Εθνικο Καποδιστριακο Πανεπιστημιο Αθηνων
Τμημα Μαθηματικων
Μεταπτυχιακο Προγραμμα Λογικης Και Αλγοριθμων

Implementing Approximate Voronoi Diagrams for Approximate Nearest Neighbor Searching

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αλέξανδρου Κωνσταντινάκη - Κάρμη
ΑΜ 201010

Επιβλέπων: Ιωάννης Εμίρης
Καθηγητής Ε.Κ.Π.Α.

Αθήνα, Αύγουστος 2012

....
Copyright © Αλέξανδρου Κωνσταντινάκη - Κάρμη, 2012.
Με επιφύλαξη παντός δικαιώματος. All rights reserved .

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Καποδιστριακού Πανεπιστημίου Αθηνών.

Περίληψη

Η αναζήτηση πλησιέστερου γείτονα είναι ένα θεμελιώδες πρόβλημα στην Επιστήμη των Υπολογιστών. Ένα άπο τα κύρια θεωρητικά ζητήματα ειναι η εξισορρόπηση του χρόνου που απαιτείται για να απαντήσει η δομή σε ερωτήματα και η χωρική πολυπλοκότητά της: η πρόσφατη μελέτη στο Προσεγγιστικό Διάγραμμα Voronoi(AVD) οδηγεί σε βέλτιστο χρόνο ερωτήματος δίνοντας τη δυνατότητα στο χρήστη να επιλέξει μεταξύ λιγότερου χώρου ή γρηγορότερου χρόνου ερωτήματος. Η βασική ιδέα είναι να χωριστεί ο χώρος γύρω από τα σημεία εισόδου και να γίνει χρήση κουτιών διαφόρων μεγεθών για να καλυφθεί. Σε κάθε κουτί αποδίδουμε ένα υποσύνολο των σημείων εισόδου, τα οποία ειναι υποψήφιοι ε - προσεγγιστικοί πλησιέστεροι γείτονες για κάθε σημείο σε αυτό το κουτί. Η υλοποίηση του AVD είναι ένα δύσκολο έργο που δεν έχει υλοποιηθεί ώστε να δέχεται τη διάσταση d των σημείων ως είσοδο. Προσφέρουμε μια αποτελεσματική, παράλληλη εφαρμογή που κατασκευάζει το AVD, προσφέροντας ιδέες τροποποίησης του αλγορίθμου έτσι ώστε να καθίσταται η κατασκευή εφικτή. Στα πειραματικά μας αποτελέσματα δείχνουμε ότι τα η δομή μασ απαντά πιο γρήγορα και με μεγαλύτερη ακρίβεια σε σχέση με την καλυτερη υλοποίηση KD tree σε μικρές διαστάσεις.

Λέξεις Κλειδιά:
Προσεγγιστικός Πλησιέστερος Γείτονας, Προσεγγιστικό Διάγραμμα Voronoi, βέλτιστος χρόνος ερωτήματος, KD/BBD tree, Quadtree

Abstract

Nearest neighbor searching is a fundamental problem in computer science. One of the main theoretical issues is to balance query time and space complexity: the recent Approximate Voronoi Diagram (AVD) [3] leads to optimal query time while making the tradeoff with space usage explicit. The key idea is to cluster the space around the input points and use boxes of varying size to cover it. Each cluster is assigned a subset of the points, which are candidate ε-approximate nearest neighbors to every point in that cluster. However, the implementation of AVD has been a daunting task and was never completed. We offer the first, efficient, parallel implementation of the AVD which accepts dimension as input and introduce certain ideas and modifications that make the construction feasible. In our experimental results, we show that our data structure is much faster and more accurate than a standard KD-tree in low dimensions.

Keywords:
Approximate Nearest Neighbor, Approximate Voronoi Diagram, optimal query time, KD/BBD tree, Quadtree

