$ lies inside a circle of diameter 1; hence, we can assume that $\varkappa < 1$. Let $\varepsilon_1, \varepsilon_2 > 0$ be positive constants that are very small compared to 1.

Definition 1 *A curve $\mathscr{S}$ is said to be* ε-sampled *if the length of the curve between any two consecutive sample points is no more than ε.*

Assumption: $\varepsilon \leq \min\{\varepsilon_1 b, \varepsilon_2 \varkappa\}$.

Note: This assumption also guarantees that $\varepsilon \leq \min\{\varepsilon_1, \varepsilon_2\}$.

Let $\mathscr{E}_B$ be the ellipse obtained from $\mathscr{E}_{opt}$ by doing the following transformation: Scale (enlarge) $\mathscr{E}_{opt}$ until it hits a sample point p_1. Continue scaling the ellipse, but now impose the constraint that it passes through point p_1; let p_2 be the next sample point it hits. Now, keeping the ellipse in contact with both p_1 and p_2, continue enlarging the ellipse, while increasing both of its axes, until the ellipse hits a third point, p_3. If the triangle $\Delta p_1 p_2 p_3$

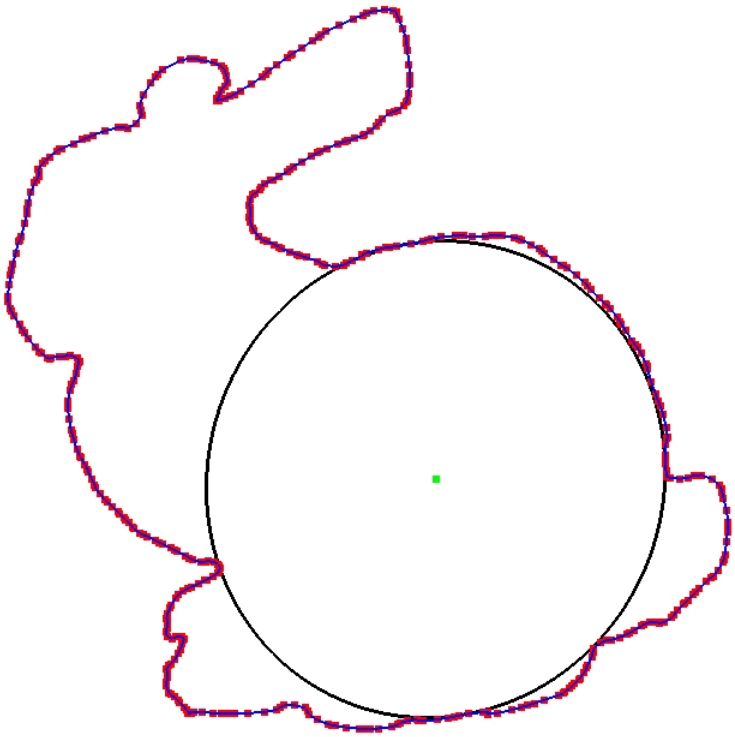


Figure 62: A maximum-area ellipse inside a projection of the bunny.

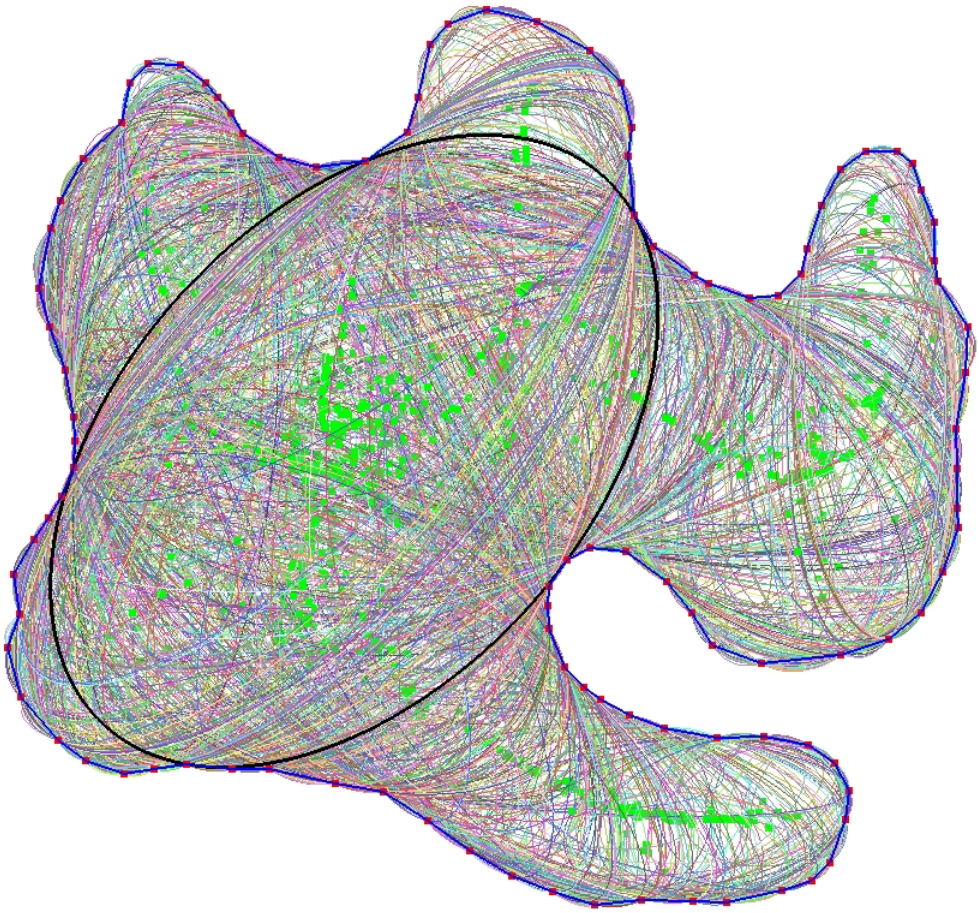


Figure 63: The set of all *feasible* ellipses for a given sampling of the boundary curve. The (dark) black ellipse is the maximum-area ellipse inside the sampled curve.

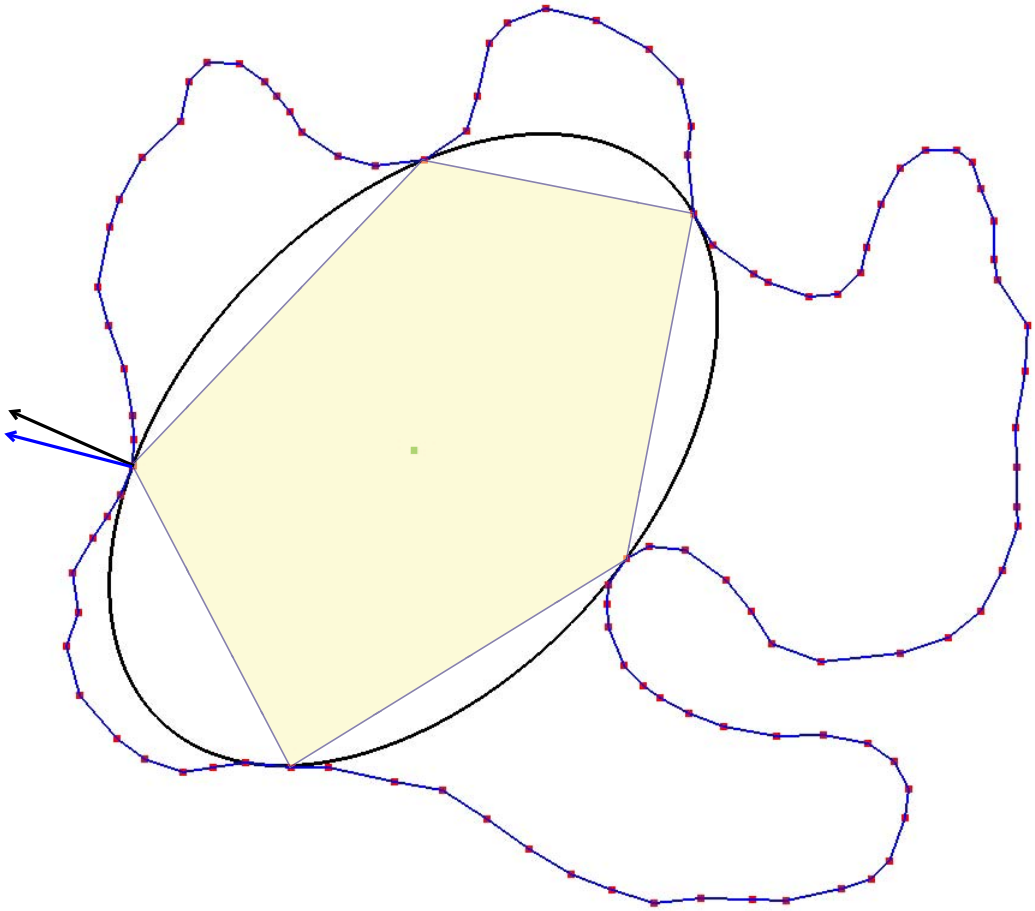


Figure 64: The maximum-area ellipse drawn inside the sampling.

contains the center of the current ellipse, then we output $\mathscr{E}_B$. Otherwise, the current ellipse can be translated away from its contact with the three points p_1, p_2, and p_3. We then restart the enlarging process again, and continue until we find an ellipse $\mathscr{E}_B$ that is supported by three sample points, p_1, p_2, and p_3, with $\Delta p_1 p_2 p_3$ containing the center of the ellipse.

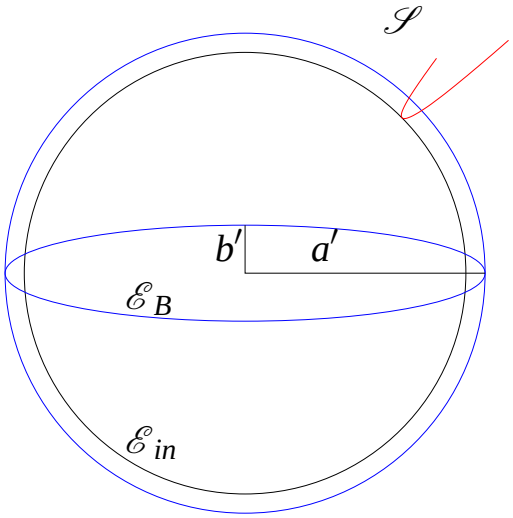


Figure 65: The Kepler circle of $\mathscr{E}_B$.

Lemma 8.4.1 *Let $\mathscr{E}_{in} = (1 - \frac{\varepsilon}{b})\mathscr{E}_B$. Then Ar $\mathscr{E}_{in} \leq$ Ar $\mathscr{E}_{opt} \leq$ Ar $(\frac{b}{b-\varepsilon})\mathscr{E}_{in}$, where Ar $\mathscr{E}$ denotes the area of ellipse $\mathscr{E}$.*

Proof. The process of inflating $\mathscr{E}_{opt}$ to $\mathscr{E}_B$ means that Ar $\mathscr{E}_B \geq$ Ar $\mathscr{E}_{opt}$ and hence Ar $\mathscr{E}_{opt} \leq$ Ar $(\frac{b}{b-\varepsilon})\mathscr{E}_{in}$. Note that the interior of $\mathscr{E}_B$ is free of sample points. Let the radii of $\mathscr{E}_B$ be $a' > a$ and $b' > b$. Now, using the definition of sampling, $\mathscr{S}$ cannot penetrate $\mathscr{E}_B$ more than ε, since $\mathscr{E}_B$ is free of sample points. Scale the space such that $\mathscr{E}_B$ becomes a circle with radius a'. (Such a circle is also known as the *Kepler circle* of $\mathscr{E}_B$; see figure 65.) Hence, in the rescaled space, $\mathscr{S}$ cannot penetrate the Kepler circle of $\mathscr{E}_B$ by more than $\frac{a'\varepsilon}{b'}$, since distances are stretched in this space by at most $\frac{a'}{b'}$. Let $\mathscr{E}'_{in} = \left(1 - \frac{\varepsilon}{b'}\right)\mathscr{E}_B$. By

the above argument, $\mathscr{E}'_{in} \subseteq \mathscr{S}$ and hence $\mathrm{Ar}\,\mathscr{E}'_{in} \leq \mathrm{Ar}\,\mathscr{E}_{opt}$. Also, $b' \geq b$ implies that $\left(1-\frac{\varepsilon}{b}\right) < \left(1-\frac{\varepsilon}{b'}\right)$, which means that $\mathscr{E}_{in} \subseteq \mathscr{E}'_{in}$, completing the proof. □

The main idea of the algorithm is to determine an ellipse that is close in some sense to $\mathscr{E}_B$; then, by the above lemma, this implies that the ellipse is also close to $\mathscr{E}_{opt}$.

Next, we bound the angle between the normal of $\mathscr{E}_B$ and the normal to the curve $\mathscr{S}$ at each of the three points p_1, p_2, p_3. Towards this goal, we prove the following lemma about an arbitrary sample point p_i:

Lemma 8.4.2 *If $\mathscr{E}_B$ and $\mathscr{S}$ both pass through the sample point p_i, then the angle between their normals at p_i is $O(\varepsilon)$.*

Proof. Let p_{i-1} and p_{i+1} be the sample points before and after p_i. It can be shown using our sampling criterion that the angle spanned by $p_{i-1}p_ip_{i+1}$ is greater than $\pi - 4\arcsin(\varepsilon/2)$ (see Lemma 10, [19]). A simple calculation shows that the normal to $\mathscr{S}$ at p_i and the normal to the edge $p_{i-1}p_i$ make an angle of at most $O(\varepsilon)$. If $\mathscr{E}_B$ were arbitrarily large, then this would already prove that the angle between the normals to $\mathscr{S}$ and to $\mathscr{E}_B$ at p_i is $O(\varepsilon)$; however, the curvature of $\mathscr{E}_B$ adds a small angle to the deviation, so we now show that this deviation is small.