Contents

1 Introduction **5**

1.1 Previous Work . . . 5

1.2 Our Contribution . . . 6

2 KD and BBD tree **7**

2.1 KD tree . . . 7

2.1.1 Construction . . . 7

2.1.2 Adding elements . . . 9

2.1.3 Removing elements . . . 9

2.1.4 Nearest neighbour search . . . 9

2.2 BBD tree . . . 10

2.2.1 Construction . . . 11

2.2.2 Nearest neighbour search . . . 12

2.2.3 Comparison with KD tree . . . 13

3 The Main Algorithm **14**

3.1 Step 1: Preprocessing the input points . . . 14

3.2 Step 2: WSPD . . . 14

3.3 Step 3: Finding the overlapping quadtree boxes . . . 15

3.3.1 Step 3′ . . . 16

3.4 Step 4: Deciding the representatives . . . 16

4 Our Variant **17**

4.1 Quadtree boxes that cover a sphere . . . 17

4.2 Dealing with space requirements . . . 18

4.2.1 Method B: Keep Step 3 and 4 separate . . . 20

5 Our Implementation **21**
5.1 Method A 21
5.2 Method B 22
6 Code Snippets **24**
7 Experimental Results **29**
7.0.1 Method A 30
7.0.2 Method A′ 30
7.0.3 Method B 31
7.0.4 Method Summary 32
8 Conclusion **33**
9 Appendix **35**

Chapter 1

Introduction

Nearest neighbor searching (NNS) is a fundamental problem in computer science with several important applications, including machine learning, geometric inference and high-dimensional optimization. The problem can be formulated in the following way: Given a point set S in d dimensions and a query point q, we want to efficiently calculate its nearest point $s \in S$, for which $dist(s, q)$ is minimum, using some distance function (Euclidean, Manhattan etc.). In the approximate version of the problem, an approximation factor ε is also part of the input and the answer s' is allowed to be at most ε times further away than the real nearest neighbor s, that is $dist(s', q) \leq (1+\varepsilon) \cdot dist(s, q)$.

1.1 Previous Work

Research is concentrated on preprocessing the points into a data structure that allows for fast query time. We focus on solutions that work well for small dimensions (≤ 10), therefore we omit methods such as Locality Sensitive Hashing [12] or FLANN [14] [13] which are aimed at high dimensions. The principal complexity issues are to determine the query time and space. The traditional Voronoi Diagram has space complexity $O(n^{\lceil d/2 \rceil})$ and query time $O(d \log n)$, where $n = |S|$. J. L. Bentley described the KD tree in [5], a data structure which was subsequently improved upon by many researchers. At its core, it is a binary tree where every inner node can be thought of as a splitting hyperplane that divides space into two along some axis. Construction time is $O(dn \log n)$ and the required space is $O(n)$. A second important result came with the BBD tree by Arya, Mount et al. [4]. It is a variation of the KD tree that uses an extra shrink operation that achieves query time $O(\log n + 1/\varepsilon^{d-1})$. An advanced implementation of the KD and BBD tree is found in the ANN library [1], constructed by Mount et al.

A conscious effort has been made to remove dependencies on ε in query time. T.M.Chan [7] and K.L.Clarkson [8] had positive results in this direction. Har-Peled in [11] proposes a faster algorithm that constructs a balanced quadtree-like structure. Each cell of this subdivision stores a representative point of S, which is an ε-nearest neighbor of any query point in the cell. Using this Approximate

[1] http://www.cs.umd.edu/~mount/ANN/

Voronoi Diagram (AVD) to answer queries, we only need to point-locate the particular cell by descending the tree. This structure answers queries in $O(\log(n/\varepsilon))$ with space $O((n/e^d)(\log n)\log(n/\varepsilon))$. It is evident that we get a much faster query time at the expense of a much bigger data structure. This work was improved by Arya, Malamatos and Mount in [1] [2] [3]. A new trade-off parameter, $\gamma \in [2, \frac{1}{\varepsilon})$ is defined in their work.
At one extreme ($\gamma = 2$) it provides time and space

$$\begin{array}{ll} \text{Time:} & O(\log n + 1/\varepsilon^{(d-1)/2}) \\ \text{Space:} & O(n\log(1/\varepsilon)) \end{array}$$

At the other extreme ($\gamma = 1/\varepsilon$) it provides time and space

$$\begin{array}{ll} \text{Time:} & O(\log(n/\varepsilon)) \\ \text{Space:} & O((n/\varepsilon^{d-1})\log(1/\varepsilon)) \end{array}$$

The key difference to Har-Peled's work is the addition of the new t parameter; each leaf is allowed at most t representatives. It is important to note that the complexities mentioned above assume that the dimension d is a constant.

1.2 Our Contribution

Our work is focused on implementing this last algorithm by Arya, Malamatos and Mount [3] and researching its potential as a practical approach to calculating approximate nearest neighbors in low dimensions. Our main result is that we are able to construct a data structure which answers queries faster and more exact compared to the fastest KD tree implementation for dimensions 3-5. Construction time and memory consumption are much greater than for KD trees but remain manageable for modern computers.