We need to bound the angle between the normal to the ellipse $\mathscr{E}_B$ and the normal to the segment $p_{i-1}p_i$. We also know that $|p_{i-1}p_i| \leq \varepsilon$. Without loss of generality, assume that $\mathscr{E}_B$ is axis-aligned for this proof. Then the equation of $\mathscr{E}_B$ is

$$\frac{x^2}{a'^2} + \frac{y^2}{b'^2} = 1.$$

Note that as ε decreases, the angle between the normal to $\mathscr{E}_B$ at p_i and the normal to the segment $p_{i-1}p_i$ decreases. The maximum deviation of the normal occurs when the ellipse

passes through either $p_{i-1}p_i$ or $p_{i+1}p_i$; thus, assume that $\mathscr{E}_B$ passes through $p_{i-1}p_i$. We now need to calculate the maximum angle between the normal of the ellipse at p_i and the normal to $p_{i-1}p_i$. Let $\tan(\theta')$ be the slope of the line passing through the center of the ellipse and p_i. Let 2δ be the angle subtended by $p_{i-1}p_i$ with respect to the center of the ellipse, so that p_{i-1} subtends an angle $\theta' + 2\delta$ on the x-axis. It is not hard to show that the maximum angle subtended by $p_{i-1}p_i$ is achieved at $\theta' = \pi/2$. A simple calculation shows that $\delta_{max} = O(\frac{\varepsilon}{b})$. Let us sweep the ellipse $\mathscr{E}_B$ with a sector of angle δ_{max}. Let the chord subtended by δ_{max} be $p_{i-1}p_i$. Note that this only enlarges $p_{i-1}p_i$ and hence increases the deviation in the normal. Since we want to bound the deviation from above, we can afford to do this. It is not hard to show that the slope of such a chord is $-\frac{b'}{a'}\cot(\theta' + \delta_{max})$, and the slope of the tangent at p_i is $-\frac{b'}{a'}\cot(\theta')$. Using the fact that $b' >> \varepsilon$ and $\cos\delta \geq 0.5$, one can show that the angle between the slopes above is bounded by $O(\delta_{max})$. Hence, the total deviation added by the curvature of the ellipse is $O(\varepsilon)$. This implies that the the angle between the normals of $\mathscr{E}_B$ and $\mathscr{S}$ at p_i is $O(\varepsilon)$. See figure 64 for an illustration. □

We are now ready to show the following lemma.

Lemma 8.4.3 *Let p, q, and r be three sample points on the boundary of $\mathscr{E}_B$. Any ellipse passing through p, q, and r whose center is contained in triangle Δpqr, whose normals at p, q, and r are close to $\mathscr{E}_B$, and whose area is greater than Ar $\mathscr{E}_B$ cannot have its smaller radius of $O(\varepsilon)$.*

Proof. Without loss of generality, assume that $\mathscr{E}_B$ has the equation

$$\frac{x^2}{a'^2} + \frac{y^2}{b'^2} = 1.$$

The proof proceeds by contradiction. We will assume the existence of an ellipse $\mathscr{E}'$ with small radius $c_1\varepsilon$, passing through p, q, and r, with center contained in Δpqr and Ar $\mathscr{E}' \geq$

Ar $\mathscr{E}_B$. Let l_1 and l_2 be a pair of parallel lines on either side of the origin, at the same distance, $2c_1\varepsilon$, from the origin. Refer to figure 66.

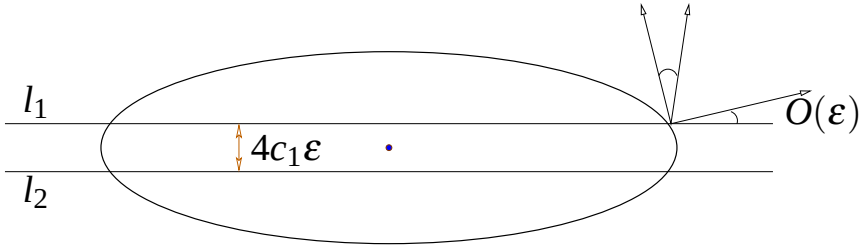


Figure 66: An illustration for the proof of Lemma 8.4.3.

Since $\mathscr{E}'$ contains the origin, we can align l_1 and l_2 in such a way that the slab they define contains $\mathscr{E}'$. Note that the large radius of $\mathscr{E}'$ is greater than $\frac{a'b'}{c_1\varepsilon}$, which is very large compared to the extent of $\mathscr{E}_B$ intersecting the slab defined by l_1 and l_2. Let δ_p be the angle between the normals of $\mathscr{E}_B$ and $\mathscr{E}'$ at p. Then,$\max(\delta_p, \delta_q, \delta_r)$ is minimized when l_1 and l_2 are parallel to the x-axis, assuming the distance between l_1 and l_2 is small compared to a' and b'. Note that, in any other orientation, one of the δ_p, δ_q, or δ_r will be obtuse.

If we fix l_1 and l_2 to be parallel to the x-axis, it is not hard to show that the angle subtended by the cone of all normals of $\mathscr{E}_B$ lying inside the slab determined by l_1 and l_2 is $O(\varepsilon)$, and the axis of this cone is aligned with the x-axis. The cone that contains all of the normals of $\mathscr{E}'$ is also small, but its axis is aligned with the y-axis.

Since both of these cones are well separated, the angle between them cannot be $O(\varepsilon)$, and, thus, the normal condition will be violated at the points of intersection of $\mathscr{E}_B$ and $\mathscr{E}'$. Hence, we obtain a contradiction to the assumption that $\mathscr{E}'$ can have a very small radius. □

Consider the conic defined by the equation

$$f(\mathbf{x}) = q_{11}x^2 + 2q_{12}xy + q_{22}y^2 + r_1x + r_2y + f = 0. \tag{67}$$

The above expression may also be written in matrix form as

$$f(\mathbf{x}) = \mathbf{x}^t Q\mathbf{x} + R\mathbf{x} + 1 = 0, \tag{68}$$

using the assumption, without loss of generality, that $f = 1$.

Note that Q is symmetric and that $f(\mathbf{x})$ represents an ellipse if and only if Q is positive definite. The area of the ellipse is directly proportional to $\det Q^{-1}$. Now we are ready to present the approximation algorithm; refer to Algorithm 12.

The first step in the algorithm is *linearization* [90]. In general, an ellipsoid in d dimensions admit a *linearization* of dimension $d + \frac{d(d+1)}{2}$; thus, for our purposes, we lift to $\mathbb{R}^5$. The complexity of the convex hull $\mathfrak{C}$ of the linearization L' is, in the worst case, $O(n^2)$. Note that any conic in $\mathbb{R}^2$ in the form of equation 67 becomes a half-plane in $\mathbb{R}^5$ after linearization. Also note that a half-space in $\mathbb{R}^2$ maps to a half-space in $\mathbb{R}^5$ after linearization.

After the linearization step, the next step (Step 3) of the algorithm is to examine each 2-simplex (triangle) of $\mathfrak{C}$ to determine there is an ellipse through its three vertices that is almost contained in $\mathscr{S}$; we want to maximize the area among such ellipses. Note that each 2-simplex is adjacent to two 3-simplices and at most five 4-simplices; each 3-simplex is adjacent to two 4-simplices. Each of the neighbors of a 2-simplex is obtained in constant time, assuming we have any standard adjacency representation of the simplices in $\mathfrak{C}$.

The next step in the algorithm is to parse each 2-simplex $\tau \in \mathfrak{C}$ and solve a corresponding optimization problem on τ. The number of such 2-simplices is $O(n^2)$. Once we fix a τ to process, we know that the ellipse we seek passes through the vertices of τ (in $\mathbb{R}^2$). Let the vertices be p, q, and r. Then we have the relations in Step 4 of the algorithm. These three relations help us to reduce the problem in 2-dimensional space, $((r_1, r_2))$. We can now represent the equation of a conic in terms of $R = (r_1, r_2)$. Now we solve the optimization problem in Step 5, which computes an ellipse $\mathscr{E}_r$ passing through p, q, and r, whose

Algorithm 12 Algorithm for computing an approximate maximum-area inscribed ellipse.

Require: A point set $L = \{p[1], p[2], ..., p[n]\} \in \mathbb{R}^2$

1: $L' = \{(x^2, y^2, \frac{1}{2}xy, x, y) | (x,y) \in L\}$

2: $\mathfrak{C} = \text{ConvexHull}(L')$

3: **for all** $\tau_{p,q,r} \in \mathfrak{C}$ such that $\tau_{p,q,r}$ is a 2-simplex. **do**

4: Using the relations

$$p^t Q p + R p + f = 0 \tag{69}$$
$$q^t Q q + R q + f = 0 \tag{70}$$
$$r^t Q r + R r + f = 0 \tag{71}$$

represent $q_{11} = \psi_{11}(R)$, $q_{22} = \psi_{22}(R)$ and $q_{12} = \psi_{12}(R)$. Note that $\psi_{11}(R), \psi_{22}(R)$ and $\psi_{12}(R)$ are all linear in $R = \begin{pmatrix} r_1 & r_2 \end{pmatrix}$.

5: Solve the following optimization problem

$$\min A_{\tau_{p,q,r}} = \psi_{11}(R)\psi_{22}(R) - \psi_{12}^2(R)$$

subject to

$$\begin{aligned} \psi_{11}(R)\psi_{22}(R) - \psi_{12}^2(R) &> 0 \\ \psi_{11}(R) &> 0 \\ \psi_{22}(R) &> 0 \end{aligned}$$

The ellipse hyperplane supports $\mathfrak{C}$

The ellipse satisfies the normal constraints at p, q, r

The center of the ellipse lies inside Δpqr.

The small radius of the ellipse $\geq c\varepsilon$.

6: **if** The problem is infeasible **then**

7: $A_{\tau_{p,q,r}} = \infty$

8: **else**

9: Let $\mathscr{E}_r$ be the ellipse corresponding to the solution of the optimization problem.

10: **end if**

11: **end for**

12: Return $\mathscr{E}_{alg} =$ Max area $\mathscr{E}_r$ found in step 3.

normals at p,q,r each make a small angle with the corresponding normals of $\mathscr{S}$, whose center is contained in the triangle Δpqr, and whose small radius is at least $c\varepsilon$ for some constant much larger than 1. But since the ellipse has to match the normal and the triangle condition, its not hard to see that the ellipse cannot have all of the sample points inside it; otherwise, the normal condition is violated. Now we have an ellipse $\mathscr{E}_r$, which has all points of L outside or on it, passes through p, q, and r, and has *maximal* area.

The algorithm just returns the area that is maximum over all areas computed for feasible ellipses. We still need to show that the optimization problem can be solved in constant time to prove a running time of $O(n^2)$. Before we delve into that, we show the approximation guarantee.

Theorem 8.4.1 *Algorithm 12 returns an ellipse $\mathscr{E}_{alg}$ such that*

$$(1-\frac{1}{2c})Ar\,\mathscr{E}_{alg} \leq Ar\,\mathscr{E}_{opt} \leq Ar\,\mathscr{E}_B \leq Ar\,\mathscr{E}_{alg}.$$

Proof. Note that $\mathscr{E}_{alg}$ passes though three points of L and has no sample points inside it. Its small radius is at least $c\varepsilon$, where c is a constant much larger than 1. (The existence of such a c is guaranteed by Lemma 8.4.3.) We also know that $\mathscr{E}_B$ is a feasible solution of the optimization problem that we solve using the algorithm, and the algorithm maximizes the area; hence, $\mathrm{Ar}\,\mathscr{E}_{alg} \geq \mathrm{Ar}\,\mathscr{E}_B \geq \mathrm{Ar}\,\mathscr{E}_{opt}$. Since $\mathscr{E}_{alg}$ is free of sample points, $\mathscr{S}$ can penetrate it by a distance of at most $\frac{\varepsilon}{2}$, and hence if we shrink $\mathscr{E}_{alg}$ by a factor of $(1-\frac{1}{2c})$, it will be totally inside $\mathscr{S}$ and hence $(1-\frac{1}{2c})\mathrm{Ar}\,\mathscr{E}_{alg} \leq \mathrm{Ar}\,\mathscr{E}_{opt}$, proving the lemma. □

Theorem 8.4.2 *The running time of Algorithm 12 is $O(n^2)$.*

Proof. The main time taken by the algorithm is to examine each 2-simplex of $\mathfrak{C}$ and solve the optimization problem. We will show in the next section that the optimization problem

in line 5 of the algorithm can be solved in constant time. This readily implies that the running time of the algorithm is $O(n^2)$. □

8.4.1 The Optimization Problem

We first examine the constraints. The constraint

$$\psi_{11}(R)\psi_{22}(R) - \psi_{12}^2(R) > 0$$

is clearly a quadratic constraint in two dimensions, since $\psi_{ij}(R)$ is linear in R. The constraints of the type $\psi_{ij}(R) > 0$ are trivially linear. The constraint that the hyperplane corresponding to the ellipse in $\mathbb{R}^5$ should have $\mathfrak{C}$ on one side of it can be written as a linear constraint. Note that there are 5 simplices incident on the 2-simplex under consideration; each one of these five simplices has 2 vertices in addition to p, q, and r. So if we make our ellipse equation have all of these 10 points on one side, this will automatically mean, by convexity arguments, that the ellipse we get passing through p, q, and r supports $\mathfrak{C}$ on one side. Thus, we put each of these 10 points into equation 67 and require that all of the 10 expressions have the same sign. Note that since q_{11}, q_{22} and q_{12} are each a function of R, each of these equations is a linear constraint in R. To determine the sign that we need to use, we can use the centroid of Δpqr and use the opposite sign for all the 10 points.

The condition that the normal to the ellipse and the normal to the curve should not be large is a quadratic constraint. At a sample point p', the normal to the desired ellipse is of the form

$$\left(\frac{2q_{11}p'_x + 2q_{12}p'_y + r_1}{\varsigma}, \frac{2q_{22}p'_y + 2q_{12}p'_x + r_2}{\varsigma} \right), \tag{72}$$

where

$$\varsigma = \sqrt{(2q_{11}p'_x + 2q_{12}p'_y + r_1)^2 + (2q_{22}p'_y + 2q_{12}p'_x + r_2)^2}.$$

We can use the normal estimate at the sample points, take the dot product with the vector in equation 73, and bound it with $1-\ O(\varepsilon)$. (This is true assuming ε is small, since we can then approximate $\cos(\varepsilon)$ with $1-\varepsilon$.) Hence, the constraint becomes

$$\left(\frac{2q_{11}p'_x+2q_{12}p'_y+r_1}{\varsigma},\frac{2q_{22}p'_y+2q_{12}p'_x+r_2}{\varsigma}\right).(n^{\mathscr{S}}_{p'_x},n^{\mathscr{S}}_{p'_y})\leq 1-O(\varepsilon). \tag{73}$$

This can be written as a conic constraint by squaring the equation and clearing the denominator. Hence, the normal constraints can be implemented as a quadratic constraint in R.

The constraint that the center of the ellipse lies inside triangle Δpqr can also be written as a quadratic constraint in R. The equation of the center of the conic given in equation 68 is given by solving the linear system $Q\mathbf{x_c}=-R$. Let us suppose that the half-space passing through pq and containing r has equation $h_1x+h_2y+h_3\geq 0$. Note that we already know the values of h_1,h_2 and h_3 since our simplex is fixed and we know p,q,r. Now this half-space should contain $\mathbf{x_c}$ and hence $h_1\mathbf{x_c}_x+h_2\mathbf{x_c}_y+h_3\geq 0$. Here,

$$\begin{bmatrix}\mathbf{x_c}_x\\ \mathbf{x_c}_y\end{bmatrix}=\frac{1}{q_{11}q_{22}-q_{12}^2}\begin{bmatrix}q_{11} & -q_{12}\\ -q_{12} & q_{22}\end{bmatrix}\begin{bmatrix}r_1\\ r_2\end{bmatrix}. \tag{74}$$

Substituting the formula for $\mathbf{x_c}$ into the half-space inequality results in a constraint that can be made quadratic in R using the substitution for q_{ij} as a linear function of R. This substitution of q_{ij} as a linear function of R means that we are only interested in ellipses that pass through p, q, and r and have their centers inside Δpqr.

Similarly, the last constraint is again a quadratic constraint, since it bounds the inverse square root of the eigenvalues of Q. What we want is the constraint $\frac{1}{\sqrt{\lambda_1}}\leq\frac{1}{\sqrt{\lambda_2}}\leq c\varepsilon$. Since q_{11} and $q_2 2$ are constrained to be positive, the smaller eigenvalue of Q is given by

$$\lambda_2=\frac{1}{2}\left((q_{11}+q_{22})-\sqrt{(q_{11}+q_{22})^2-4(q_{11}q_{22}-q_{12}^2)}\right)\geq c\varepsilon.$$

Hence, the constraint is a quadratic constraint in q_{ij} and hence a quadratic constraint in R.

Each of the quadratic constraints that we have derived can be written as a conic constraint in two dimensions. If we assume that our input consists of rational numbers, we can compute the exact arrangement of the conics in the plane using a modification of the Bentley-Ottmann sweep paradigm; efficient implementations are also available for these purposes in the CGAL library [49]. Note that what we need is only one (possibly non-convex) cell of this arrangement in the plane. Let us denote this cell by C. Note that this cell is of constant complexity and can be computed in constant time, since the total number of constraints is constant (we are assuming that the dimension is a constant here).

Once we have such a cell, what we want to do is to minimize a conic in three dimensions subject to its projection lying in the cell that we computed. Let us call the 3rd axis, the z-axis. We are not interested in anything with $z < 0$. (Since, $z = f(r_1, r_2) = \psi_{11}(R)\psi_{22}(R) - \psi_{12}^2(R)$, this follows from our first constraint.) The first thing we do is to compute the intersection of our conic that we want to optimize with the $z = 0$ plane. Let us call this curve on the plane $\mathfrak{z}$. If $\mathfrak{z} \cap \mathsf{C} \neq \phi$ we declare the problem to be infeasible (since $z = 0$ is attained in this case). Otherwise, either the minimum of f is attained in the interior, *int* C, or on the boundary of C. If the minimum is achieved in the interior, $\nabla f(r_1^{opt}, r_2^{opt}) = 0$ and $r_1, r_2 \in$ *int* C (which is easy to solve). If this is not the case, then the optimum is achieved on the boundary of C. We can independently solve the problem on each of the curved segments or linear segments and output the minimum feasible solution in this case. Each of these curved segments can be parametrized as a single variable function. One can do a case analysis of corresponding constraints to the curved segment to determine if the curved segment comes from an ellipse, hyperbola, parabola or a line. One can then get these conic sections in standard form, without changing the minimum of f. If the curved

segment is an ellipse, we can use the parametrization $(a\cos t, b\sin t)$ and represent f as a fourth-degree polynomial in $\cos t$ and solve the problem on the arc by finding the roots of the polynomial. A similar situation happens if we use the parametrization $(a\sec t, b\tan t)$ of the right branch of the standard formula for a hyperbola. In this case, we get a polynomial of fourth degree in $\tan t$. For a parabolic arc of a standard parabola, we could use the standard parametrization $(t, \sqrt{4at})$, which results in a quadratic polynomial that is easy to solve for a minimum. Essentially what we are doing is lifting each curved arc segment of C on $f \geq 0$ and solving the minimization problem on the projected curve (which turns out to be easy).

8.4.2 Implementation

We implemented a simple algorithm based on Algorithm 12. The algorithm we implemented can be made to fail on contrived examples but usually works well in practice for many input datasets. It relies on the observation that for a dense enough sampling, *usually* there exists an ellipse comparable in area to $\mathscr{E}_B$ passing through 5 sample points. In this case we can just linearize the problem, and then for each ellipse passing through the 4-simplices, check feasibility by testing (1) the normal condition, (b) the condition that the ellipse center lies inside the convex hull of the supporting points, and (3) the fatness of the ellipse. The implementation outputs the ellipse that has maximum area over all feasible ellipses. An example run of the implementation is shown in figure 63. The implementation is written in C++ and uses qhull for the convex hull computation in $\mathbb{R}^5$ [26].

We have seen how to use a linearization of 2-dimensional ellipses to find the largest area ellipse inside a well-sampled smooth curve. Although one could try to generalize this algorithm to higher dimensions, its running time would be exponential in the dimension d.

Even for $d = 3$, it seems that the linearization step is too slow to be practical. Thus, we have developed an alternate approach, based on maximal k-dops, which we describe in the next section.

8.5 Computing Empty k-dops

We now describe a method for computing large empty convex bodies based on growing special types of convex polytopes known as "k-dops" within a geometric domain. This research was done in collaboration with Uday Chebrolu and Joseph Mitchell.

Our algorithm has a number of components to it: finding good seed points, expanding the k-dop, and merging k-dops. Our approach for each of these subproblems is explained in the following sections, after we introduce k-dops.

8.5.1 Convex Bodies – k-dops

Our choice of convex bodies is a "k-dop". A *k-dop* is a convex polytope whose facets are determined by half-spaces with outward normals taken from a fixed set of k distinct directions. A bounding k-dop gives an approximation to the convex hull of a body. An axis-aligned bounding box is a special case of a k-dop; as k increases, if the set of k outward normals is "well distributed", then a bounding k-dop approaches the convex hull. Also, k-dops have the advantage that intersection tests and other computations are relatively fast, making them a good choice for collision detection methods based on bounding volume hierarchies [126]. figure 67 shows a (2-dimensional) 4-dop and an 8-dop of the silhouette of the Stanford bunny model.

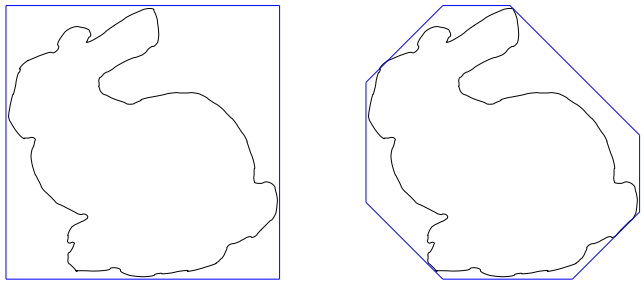


Figure 67: An illustration of a bounding 4-dop and a bounding 8-dop of the silhouette of the Stanford bunny model.

8.5.2 Finding Seed Points

Average and Random Locations: The naive method used was to place the seed point at the center of environment. The seed point is checked whether it is inside the environment and if so it is placed there. The second method was to randomly generate a seed point and check whether it is inside the environment, and, if so, place it there. In case the generated location is outside the environment, another point is generated repeatedly until one that is inside is generated. This method is also used in the naive method when the initial center point is outside the environment.

Approximate Nearest Neighbors: The input vertices are added to the Approximate Nearest Neighbors (ANN) [144] algorithm, where they are preprocessed into a data structure to facilitate fast (approximate) nearest neighbor queries. A 3-dimensional grid of points is placed over the input environment with the grid size being a variable factor (finer grid for better sampling). These grid points form the set of query points for ANN. The algorithm finds the query point that has the furthest nearest neighbor amongst all of the query points, and this point is used as a seed point. This is similar to placing the seed point at the center of the largest sphere that can be inscribed in the input environment amongst

all of the points in the grid. Ideally, we would like to compute the seed point taking the distance between the seed point and the polygons into consideration, but since evaluating this distance is not trivial, we use the distance between the polygon vertices and the seed point as an approximation.

8.5.3 Expanding a *k*-dop

Increase volume by scaling: A *k*-dop is placed centered at the seed point that has been generated earlier. This *k*-dop is scaled by repeatedly doubling the scaling factor. Once it collides with the environment or when it encloses the environment, the scaling factor is linearly decreased until the first time it does not collide with the environment. The method is illustrated in figure 68. This position is stored as an inner approximation. Further experimentation showed that instead of finding the largest scaling factor that does not induce a collision with the environment, a better and faster way is to use the halved scaling factor that induced a first collision. This gives us a *k*-dop that is potentially smaller than the earlier method but is better suited for further *Expansion by Sliding* which is described in 8.5.3.2.

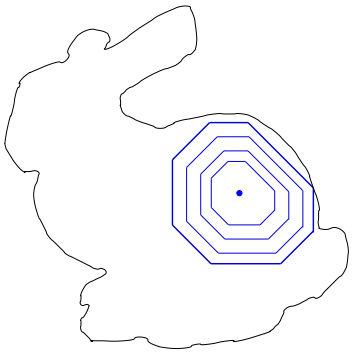


Figure 68: An example of expansion by scaling a *k*-dop.

8.5.3.1 Increasing volume by translation

Once scaling of the *k*-dop was done, it is translated if possible away from the point of contact with the environment and scaled again. This is similar to a balloon expansion when it hits a surface and moves away from it while still expanding. The method is illustrated in figure 69. The introduction of ANN for seed point generation made expansion by translation unnecessary, as not much increase in volume was observed.

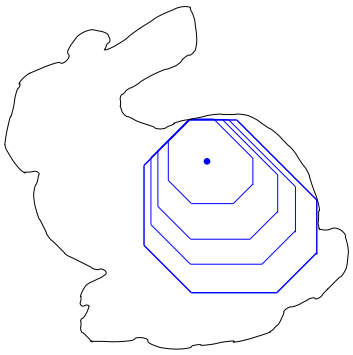


Figure 69: Expansion by translating and scaling the *k*-dop.

8.5.3.2 Increasing volume by sliding

This is the method that results in maximum increase in volume. As stated earlier a *k*-dop is a convex polytope bounded by half-spaces of a given orientation. Initially, the *k*-dop is scaled at a seed point generated by ANN.

After scaling, the algorithm loops over all the facets of the *k*-dop and for each facet, the corresponding half-space is moved (*slided*) in the same direction until either of the following happens. The resulting *k*-dop collides with the environment or until the half-space has no effect on the *k*-dop. These two cases are shown in fig 70.

The complete algorithm is illustrated in figure 71. The seed point generated by ANN

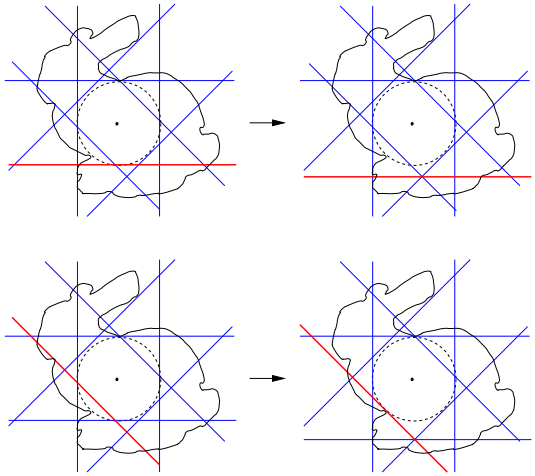


Figure 70: Two cases in which sliding half-spaces is terminated.

is at the center of the circle. The configuration of the *k*-dop after scaling is shown by the intersection of the inner most red lines. The lines are looped in an anti-clockwise fashion starting with the red line with orientation (1,0). This red line is moved to the corresponding green line. The next line that is moved is the line with orientation (1,1). This is continued until all the 8 lines are done and a new loop is started. This process is continued until no half-space can be moved further. The final result is shown as a shaded region.

8.5.4 Packing *k*-dops

To pack a given number of *k*-dops or to pack *k*-dops until a certain percentage of the volume is covered, the algorithm is repeatedly called after some updates to the QuickCD data structures and to the ANN data structures. Specifically, the already computed *k*-dops are added to the environment. This is done to detect collision with the already computed *k*-dops.

The seed points for the subsequent *k*-dops are obtained from ANN, by finding the center of the next largest empty sphere. ANN is updated in a similar fashion by adding the grid

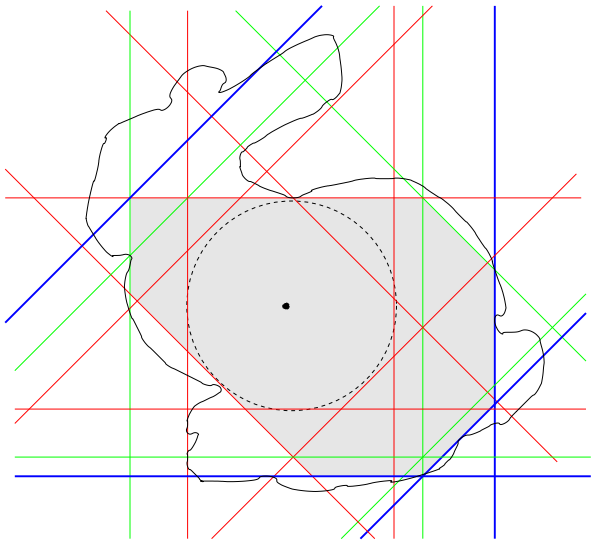


Figure 71: Expansion by sliding the half-spaces.

points that lie inside the computed *k*-dops to the environment. An example of packing 5 18-dops in the bunny is in figure 72.

8.5.5 Merging *k*-dops

Once we obtain a packing of *k*-dops, it was observed that in long thin regions of the environment, a number of adjacent *k*-dops were being computed instead of a long *k*-dop. We would like to have a minimal number of *k*-dops for use in occlusion. Hence to reduce the number of *k*-dops, merging of the *k*-dops was considered. This involves finding the initial packing of *k*-dops and then merging adjacent *k*-dops. Another method is to find an initial sphere packing of the environment using ANN. This is very fast as it involves only nearest neighbor queries. Consider two adjacent spheres in this configuration. A seed *k*-dop that has the shape of a long cuboid (a single triangle would also be okay) covering the two centers of the spheres is placed and the *k*-dop expansion techniques are employed. This

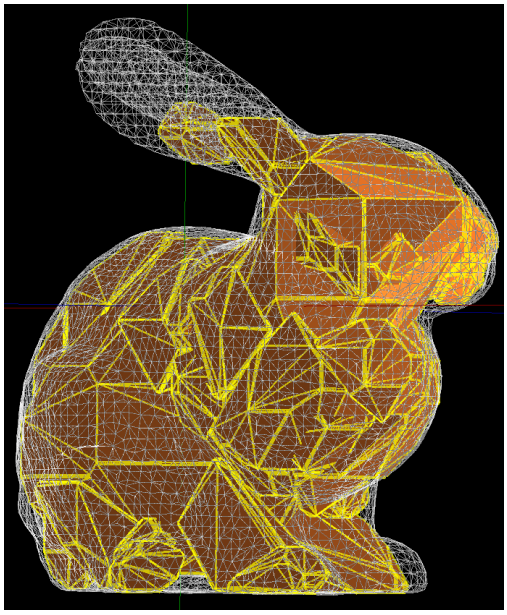


Figure 72: A packing of the bunny model with disjoint 18-dops.

resulted in a *k*-dop that covers the space occupied by two spheres. Among these two methods, the later was preferred as the first method involves computation of a large number of *k*-dops which makes it slower compared with the second method.

In the second method a question that arises is which two or more spheres to merge. The centers of the spheres in the sphere packing approximately lie on the medial axis of the model. Merging spheres that lie in the same section of the medial axis is preferable. This might involve merging more that two spheres. The merging we implemented is straight forward in that it finds the largest sphere and then finds its largest neighboring sphere. These two spheres are merged using the second method described above.

Merging of *k*-dops involves methods that haven't been completely investigated. A method that would give us fewer *k*-dops is preferable. It would also be better if the initial computation itself results in *k*-dops that are similar to the merged ones. This will avoid

the further step of merging. Merging of *k*-dops needs further study and experimentation.

8.5.6 Implementation and Results

The algorithms were implemented in Gnu C++ 3.2.2 and the graphical user interface was done in fltk. Two libraries, Approximate Nearest Neighbor(ANN) and QuickCD were used. The input to the algorithm is a set of triangles (the environment), specified as a list of vertices and indices. The output is a list of *k*-dops whose union covers most of the interior of the environment. The input data is preprocessed and added to the QuickCD environment hierarchy. The input vertices are also passed to ANN to build its hierarchy. Additionally, vertex normals are extracted for inside/outside tests. Most of the project was done on a laptop with Pentium III 1GHz processor, 512MB RAM, 25GB hard disk space.

8.5.6.1 k-dop Representation

k-dops are represented as a set of *k* half-spaces. Additionally, vertices on the convex hull are also stored along with the *k*-dop. *k*-dop scaling is done by scaling the vertices of the convex hull and computing its bounding half-spaces. Similarly, *k*-dop translation is done by translating the convex hull vertices and scaling them again. *k*-dop sliding is done by moving each half-space and repeatedly checking for collisions using QuickCD.

8.5.6.2 QuickCD

QuickCD [126, 128, 127] is a collision detection library written by Jim Klosowski, Martin Held and Joseph Mitchell. It takes as input an environment and a flying object and reports collision details between them. The main idea is to build two bounding volume hierarchies one for the environment and the other for the flying object. The bounding volume is chosen to be a *k*-dop. By using *k*-dops, fast intersection tests can be performed while still having

a tight enough bounding volume. To test for collision, the two hierarchies are recursively traversed until they reach the lowest primitives (triangles) and then report the final result, whether there is a collision or not.

In our implementation, the input triangles are sent to QuickCD to be built as an environment hierarchy and the seed *k*-dop is built into a flying object hierarchy. Whenever the *k*-dop is modified during expansion, the old flying object hierarchy is deleted and a new hierarchy is built. Since this hierarchy can be as small as a single node, performing this operation is very fast and can be repeated frequently. Collision detection is then performed on the two hierarchies. In the case of packing *k*-dops, the environment hierarchy has to be modified every time a new *k*-dop is found. This is required to treat the already computed *k*-dops as part of the environment, thus aiding in collision detection.

8.5.6.3 Approximate Nearest Neighbors

Approximate Nearest Neighbors (ANN) [144, 25, 24] is a library which supports exact and approximate nearest neighbor searching in arbitrary dimensions implemented in C++ by David Mount and Sunil Arya. The input data is a set of points which are processed into a data structure. Given a query point, the program can return its nearest neighbor and/or its *k* nearest neighbors. Put in another way, the distance to the nearest neighbor from a query point is the radius of the largest empty sphere centered on the query point. This is used in our algorithm to find large empty spheres whose centers form good seed points.

Input triangle vertices are added to and processed into the ANN hierarchy. The bounding box of the environment is sampled into a regular grid of specified resolution. Nearest neighbor queries are performed with the grid points as query points and the sample point with the furthest nearest neighbor is made the seed point. Additional inside/outside tests

Dataset	# Vertices	Time for 18-dop(sec)	Time for 26-dop(sec)
Missile	1176	6.6	3.73
Beethoven	2655	1.91	10.7
Dragon	5205	1.97	6.48
Porsche	5247	1.6	4.78
Buddha	7108	1.98	5.26
Bunny	8171	2.55	7.98
Tool	10166	3.17	16.51
Horse	48485	4.85	7.17

Table 4: Timings for computation of largest 18-dops and 26-dops.

have to be performed on the query point. This utilizes the normals evaluated at the input vertices.

During packing k-dops, after each additional k-dop is computed, the sample points that lie inside this k-dop are added to the ANN hierarchy. This is necessary for two reasons, one being not to include these points in the set to query points and secondly, it takes care that the new seed point is far from both the k-dop vertices and also its facets.

8.5.6.4 Memory Requirements

The total memory consumption is a sum of memory consumptions of QuickCD, ANN and our program. QuickCD uses $(16k+108)n$ bytes for the environment, where k is dependent on our choice of k-dop and n is the number of input triangles. Since our flying object contains a single k-dop, its memory consumption can be taken as a small constant. Refer [128] for further analysis of QuickCD's memory consumption.

ANN uses $O(n\log n)$ space. Refer [25] for further details. Our program additionally computes normals for each vertex which takes $36n$ bytes. In packing k-dops, the computed k-dops are stored in a std::vector. Each k-dop takes $8k$ bytes. Thus if the packing yields p number of k-dops, the total memory consumption is $((16k+144)n+8kp)$bytes and the storage for ANN.

8.5.6.5 Experiments

The algorithm has been tested on various datasets some with input normals and the others without them. The two main tasks that were performed were to find the single largest *k*-dop and to pack the given object with a minimal number of large *k*-dops. Two *k*-dops, 18-dops and 26-dops were used as basic convex bodies.

Normal Estimation and Inside/Outside Tests. The datasets come in two formats, with and without normals. In the case when the normals are given, they are used for inside/outside tests. Some of the datasets had normals only for select vertices; in such a case the rest of the normals were computed. We use a simple method for normal estimation: compute the normal of the facets of the environment and then for a particular vertex, average the normals of the facets that share this vertex. This can be made a little bit better by taking the area of the facet into consideration.

To perform inside/outside tests, the nearest neighbor is found using ANN, and the dot product of the vector from the query point to its nearest neighbor and the normal of the nearest neighbor is evaluated. A negative value indicates that the query point is outside the environment. To increase the robustness of this method, this test is performed with the *k*-nearest neighbors and a vote is taken among them.

Volume Estimation. In order to measure the effectiveness of our algorithm, we compare the volume covered by the union of all *k*-dops with the volume covered by the environment object. The bounding box of the environment is sampled on a regular grid and we use the ratio of number of points that lie inside the union of all *k*-dops with the number of points that lie inside the environment. The same grid used for finding seed points using ANN is used again for volume estimation.

Dataset	# Vertices	Normals	# 18-dops	Time(sec)	% Coverage
Beethoven	2655	yes	15	18.57	58
Dragon	5205	no	17	28.72	39
Porsche	5247	yes	10	22.25	66
Buddha	7108	no	39	59.13	60
Bunny	8171	no	45	62.51	72
Horse	48485	no	10	55.65	54

Table 5: Percentage of volume covered by the union of *18-dops* and the time taken to compute them.

Dataset	# Vertices	Normals	# 18-dops	Time(sec)	% Coverage
Beethoven	2655	yes	13	27.63	61
Dragon	5205	no	33	167.0	52
Porsche	5247	yes	4	33.16	55
Buddha	7108	no	10	41.7	47
Bunny	8171	no	40	223.47	75
Horse	48485	no	12	102.4	55

Table 6: Percentage of volume covered by the union of *26-dops* and the time taken to compute them.

8.5.6.6 Results

Our algorithm has performed reasonably well in finding large convex bodies (*k*-dops) in three dimensional environments. In datasets that have a considerable empty space, the algorithm was always able to find it as shown in bunny and buddha. The bunny dataset also has holes which were avoided by the algorithm. We were also able to detect thin long convex shapes that were in one of the directions of the *k*-dop facets as shown in the missile 74 and tool 73 renderings.

Since the algorithm greedily finds the largest inscribed sphere and uses its center as the seed point, packing of the environment completely becomes a problem. The small and narrow features are not covered as the inscribed sphere at those locations will be small. This is shown in rendering of the horse 75, where its body, neck and head were easily

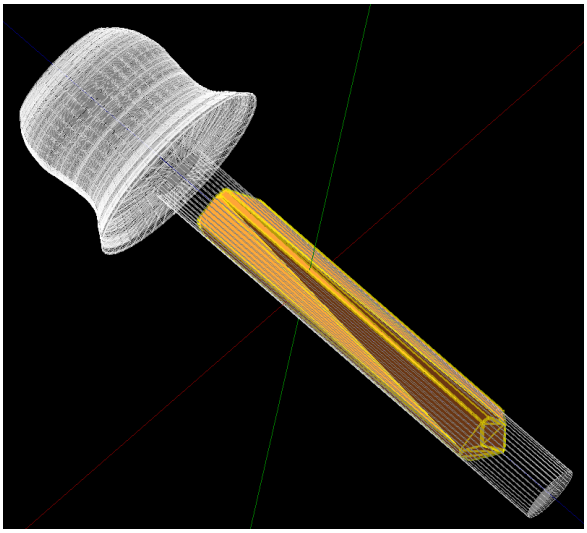


Figure 73: Tool.

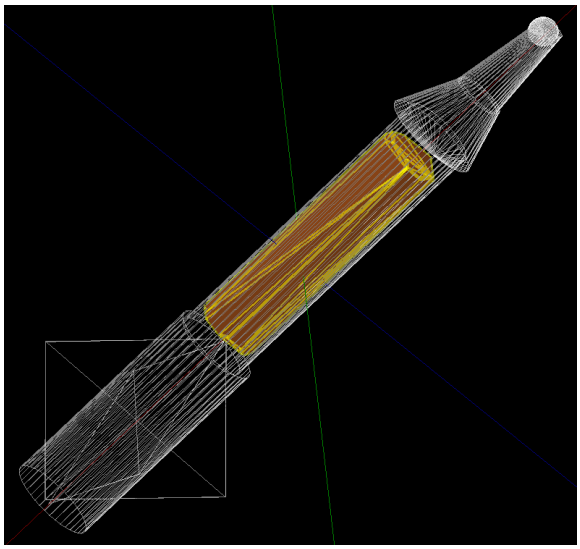


Figure 74: Missile.

computed, but the legs are still not found after using 12 26-dops.

The tables 5 and 6 show the timings and percentage of volume covered by the union of 18-dops and 26-dops for some datasets. Packing of *k*-dops ceases after the program is unable to find anymore valid seed points. As an example in table 6, the *Dragon* dataset was packed with 33 26-dops and covered 52% of volume. Further increase in volume coverage was not possible as valid seed points were not found. This is a consequence of our inside/outside tests which rely on normal computation which is not accurate. This is a serious problem and has to be corrected to improve the performance of the algorithm.

The graphs 76 and 77 show the increase in percentage of volume covered as a result of increase in the number of 18-dops. It can be observed that the increase is more for the earlier *k*-dops and gets smaller as more and more *k*-dops are added.

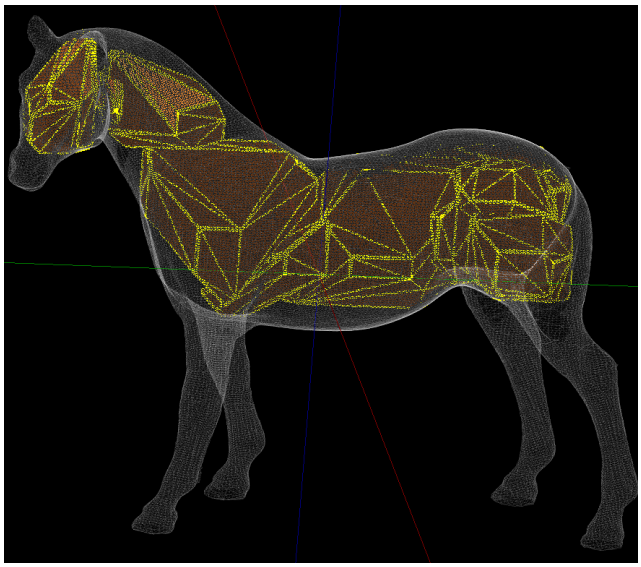


Figure 75: Packing of *k*-dops in the horse dataset. Note the lack of *k*-dops in the legs due to the difficulty in finding seed points in narrow areas.

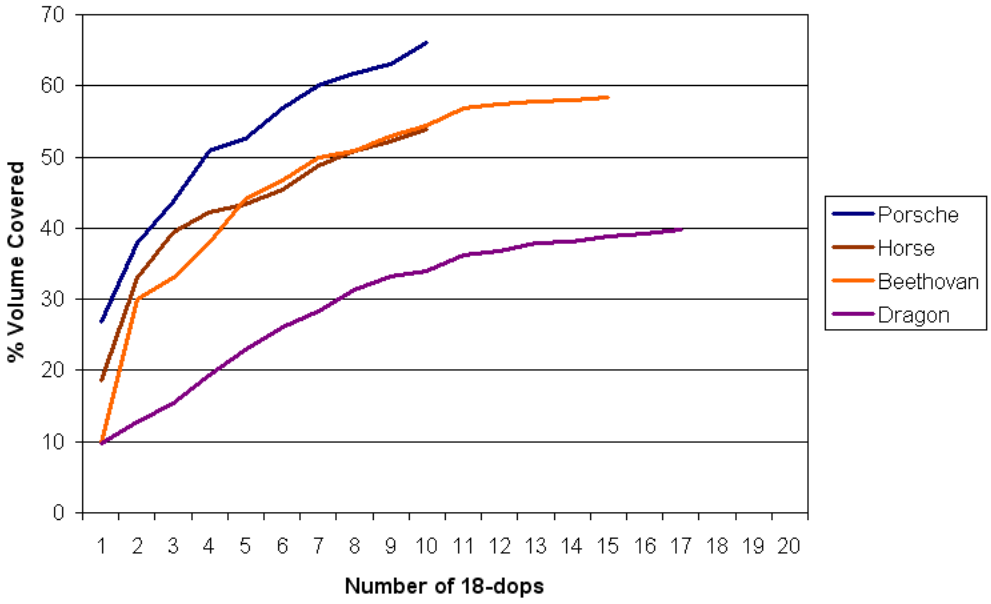


Figure 76: Increase in percentage of volume covered with increase in number of 18-dops.

8.5.6.7 Problems with Normals

In finding seed points, inside/outside tests have to be robust. Since the method we use relies on surface normals, incorrect normals produce seed points that are actually outside the environment. Further pruning of seed points is necessary to avoid incorrect solutions. One method is to check whether the *k*-dop that is computed intersects with the *k*-dop of the whole environment. If so that seed point and the corresponding *k*-dop are rejected. This process also wastes time as *k*-dops are expanded only to be found later that they intersect with the *k*-dop of the environment and pruned. In datasets that have pointed features, the algorithm fails to compute a good solution as most of the normals are incorrect at the pointed features.

8.5.7 Conclusions and Future Work

We have shown that simple techniques like translation, scaling and moving the facets of *k*-dops are able to find good convex approximations of the interior of environments. Datasets that have a good sampling rate with less pointed features are good candidates for our algorithm. It is useful to cover large areas with *k*-dops. In general small features will be missed, or an unnecessarily large number of *k*-dops have to be computed to cover the small features. The algorithm is reasonably fast for medium sized datasets up to 50,000 vertices. As convex approximations, *k*-dops have performed well and further increasing in complexity of convex objects seems unnecessary.

We have faced a number of problems in the process and these are potential topics that need further research and investigation. The first problem that arises is that of normal estimation. We started with the aim of not using normals in our computations, but for implementing inside/outside tests robustly they became necessary. More robust methods

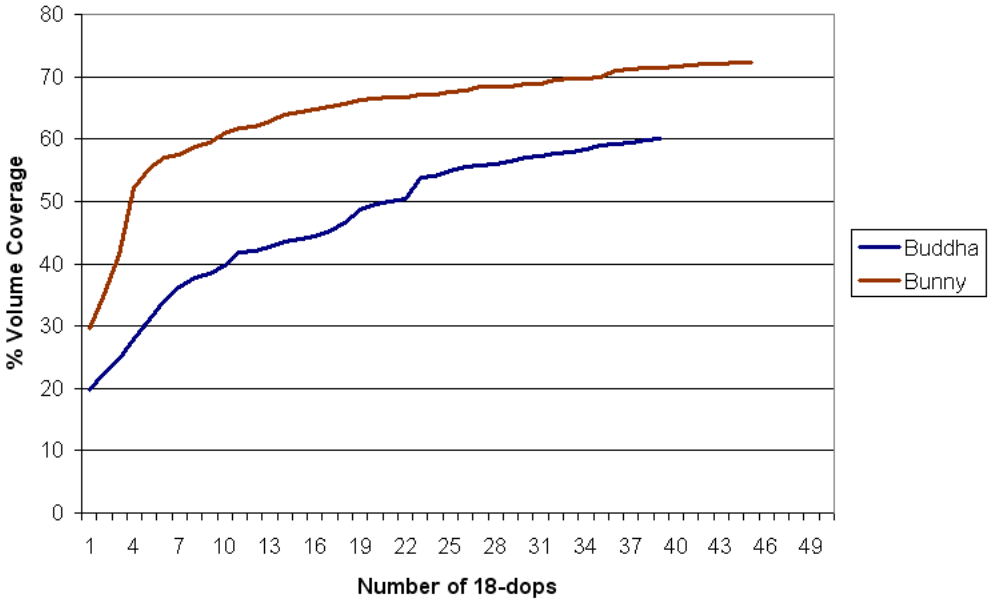


Figure 77: Increase in percentage of volume covered with increase in number of 18-dops.

to evaluate normals are necessary; results from surface reconstruction could be used here. The second problem is that of storage. For large datasets, paging starts and this slows the computation process. Our program uses QuickCD and ANN and these consume additional memory. In addition as a number of queries are performed on static trees, they could benefit from cache oblivious layouts.

Other problems include finding better seed points and expanding methods for *k*-dops. More experimentation has to be performed with merging *k*-dops. And finally these convex approximations have to be put to use in an application and the results evaluated.

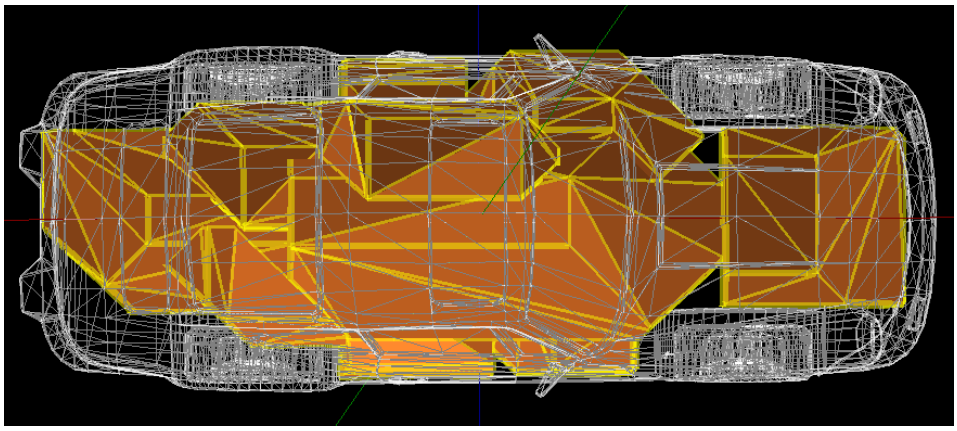


Figure 78: The top view of the Porsche model.

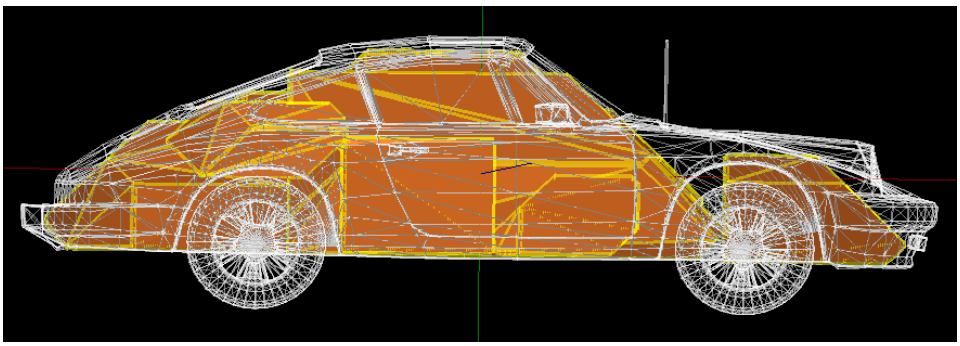


Figure 79: The side view of the Porsche model.

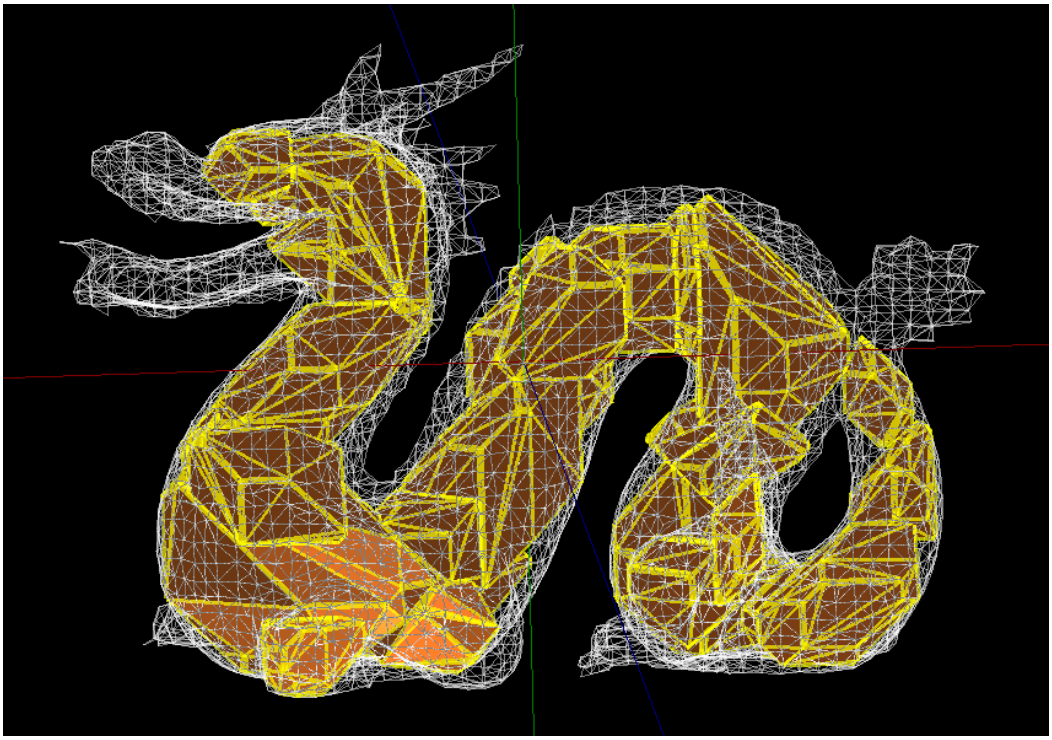


Figure 80: A packing of k-dops in the dragon model.

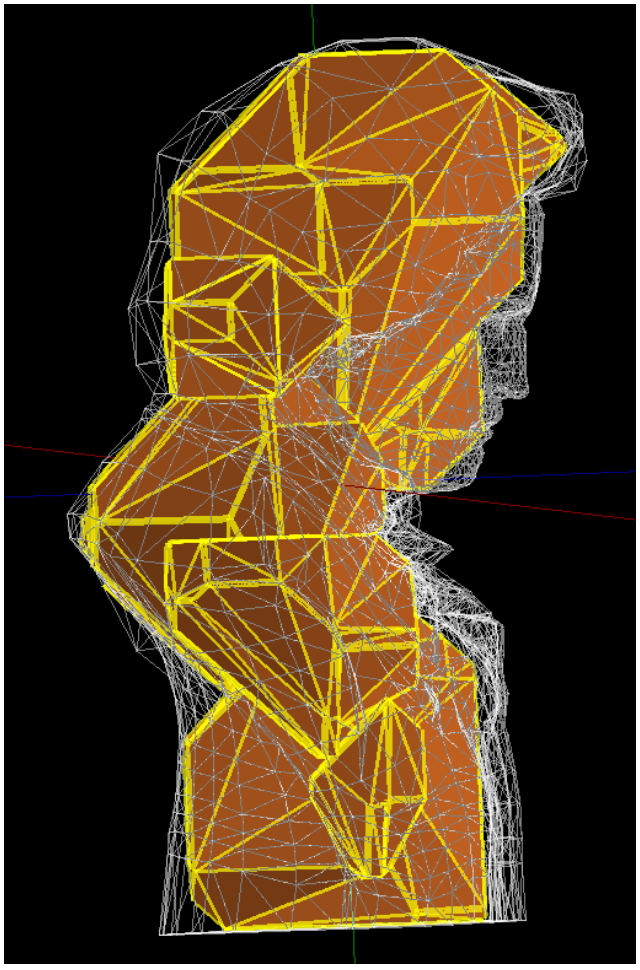


Figure 81: A packing of k-dops in the Beethoven model.

References

[1] ADLER, I. and SHAMIR, R., "A randomization scheme for speeding up algorithms for linear and convex quadratic programming problems with a high constraints-to-variables ratio," *Mathematical Programming*, vol. 61, pp. 39–52, 1993. 3

[2] AGARWAL, P. K., POREDDY, R., VARADARAJAN, K. R., and YU, H., "Practical Methods for Shape Fitting and Kinetic Data Structures using Core Sets," in *Proceedings of 20th Annual ACM Symposium on Computational Geometry*, To Appear, 2004. 3, 3.5

[3] AGARWAL, P. K., ARONOV, B., and SHARIR, M., "Computing Envelopes in Four Dimensions with Applications," in *Symposium on Computational Geometry*, pp. 348–358, 1994. 8.1

[4] AGARWAL, P. K. and MATOUŠEK, J., "On range searching with semialgebraic sets," *Discrete Computational Geometry.*, vol. 11, pp. 393–418, 1994. 6.2.1

[5] AGGARWAL, A., ALPERN, B., CHANDRA, A. K., and SNIR, M., "A model for hierarchical memory," in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 305–313, 1987. 5, 5.1

[6] AGGARWAL, A. and CHANDRA, A. K., "Virtual memory algorithms," in *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 173–185, 1988. 5

[7] AGGARWAL, A., CHANDRA, A. K., and SNIR, M., "Hierarchical memory with block transfer," in *Proceedings of 28th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 204–216, 1987. 5

[8] AGGARWAL, A. and VITTER, J. S., "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, pp. 1116–1127, 1988. 5, 6

[9] ALIZADEH, F. and GOLDFARB, D., "Second-order cone programming," Tech. Rep. RRR 51–2001, Rutgers University, Piscataway, NJ 08854, 2001. 2.1, 2.1

[10] ALON, N., DAR, S., PARNAS, M., and RON, D., "Testing of Clustering," in *Proceedings of 41st Annual IEEE Symposium on Foundations of Computer Science*, pp. 240–250, IEEE Computer Society Press, Los Alamitos, CA, 2000. 2

[11] ALPERN, B., CARTER, L., and FEIG, E., "Uniform memory hierarchies," in *Proceedings of 31st Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 600–608, 1990. 5, 5.1

[12] ALPERN, B., CARTER, L., FEIG, E., and SELKER, T., "The uniform memory hierarchy model of computation," *Algorithmica*, vol. 12, no. 2-3, 1994. 5

[13] ALTHAUS, E. and MEHLHORN, K., "Polynomial time TSP-based curve reconstruction," in *Proceedings of 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 686–695, Jan. 2000. 4.1

[14] AMATO, N. M., GOODRICH, M. T., and RAMOS, E. A., "Computing the arrangement of curve segments: divide-and-conquer algorithms via sampling," in *Proceedings of 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 705–706, 2000. 6.2.1

[15] AMATO, N. M. and RAMOS, E. A., "On computing Voronoi diagrams by divide-prune-and-conquer," in *Proceedings of the 12th Annual ACM Symposium on Computational Geometry*, pp. 166–175, 1996. 5.2.1

[16] AMENTA, N. and CHOI, S., "One-pass Delaunay filtering for homeomorphic 3D surface reconstruction." Manuscript, 1999. 4.2

[17] AMENTA, N., CHOI, S., and ROTE, G., "Incremental constructions con brio," in *Proceedings of the 19th Annual ACM Symposium on Computational Geometry*, pp. 211–219, 2003. 6.3.1

[18] AMENTA, N. and BERN, M., "Surface reconsruction by Voronoi filtering," *Discrete Computational Geometry.*, vol. 22, no. 4, pp. 481–504, 1999. 4, 4.3, 4.4

[19] AMENTA, N., BERN, M., and EPPSTEIN, D., "The Crust and the β-Skeleton: Combinatorial Curve Reconstruction," *Graphical Models and Image Processing*, vol. 60, pp. 125–135, 1998. 4.1, 8.4

[20] ANSTREICHER, K. M., "Improved complexity for maximum volume inscribed ellipsoids," *SIAM Journal on Optimization*, 2003. To Appear. 3

[21] ANSTREICHER, K. M., "Ellipsoidal approximations of convex sets based on the volumetric barrier.," *Mathematics of Operations Research*, vol. 24, pp. 193–203, February 1999. 8.1

[22] ANSTREICHER, K. M., "Improved complexity for maximum volume inscribed ellipsoids.," *SIAM Journal on Optimization*, vol. 13, pp. 309–320, June 2001. 8.1

[23] ARGE, L., BENDER, M. A., DEMAINE, E. D., HOLLAND-MINKLEY, B., and MUNRO, J. I., "Cache-oblivious priority queue and graph algorithm applications," in *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 268–276, 2002. 5.8

[24] ARYA, S. and MOUNT, D. M., "Algorithms for fast vector quantization," in *Proceedings DCC'93 (IEEE Data Compression Conference)* (STORER, J. A. and COHN, M., eds.), (Snowbird, UT, USA), pp. 381–390, 1993. 8.5.6.3

[25] ARYA, S. and MOUNT, D. M., "Approximate nearest neighbor queries in fixed dimensions," in *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1993. 8.5.6.3, 8.5.6.4

[26] BARBER, C. B., DOBKIN, D. P., and HUHDANPAA, H., "The quickhull algorithm for convex hulls," *ACM Transactions on Mathematical Software*, vol. 22, pp. 469–483, Dec. 1996. 8.4.2

[27] BAREQUET, G. and HAR-PELED, S., "Efficiently approximating the minimum-volume bounding box of a point set in three dimensions," *Journal of Algorithms*, vol. 38, pp. 91–109, 2001. 3

[28] BARTZ, D., MEISSNER, M., and HÜTTNER, T., "OpenGL-assisted occlusion culling for large polygonal models," *Computers and Graphics*, vol. 23, no. 5, pp. 667–679, 1999. 8

[29] BEN-DAVID, S., EIRON, N., and SIMON, H. U., "The computational complexity of dense region detection," in *Proceedings of the Thirteenth Annual Conference on Computational Learning Theory*, pp. 255–265, 2000. 2

[30] BEN-HUR, A., HORN, D., SIEGELMANN, H. T., and VAPNIK, V., "Support vector clustering," *revised version Jan 2002*, 2002. 2, 7.6

[31] BEN-MOSHE, B., HALL-HOLT, O., M.J.KATZ, and J.S.B.MITCHELL, "Computing the visibility graph of points within a polygon," in *Proceedings of 20th Annual ACM Symposium on Computational Geometry*, To Appear, 2004. 8.3

[32] BENDER, M. A., DEMAINE, E., and FARACH-COLTON, M., "Cache-oblivious B-trees," in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 399–409, 2000. 5.2.3, 6

[33] BENDER, M. A., DUAN, Z., IACONO, J., and WU, J., "A locality-preserving cache-oblivious dynamic dictionary," in *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 29–38, 2002. 5.2.3, 5.8, 5.9, 6

[34] BERNARDINI, F., KLOSOWSKI, J. T., and EL-SANA, J., "Directional discretized occluders for accelerated occlusion culling," *Computer Graphics Forum*, vol. 19, Aug. 2000. 8.1

[35] BETKE, U. and HENK, M., "Approximating the volume of convex bodies," *Discrete Computational Geometry*, vol. 10, pp. 15–21, 1993. 3.3

[36] BISCHOFF, S. and KOBBELT, L., "Ellipsoid decomposition of 3d-models.," in *Proceedings of 1st International Symposium on 3D Data Processing Visualization Transmission*, pp. 480–488, 2002. 8.1

[37] BLELLOCH, G. E., LEISERSON, C. E., MAGGS, B. M., PLAXTON, C. G., SMITH, S. J., and ZAGHA, M., "A comparison of sorting algorithms for the connection machine CM-2," in *ACM Symposium on Parallel Algorithms and Architectures*, pp. 3–16, 1991. 5.6.1, 5.6.1

[38] BLELLOCH, G. E., MILLER, G. L., HARDWICK, J. C., and TALMOR, D., "Design and implementation of a practical parallel Delaunay algorithm," *Algorithmica*, vol. 24, no. 3, pp. 243–269, 1999. 6.5

[39] BOUVILLE, C., "Bounding ellipsoids for ray-fractal intersection," in *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pp. 45–52, ACM Press, 1985. 3

[40] BRODAL, G. S. and FAGERBERG, R., "Cache-oblivious distribution sweeping.," in *Proceedings of 29th International Colloquium on Automata, Languages, and Programming*, vol. 2380 of *LNCS*, pp. 426–438, 2002. 6, 6.1

[41] BRODAL, G. S. and FAGERBERG, R., "Funnel heap - a cache oblivious priority queue," in *Proceedings of 13th Annual International Symposium on Algorithms and Computation*, Lecture Notes in Computer Science, Springer Verlag, 2002. 5.8, 6.6

[42] BRODAL, G. S., FAGERBERG, R., and JACOB, R., "Cache oblivious search trees via binary trees of small height," Tech. Rep. BRICS-RS-01-36, BRICS, Department of Computer Science, University of Aarhus, October 2001. 5.8, 5.9

[43] BRODAL, G. S., FAGERBERG, R., and JACOB, R., "Cache oblivious search trees via binary trees of small height," in *Proceedings of 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 39–48, Jan. 2002. 6

[44] BRUNET, P., NAVAZO, I., ROSSIGNAC, J., and SAONA-VÁZQUEZ, C., "Hoops: 3d curves as conservative occluders for cell visibility," *Computer Graphics Forum*, vol. 20, no. 3, 2001. 8.1

[45] BĂDOIU, M. and CLARKSON, K. L., "Optimal core-sets for balls.," manuscript, Bell Labs, 2002. 2, 2.4

[46] BĂDOIU, M. and CLARKSON, K. L., "Smaller core-sets for balls," in *Proceedings of 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 801–802, 2003. 2, 2.4, 3, 3.5, 7.6, 7.6

[47] BĂDOIU, M., HAR-PELED, S., and INDYK, P., "Approximate clustering via core-sets," in *Proceedings of 34th Annual ACM Symposium on Theory of Computing*, pp. 250–257, 2002. 2, 2.2.1, 2.3, 2.3, 2.3, 3, 3.5, 7.6

[48] BURGES, C. J. C., "A Tutorial on Support Vector Machines for Pattern Recognition," *Data Mining and Knowledge Discovery*, vol. 2, no. 2, pp. 121–167, 1998. 2

[49] CGAL, "Computational Geometry Algorithms Library." `http://www.cgal.org`. 8.4.1

[50] CHAN, T. M., "Faster core-set constructions and data stream algorithms in fixed dimensions," in *Proceedings of 20th Annual ACM Symposium on Computational Geometry*, To Appear, 2004. 3, 3.5

[51] CHAN, T. M., SNOEYINK, J., and YAP, C. K., "Primal dividing and dual pruning: Output-sensitive construction of 4-d polytopes and 3-d Voronoi diagrams," *Discrete Comput. Geom.*, vol. 18, pp. 433–454, 1997. 6.4.1

[52] CHAPELLE, O., VAPNIK, V., BOUSQUET, O., and MUKHERJEE, S., "Choosing multiple parameters for support vector machines," *Machine Learning*, vol. 46, no. 1/3, p. 131, 2002. 2

[53] CHATTERJEE, S. and SEN, S., "Cache-Efficient Matrix Transposition," in *International Symposium on High Performance Computer Architecture*, pp. 195–205, 2000. 5.3, 5.7, 5.9

[54] CHAUDHRY, G., CORMEN, T. H., and WISNIEWSKI, L. F., "Columnsort lives! an efficient out-of-core sorting program," in *ACM Symposium on Parallel Algorithms and Architectures*, pp. 169–178, 2001. 5.6.1

[55] CHAUDHURI, J., NANDY, S. C., and DAS, S., "Largest empty rectangle among a point set," *Journal of Algorithms*, vol. 46, no. 1, pp. 54–78, 2003. 8.1

[56] CHAZELLE, B., "Cutting hyperplanes for divide-and-conquer," *Discrete Computational Geometry*, vol. 9, no. 2, pp. 145–158, 1993. 6.2.1, 6.4.2

[57] CHAZELLE, B., DRYSDALE, III, R. L., and LEE, D. T., "Computing the largest empty rectangle," *SIAM Journal on Computing*, vol. 15, pp. 300–315, 1986. 8.1

[58] CHAZELLE, B. and MATOUŠEK, J., "On linear-time deterministic algorithms for optimization problems in fixed dimension," in *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 281–290, Society for Industrial and Applied Mathematics, 1993. 3, 3

[59] CHENG, S. W., FUNKE, S., GOLIN, M., KUMAR, P., POON, S., and RAMOS, E., "Curve Reconstruction from Noisy Samples," in *Proceedings of 19th Annual ACM Symposium on Computational Geometry*, pp. 302–311, 2003. 4, 4.1

[60] CHIANG, Y.-J., GOODRICH, M. T., GROVE, E. F., TAMASSIA, R., VENGROFF, D. E., and VITTER, J. S., "External-Memory Graph Algorithms," in *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms*, pp. 139–149, 1995. 5.8

[61] CLARKSON, K. L. and SHOR, P. W., "Applications of random sampling in computational geometry, II," *Discrete Computational Geometry*, vol. 4, pp. 387–421, 1989. 6, 3

[62] CLARKSON, K. L., "Randomized geometric algorithms," in *Computing in Euclidean Geometry* (DU, D.-Z. and HWANG, F., eds.), vol. 4 of *Lecture Notes Series on Computing*, pp. 149–194, Singapore: World Scientific, 2nd ed., 1995. 6

[63] COHEN-OR, D., LEV-YEHUDI, S., KAROL, A., and TAL, A., "Inner-cover of non-convex shapes," 2000. 8.1

[64] COMANICIU, D. and MEER, P., "Robust analysis of feature spaces: Color image segmentation," in *IEEE Conf. Computer Vision and Pattern Recognition (CVPR'97)*, pp. 750–755, 1997. 3

[65] COORG, S. R. and TELLER, S. J., "Real-time occlusion culling for models with large occluders," in *Symposium on Interactive 3D Graphics*, pp. 83–90, 189, 1997. 8

[66] COPPERSMITH, D. and WINOGRAD, S., "Matrix multiplication via arithmetic progression," *Journal of Symbolic Computation*, vol. 9, pp. 251–280, 1990. 5.4

[67] CORMEN, T. H., LEISERSON, C. E., and RIVEST, R. L., *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990. 5, 5.2.1, 5.2.1, 5.2.1, 5.2.3, 5.4

[68] CRAUSER, A., FERRAGINA, P., MEHLHORN, K., MEYER, U., and RAMOS, E. A., "Randomized external-memory algorithms for line segment intersection and other geometric problems," *International Journal of Computational Geometry and Applications*, vol. 11, pp. 305–337, 2001. 6, 6, 6.3, 6.3.1, 6.3.2, 6.3.2

[69] CUISENAIRE, O. and MACQ, B., "Fast and exact signed euclidean distance transformation with linear complexity," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 6, pp. 3293–3296, Mar. 1999. 7.4

[70] DANIELS, K., MILKENKOVIC, V., and ROTH, D., "Finding the Largest Area Axis-Parallel Rectangle in a Polygon.," *CGTA: Computational Geometry: Theory and Applications*, vol. 7, 1997. 8.1

[71] DATTA, A., "Efficient algorithms for the largest rectangle problem," *Information Sciences*, vol. 64, pp. 121–141, 1992. 8.1

[72] DEHNE, F., DENG, X., DYMOND, P., FABRI, A., and KHOKHAR, A. A., "A randomized parallel 3D convex hull algorithm for coarse grained multicomputers," in *Proceedings the of 7th ACM Symposium on Parallel Algorithms and Architectures.*, pp. 27–33, 1995. 6.4.1

[73] DEY, T. K. and KUMAR, P., "A simple provable algorithm for curve reconstruction.," in *Proceedings of 10th Annual ACM-SIAM Symposium on Discrete Algorithms.*, pp. 893–894, 1999. 4.1, 4.1.1, 3

[74] DEY, T. K., MEHLHORN, K., and RAMOS., E., "Curve reconstruction: connecting dots with good reason.," *Computational Geometry: Theory and Applications*, vol. 15, pp. 229–244, 2000. 4.1

[75] DEY, T. K. and WENGER., R., "Reconstructing curves with sharp corners.," *Computational Geometry: Theory and Applications*, vol. 19, pp. 89–99, 2001. 4.1

[76] DEY, T. K. and WENGER., R., "Fast reconstruction of curves with sharp corners.," *Computational Geometry: Theory and Applications*, vol. 12, pp. 353–400, 2002. 4.1

[77] DOBKIN, D. P. and LIPTON, R. J., "Multidimensional searching problems," *SIAM Journal on Computing*, vol. 5, pp. 181–186, 1976. 6.2.1

[78] DOBKIN, D., EDELSBRUNNER, H., and OVERMARS, M., "Searching for empty convex polygons," in *Proceedings of the 4th Annual ACM Symposium on Computational Geometry*, pp. 224–228, 1988. 8.1

[79] DUDA, R. O., HART, P. E., and STORK, D. G., *Pattern classification*. Wiley, 2nd ed., 2001. 7.6

[80] DUIN, R. P. W., "Prtools, a matlab toolbox for pattern recognition. `http://www.ph.tn.tudelft.nl/~bob/prtools.html`," 2000. (document), 2

[81] DUNAGAN, J. and VEMPALA, S., "Optimal outlier removal in high-dimensional spaces," in *Proceedings of 33rd Annual ACM Symposium on Theory of Computing*, pp. 627 – 636, 2001. 7.6, 7.6

[82] DUTA, N., JAIN, A. K., and MARDIA, K. V., "Matching of palmprint," *Pattern Recognition Letters*, vol. 23, no. 4, pp. 477–485, 2002. 7.2

[83] EBERLY, D., "Magic Software." `http://www.magic-software.com`. 4.5.1

[84] EBERLY, D., *3D Game Engine Design*. Morgan Kaufmann, 2001. 3

[85] EDELSBRUNNER, H. and MÜCKE, E. P., "Three-dimensional alpha shapes," *ACM Transactions on Graphics.*, vol. 13, pp. 43–72, Jan. 1994. 4.5

[86] EĞECIOĞLU, O. and KALANTARI, B., "Approximating the diameter of a set of points in the euclidean space," *Information Processing Letters, 32:205-211*, 1989. 2.1

[87] EIRON, N., RODEH, M., and STEINWARTS, I., "Matrix multiplication: A case study of algorithm engineering," in *2nd Workshop on Algorithm Engineering* (MEHLHORN, K., ed.), no. MPI-I-98-1-019, ISSN: 0946-011X in Research Reports MPII, pp. 98–109, 1998. 5.4

[88] ELZINGA, D. J. and HEARN, D. W., "The minimum covering sphere problem," *Management Science*, vol. 19, pp. 96–104, Sept. 1972. 2

[89] EPPSTEIN, D., "Junkyard." `http://www.ics.uci.edu/~eppstein/junkyard/circumcenter.html`. 4.5.1

[90] ERICKSON, J. and AGARWAL, P. K., "Geometric Range Searching and Its Relatives," in *Advances in Discrete and Computational Geometry, Contemporary Mathematics 223* (CHAZELLE, B., GOODMAN, J. E., and POLLACK, R., eds.), ch. 1, pp. 1–56, American Mathematical Society, 1999. 8.3, 8.4

[91] FAULKNER, K. W., "Apparatus and method for biometric identification," Aug. 1994. US Patent No. 5,335,288. 7.2

[92] FISCHER, K., "Smallest enclosing ball of balls," diploma thesis, Institute of Theoretical Computer Science, ETH Zurich, 2001. 2

[93] FOLLERT, F., SCHÖMER, E., SELLEN, J., SMID, M., and THIEL, C., "Computing a largest empty anchored cylinder, and related problems," *International Journal of Computational Geometry and Applications*, vol. 7, pp. 563–580, 1997. 8.1

[94] FRIGO, M., "Portable High-Performance Programs," Tech. Rep. MIT/LCS/TR-785, MIT, 1999. 5.7

[95] FRIGO, M., "A Fast Fourier Transform Compiler," in *PLDI'99 — Conference on Programming Language Design and Implementation*, (Atlanta, GA), 1999. 5.7, 5.9

[96] FRIGO, M., LEISERSON, C. E., PROKOP, H., and RAMACHANDRAN, S., "Cache oblivious algorithms," in *Proceedings of 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 285–298, Oct. 1999. 6, 6.1, 6.1, 6.1, 6.6

[97] FRIGO, M., E.LIESERSON, C., PROKOP, H., and RAMACHANDRAN, S., "Cache oblivious algorithms," in *Proceedings of 40th Annual IEEE Symposium on Foundations of Computer Science*, Oct. 1999. 2.4, 5, 5.1, 5.1.1, 5.1, 5.6.1, 5.6.1, 5.8

[98] FUNKE, S. and RAMOS., E. A., "Reconstructing a collection of curves with corners and endpoints.," in *Proceedings of 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 344–353, Jan. 2001. 4.1

[99] GÄRTNER, B. and SCHÖNHERR, S., "Exact primitives for smallest enclosing ellipses," in *Proceedings of 13th Annual ACM Symposium on Computational Geometry*, pp. 430–432, 1997. 3

[100] GÄRTNER, B., "Fast and robust smallest enclosing balls," in *Proc. 7th Annual European Symposium on Algorithms (ESA)*, Springer-Verlag, 1999. 2, 2.4

[101] GÄRTNER, B. and SCHÖNHERR, S., "An efficient, exact, and generic quadratic programming solver for geometric optimization," in *Proceedings of 16th Annual ACM Symposium on Computational Geometry*, pp. 110–118, 2000. 2, 2.4

[102] GIESEN, J., "Curve reconstruction, the TSP, and Menger's theorem on length," in *Proceedings of 15th Annual ACM Symposium on Computational Geometry*, pp. 207–216, 1999. 4.1

[103] GLINEUR, F., "Pattern separation via ellipsoids and conic programming," master's thesis, Faculté Polytechnique de Mons, Belgium, 1998. 3

[104] GOEL, A., INDYK, P., and VARADARAJAN, K. R., "Reductions among high dimensional proximity problems," in *Symposium on Discrete Algorithms*, pp. 769–778, 2001. 2, 2.3

[105] GOLD, C., "Crust and anti-crust: A one-step boundary and skeleton extraction algorithm," in *Proceedings of 15th Annual ACM Symposium on Computational Geometry*, pp. 189–196, 1999. 4.1

[106] GOODRICH, M. T., TSAY, J.-J., VENGROFF, D. E., and VITTER, J. S., "External-memory computational geometry," in *Proceedings of 34th Annual IEEE Symposium on Foundations of Computer Science*, pp. 714–723, 1993. 6, 6, 6.4.1

[107] GOVINDARAJU, N. K., SUD, A., YOON, S.-E., and MANOCHA, D., "Parallel occlusion culling for interactive walkthroughs using multiple GPUs," tech. rep., Department of Computer Science, UNC-Chapel Hill, 2002. 8

[108] GRAHAM, R. L., KNUTH, D. E., and PATASHNIK, O., *Concrete Mathematics*. Reading, MA: Addison-Wesley, 1989. 5.2.1

[109] GRÖTSCHEL, M., LOVÁSZ, L., and SCHRIJVER, A., *Geometric Algorithms and Combinatorial Optimization*. Springer, New York, 1988. 3, 3

[110] HAR-PELED, S., "Personal communication." Manuscript., 2002. 2.3, 2.4, 1

[111] HENNESSY, J. L. and PATTERSON, D. A., *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990. 5, 5.7

[112] HONG, J. W. and KUNG, H. T., "I/O complexity: The red-blue pebble game," in *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 326–333, 1981. 5

[113] HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., and STUETZLE, W., "Surface reconstruction from unorganized points," in *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pp. 71–78, ACM Press, 1992. 4.2, 4.3

[114] HUBBARD, P. M., "Approximating polyhedra with spheres for time-critical collision detection," *ACM Transactions on Graphics*, vol. 15, no. 3, pp. 179–210, 1996. 2

[115] ITAI, A., KONHEIM, A., and RODEH, M., "A sparse table implementation of priority queues," in *Automata, Languages and Programming, 8th Colloquium*, vol. 115, pp. 417–431, 13–17 July 1981. 5.2.3

[116] JAIN, A. K. and DUTA, N., "Deformable matching of hand shapes for verification," in *Proceedings of International Conference on Image Processing*, Oct. 1999. 7.2

[117] JAIN, A. K., PRABHAKAR, S., and ROSS, A., "Biometrics-based web access," Tech. Rep. MSU-CPS-98-33, Department of Computer Science, Michigan State University, East Lansing, Michigan, November 1998. 7.2

[118] JAIN, A. K., ROSS, A., and PANKANTI, S., "A prototype hand geometry-based verification system," in *Proceedings of 2nd Int'l Conference on Audio- and Video-based Biometric Person Authentication*, pp. 166–171, Mar. 1999. 7.2

[119] JOHN, F., "Extremum problems with inequalities as subsidiary conditions," in *Studies and Essays, presented to R. Courant on his 60th birthday January 8, 1948*, pp. 187–204, New York: Interscience, 1948. Reprinted in: *Fritz John, Collected Papers Volume 2* (J. Moser, ed), Birkhäuser, Boston, 1985, pp. 543–560. 3

[120] JOHNSON, W. and LINDENSTRAUSS, J., "Extensions of Lipschitz maps into a Hilbert space," *Contemporary Mathematics*, vol. 26, pp. 189–206, 1984. 1

[121] JOLION, J. M., MEER, P., and BATAOUCHE, S., "Robust clustering with applications in computer vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, pp. 791–802, Aug 1991. 3

[122] JR., C. R. M., RAGHAVAN, V., and QI, R., "A linear time algorithm for computing the euclidean distance transform in arbitrary dimensions," in *Proceedings of the 17th International Conference on Information Processing in Medical Imaging*, vol. 2082 of *Lecture Notes in Computer Science*, pp. 358–364, June 2001. 7.4

[123] K. FISCHER, B. G. and KUTZ, M., "Fast smallest-enclosing-ball computation in high dimensions," in *Proceedings of the 11th Annual European Symposium on Algorithms (ESA)*, vol. 2832 of *Lecture Notes Computer Science*, pp. 630–641, Springer-Verlag, 2003. 2, 2.4

[124] KHACHIYAN, L. G., "Rounding of polytopes in the real number model of computation," *Mathematics of Operations Research*, vol. 21, pp. 307–320, 1996. 3, 3.4.1, 3.4.1, 3.4.2, 3.4.2

[125] KHACHIYAN, L. G. and TODD, M. J., "On the complexity of approximating the maximal inscribed ellipsoid for a polytope," *Mathematical Programming*, vol. 61, pp. 137–159, 1993. 3, 3.2, 3.4.2, 3.5, 8.1

[126] KLOSOWSKI, J., HELD, M., and MITCHELL, J., "Quickcd." 8.2, 8.5.1, 8.5.6.2

[127] KLOSOWSKI, J., *Efficient Collision Detection for Interactive 3D Graphics and Virtual Environments*. PhD thesis, State University of New York at Stony Brook, Stony Brook, NY 11790, may 1998. 8.5.6.2

[128] KLOSOWSKI, J. T., HELD, M., MITCHELL, J. S. B., SOWIZRAL, H., and ZIKAN, K., "Efficient collision detection using bounding volume hierarchies of k-dops," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21–36, 1998. 8.5.6.2, 8.5.6.4

[129] KNORR, E. M., NG, R. T., and ZAMAR, R. H., "Robust space transformations for distance-based operations," in *Knowledge Discovery and Data Mining*, pp. 126–135, 2001. 3

[130] KNUTH, D. E., *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*. Reading, Massachusetts: Addison-Wesley, second ed., 10 Jan. 1974. 5.2.2

[131] KOLTUN, V., CHRYSANTHOU, Y., and COHEN-OR, D., "Virtual occluders: An efficient intermediate pvs representation," in *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pp. 59–70, Jun 2000. 8, 8.1

[132] KUMAR, P., MITCHELL, J. S. B., and YILDIRIM, E. A., "Computing Core-Sets and Approximate Smallest Enclosing HyperSpheres in High Dimensions," in *Proceedings of ALENEX 2003*, pp. 45–55, 2003. 2, 2.3, 2.4, 3, 3.5, 7.6

[133] LADNER, R. E., FORTNA, R., and NGUYEN, B. H., "A comparison of cache aware and cache oblivious static search trees using program instrumentation." To appear in LNCS volume devoted to Experimental Algorithmics, Apr. 2002. 5.5.1, 5.7

[134] LAMARCA, A. and LADNER, R. E., "The influence of caches on the performance of sorting," in *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 370–379, 5–7 Jan. 1997. 5.6

[135] LEDA, "Library of Efficient Data types and Algorithms." `www.algorithmic-solutions.com`. 6.5

[136] LI, W., ZHANG, D., and XU, Z., "Palmprint identification by fourier transform," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 16, no. 4, pp. 417–432, 2002. 7.2

[137] LOBO, M. S., VANDENBERGHE, L., BOYD, S., and LEBRET, H., "Applications of second-order cone programming," *Linear Algebra and Its Applications*, vol. 248, pp. 193–228, 1998. 2.1

[138] LUEBKE, D. and ERIKSON, C., "View-dependent simplification of arbitrary polygonal environments," *Computer Graphics*, vol. 31, no. Annual Conference Series, pp. 199–208, 1997. 8

[139] MATOUŠEK, J., *Lectures on Discrete Geometry*. Prague: Springer Verlag, 2001. 3.3

[140] MATOUŠEK, J. and SCHWARZKOPF, O., "A deterministic algorithm for the three-dimensional diameter problem," in *Proceedings of 25th Annual ACM Symposium on Theory of Computing*, pp. 478–484, 1993. 6.2.1

[141] MATOUŠEK, J., SHARIR, M., and WELZL, E., "A subexponential bound for linear programming," in *Proceedings of 8th Annual ACM Symposium on Computational Geometry*, pp. 1–8, 1992. 2

[142] MATOUŠEK, J., SHARIR, M., and WELZL, E., "A subexponential bound for linear programming," in *Proceedings of the eighth annual symposium on Computational geometry*, pp. 1–8, 1992. 3

[143] MENCL, R. and MULLER, H., "Graph-based surface reconstruction using structures in scattered point sets.," Research Report 661, Universität Dortmund, 1997. 4.3

[144] MOUNT, D., "ANN: Library for Approximate Nearest Neighbor Searching." `http://www.cs.umd.edu/~mount/ANN/`. 3, 8.5.2, 8.5.6.3

[145] MOUNT, D. M., NETANYAHU, N. S., PIATKO, C. D., SILVERMAN, R., and WU, A. Y., "Quantile approximation for robust statistical estimation and k-enclosing problems," *International Journal of Computational Geometry and Applications*, vol. 10, no. 6, pp. 593–608, 2000. 3

[146] MULMULEY, K., *Computational Geometry: An Introduction Through Randomized Algorithms*. Englewood Cliffs, NJ: Prentice Hall, 1993. 6, 6.3

[147] NEMIROVSKII, A. S., "On self–concordant convex–concave functions," Research Report 3/97, Optimization Laboratory Faculty of Industrial Engineering and Managament at Technion, Technion City, Haifa 32000, Israel, June 1997. 3

[148] NESTEROV, Y. E. and NEMIROVSKII, A. S., *Interior Point Polynomial Methods in Convex Programming*. Philadelphia: SIAM Publications, 1994. 2.1, 3, 3.2, 3.4.2, 7.6

[149] NESTEROV, Y. E. and TODD, M. J., "Primal-dual interior-point methods for self-scaled cones," *SIAM Journal on Optimization*, vol. 8, pp. 324–362, 1998. 2.1

[150] NYBERG, C., BARCLAY, T., CVETANOVIC, Z., GRAY, J., and LOMET, D. B., "Alphasort: A risc machine sort," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994* (SNODGRASS, R. T. and WINSLETT, M., eds.), pp. 233–242, ACM Press, 1994. 5.6

[151] ÖDEN, C., ERÇIL, A., YILDIZ, V. T., KIRMIZITAS, H., and BÜKE, B., "Hand recognition using implicit polynomials and geometric features," in *Proceedings of the Third International Conference on Audio- and Video-Based Biometric Person Authentication* (BIGÜN, J. and SMERALDI, F., eds.), vol. 2091 of *Lecture Notes in Computer Science*, pp. 336–341, Springer, June 2001. 7.2

[152] OWENS, J. D., DALLY, W. J., KAPASI, U. J., RIXNER, S., MATTSON, P., and MOWERY, B., "Polygon rendering on a stream architecture," in *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pp. 23–32, Aug. 2000. 8

[153] PELLEGRINI, M., "Randomized combinatorial algorithms for linear programming when the dimension is moderately high," in *Proceedings of 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2001. 2.4

[154] POON, S., *Curve and Surface Reconstruction from Noisy Samples.* PhD dissertation, HKUST, Department of Computer Science, June-Aug. 2004. 4, 4.1, 4.1.1

[155] PROCOPIUC, O., AGARWAL, P. K., ARGE, L., and J.S.VITTER, "Bkd-tree: A dynamic scalable kd-tree," *Unpublished Manuscript*, 2002. 6.5

[156] PROKOP, H., "Cache-oblivious algorithms," Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999. 6.1

[157] RAHMAN, N., COLE, R., and RAMAN, R., "Optimized predecessor data structures for internal memory," in *5th Workshop on Algorithms Engineering (WAE)*, 2001. 5.7

[158] RAJASEKARAN, S. and RAMASWAMI, S., "Optimal parallel randomized algorithms for the Voronoi diagram of line segments in the plane," *Algorithmica*, vol. 33, pp. 436–460, 2002. 6.4.1

[159] "Recognition systems inc. `http://www.recogsys.com`." 7.2

[160] "Recognition Systems' HandKey II product." `http://www.recogsys.com/products/hk/ac_handkey_2.htm`. 7.2

[161] REIF, J. H. and SEN, S., "Optimal parallel randomized algorithms for three-dimensional convex hulls and related problems," *SIAM J. Comput.*, vol. 21, pp. 466–485, 1992. 6, 6.4.1

[162] RENEGAR, J., *A Mathematical View of Interior-Point Methods in Convex Optimization*. MPS/SIAM Series on Optimization 3, Philadelphia: SIAM Publications, 2001. 2.1

[163] ROSS, A., "A prototype hand geometry-based verification system," M.S. project report, Computer Science & Engineering, Michigan State University, East Lansing, MI 48824, USA, 1999. 7.1, 7.2

[164] RUSINKIEWICZ, S. and LEVOY, M., "QSplat: A multiresolution point rendering system for large meshes," in *Siggraph 2000, Computer Graphics Proceedings* (AKELEY, K., ed.), pp. 343–352, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000. 8

[165] SANCHEZ-REILLO, R., SANCHEZ-AVILA, C., and GONZALEZ-MARCOS, A., "Biometric identification through hand geometry measurements," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, pp. 1168–1171, Oct. 2000. 7.2

[166] SAVAGE, J. E., "Extending the Hong-Kung model to memory hierachies," in *Proceedings of the 1st Annual International Conference on Computing and Combinatorics*, vol. 959 of *LNCS*, pp. 270–281, August 1995. 5

[167] SEN, S. and CHATTERJEE, S., "Towards a theory of cache-efficient algorithms," in *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 829–838, Jan. 2000. 5

[168] SHEWCHUK, J. R., "Triangle: Engineering a 2d quality mesh generator and Delaunay triangulator," in *First Workshop on Applied Computational Geometry*, Association for Computing Machinery, May 1996. 6.5

[169] SIDLAUSKAS, D., "3D hand profile identification apparatus," Apr. 1988. US Patent No. 4,736,203. 7.2

[170] SILVERMAN, B. W. and TITTERINGTON, D. M., "Minimum covering ellipses," *SIAM Journal on Scientific and Statistical Computing*, vol. 1, pp. 401–409, 1980. 3, 3

[171] SILVEY, S. and TITTERINGTON, D., "A geometric approach to optimal design theory," *Biometrika*, vol. 62, pp. 21–32, 1973. 3

[172] SLEATOR, D. D. and TARJAN, R. E., "Amortized efficiency of list update and paging rules," *Communications of the ACM*, vol. 28, pp. 202–208, 1985. 5.1

[173] SMOLA, A., BARTLETT, P., SCHÖLKOPF, B., and SCHUURMANS, C., *Advances in Large Margin Classifiers*. Cambridge, MA: MIT Press, 1999. 7

[174] STRASSEN, V., "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, pp. 354–356, 1969. 5.4

[175] STURM, J. F., "Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones," *Optimization Methods and Software*, vol. 11/12, pp. 625–653, 1999. 2.4

[176] SUN, P. and FREUND, R. M., "Computation of minimum volume covering ellipsoids," tech. rep., MIT Operations Research Center, 2002. 3, 3.2, 3.4.1, 3.5

[177] TARASOV, S. P., KHACHIYAN, L. G., and ERLIKH, I. I., "The method of inscribed ellipsoids," *Soviet Mathematics Doklady*, vol. 37, pp. 226–230, 1988.

[178] TAX, D. M. J., *One-class classification ; Concept-learning in the absence of counter-examples*. Ph.D. thesis, Delft University of Technology, 65, Delft, June 2001. 7.6

[179] TEMAM, O., FRICKER, C., and JALBY, W., "Cache interference phenomena," in *Measurement and Modeling of Computer Systems*, pp. 261–271, 1994. 5.7

[180] TITTERINGTON, D. M., "Optimal design: some geometrical aspects of *D*-optimality," *Biometrika*, vol. 62, no. 2, pp. 313–320, 1975. 3, 3

[181] TITTERINGTON, D. M., "Estimation of correlation coefficients by ellipsoidal trimming," *Applied Statistics*, vol. 27, no. 3, pp. 227–234, 1978. 3

[182] TOH, K. C., "Primal-dual path-following algorithms for determinant maximization problems with linear matrix inequalities," *Computational Optimization and Applications*, vol. 14, pp. 309–330, 1999. 3

[183] TOH, K. C., TODD, M. J., and TÜTÜNCÜ, R. H., "SDPT3 — a Matlab software package for semidefinite programming," *Optimization Methods and Software*, vol. 11, pp. 545–581, 1999. 2.1, 2.4

[184] TOLEDO, S., "Locality of reference in *LU* decomposition with partial pivoting," *SIAM Journal on Matrix Analysis and Applications*, vol. 18, pp. 1065–1081, Oct. 1997. 5.8

[185] TÜTÜNCÜ, R. H., TOH, K. C., and TODD, M. J., "Solving semidefinite-quadratic-linear programs using SDPT3," tech. rep., Cornell University, 2001. To appear in *Mathematical Programming*. 2.1

[186] VANDENBERGHE, L., BOYD, S., and WU, S., "Determinant Maximization with Linear Matrix Inequality Constraints," *SIAM Journal on Matrix Analysis and Applications*, vol. 19, pp. 499–533, Apr. 1998. 3

[187] WELZL, E., "Smallest enclosing disks (balls and ellipsoids)," in *New Results and New Trends in Computer Science*, Lecture Notes Computer Science, pp. 359–370, Springer-Verlag, 1991. 2

[188] WELZL, E., "Smallest enclosing disks (balls and ellipsoids)," in *Proceedings of New Results and New Trends in Computer Science* (MAURER, H., ed.), vol. 555 of *LNCS*, (Berlin, Germany), pp. 359–370, Springer, June 1991. 3, 3

[189] WISE, D. S., "Ahnentafel indexing into morton-ordered arrays, or matrix locality for free," in *Euro-Par 2000, Parallel Processing, 6th International Euro-Par Conference, Munich, Germany, August 29 - September 1, 2000, Proceedings*, vol. 1900 of *LNCS*, pp. 774–784, August 2000. 5.1, 5.3

[190] WOLBERG, W. H., STREET, W. N., HEISEY, D. M., and MANGASARIAN, O. L., "Computer-derived nuclear features distinguish malignant from benign breast cytology," *Hum Pathol*, vol. 26, pp. 792–796, Jul 1995. 7.8

[191] WOLBERG, W. H., STREET, W. N., HEISEY, D. M., and MANGASARIAN, O. L., "Computerized breast cancer diagnosis and prognosis from fine-needle aspirates," *Arch Surg*, vol. 130, pp. 511–516, May 1995. Clinical Trial. 7.8

[192] WOLBERG, W. H., STREET, W. N., and MANGASARIAN, O. L., "Machine learning techniques to diagnose breast cancer from image-processed nuclear features of fine needle aspirates," *Cancer Lett*, vol. 77, pp. 163–171, Mar 1994. 7.8, 7.8.1

[193] XU, S., FREUND, R., and SUN, J., "Solution methodologies for the smallest enclosing circle problem," tech. rep., Singapore-MIT Alliance, National University of Singapore, Singapore, 2001. 2

[194] Y. BULATOV, S. JAMBAWALIKAR, P. K. and SETHIA., S., "Hand recognition using geometric classifiers.." To Appear in Proceedings of International Conference on Biometric Authentication, 2004. 2

[195] YAO, A. C. and YAO, F. F., "A general approach to *D*-dimensional geometric queries," in *Proceedings of 17th Annual ACM Symposium on Theory of Computing*, pp. 163–168, 1985. 6.2.1

[196] YI, Q., ADVI, V., and KENNEDY, K., "Transforming loops to recursion for multi-level memory hierarchies," in *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Vancouver, Canada), pp. 169–181, ACM, June 2000. 5.1

[197] YILDIRIM, E. A. and WRIGHT, S. J., "Warm-Start Strategies in Interior-Point Methods for Linear Programming," *SIAM Journal on Optimization*, vol. 12, no. 3, pp. 782–810, 2002. 1

[198] ZHANG, Y., "An interior-point algorithm for the maximum-volume ellipsoid problem," Tech. Rep. TR98-15, Department of Computational and Applied Mathematics, Rice University, 1998. 3, 8.1

[199] ZHANG, Y. and GAO, L., "On numerical solution of the maximum volume ellipsoid problem.," 2001. 3, 3.5

[200] ZHOU, G., SUN, J., and TOH, K.-C., "Efficient algorithms for the smallest enclosing ball problem in high dimensional space," tech. rep., Singapore-MIT alliance, 2002. To appear in Procedings of Fields Institute of Mathematics. 2, 2.1

Vita

"Piyush is a low cost, 150 ($\pm$10 depending on external factors) pound, non-convex, non-linear, all-purpose computer which likes to kill its time in various ways, including thinking about various areas and related areas of computer science."

π