The rest of this thesis is organized as follows: in Chapter 2 we present the KD and BBD tree and in Chapter 3 we will briefly describe the Arya, Malamatos and Mount algorithm [3]. In Chapter 4 we describe the changes we made to make the memory consumption of the algorithm feasible. In Chapter 5 we briefly describe our implementation, in Chapter 6 we present interesting code snippets and in Chapter 7 we present our experimental results.

Chapter 2

KD and BBD tree

2.1 KD tree

The KD tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space. It is a binary tree in which every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane represent the left subtree of that node and points right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k-dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x-value of the point, and its normal would be the unit x-axis.

2.1.1 Construction

Since there are many possible ways to choose axis-aligned splitting planes, there are many different ways to construct k-d trees. The canonical method of k-d tree construction has the following constraints:

As one moves down the tree, one cycles through the axes used to select the splitting planes. (For example, in a 3-dimensional tree, the root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have z-aligned planes, the root's great-grandchildren would all have x-aligned planes, the root's great-great-grandchildren would all have y-aligned planes, and so on.)

Points are inserted by using a way to choose axis-aligned splitting planes. (Note the assumption that we feed the entire set of n points into the algorithm up-front.)

Here is a list of different ways to choose the axis-aligned splitting plane:

- **standard kd-tree splitting rule** The splitting dimension is the dimension of the maximum spread

of S. The splitting point is the median of the coordinates of S along this dimension. A median partition of the points S is then performed. This rule guarantees that the final tree has height dlog2 ne, and size O(n), but the resulting cells may have arbitrarily high aspect ratio.

- **midpoint splitting rule** This is a simple splitting rule, which guarantees that cells have bounded aspect ratio, and is called the midpoint splitting rule. It simply cuts the current cell through its midpoint orthogonal to its longest side. It can be seen as a binary variant of a quad tree, since with every d levels of the tree, a hypercube of side length x is partitioned into equal hypercubes of side length x=2 each. If there are ties, it selects the dimension with the maximum point spread. This rule can produce trivial splits, meaning that all of the points of S lie to one side of the splitting plane. As a result, the depth of and size of the resulting tree can be arbitrarily large, and both may even exceed n if the points are highly clustered.

- **sliding midpoint splitting rule** This is a simple modification of the midpoint splitting rule. It first attempts to perform a midpoint split, by the same method described above. If points of S lie on both sides of the splitting plane then the algorithm acts exactly as it would for the midpoint split. However, if a trivial split were to result, then it attempts to avoid this by sliding the splitting plane toward the points until it encounters the first data point. More formally, if the split is performed orthogonal to the ith coordinate, and all the points of S have i-coordinates that are larger than that of the splitting plane, then the splitting plane is translated so that its ith coordinate equals the minimum ith coordinate among all the points of S. Let this point be p1. Then the points are partitioned with p1 in one part of the partition, and all the other points of S in the other part. A symmetrical rule is applied if the points all have ith coordinates smaller than the splitting plane. This rule cannot result in any trivial splits, implying that the tree has maximum depth n and size $O(n)$. It is possible to generate a cell C of very high aspect ratio, but it can be shown that if it does, then C is necessarily adjacent to a sibling cell C0 that is fat along the same dimension that C is skinny. It turns out that cells of high aspect ratio are not problematic for nearest neighbor searching if this occurs.

- **fair splitting rule** This is a compromise between the standard kd-tree splitting rule (which always splits data points at their median) and the midpoint splitting rule (which always splits a box through its center. It is called the fair-split rule. The goal of this rule is to achieve as nicely balanced a partition as possible, provided that the aspect ratio bound is never violated. Given a cell, it first determines the sides of this cell that can be split in some way so that the ratio of the longest to shortest side does not exceed some predefined ASPECT RATIO. Among these sides, it selects the one in which the points have the largest spread. It then splits the points in the most even manner possible, subject to maintaining the bound on the ratio of the resulting cells. To determine that the aspect ratio will be preserved, we determine the longest side (other than this side), and determine how narrowly we can cut this side, without causing the aspect ratio bound to be exceeded. This procedure is more robust than either the kd-tree splitting rule or the pure midpoint splitting rule when data points are highly clustered. However, like the midpoint splitting rule, if points are highly clustered, it may generate a large number of trivial splits, and may generate trees whose size exceeds O(n).

- **sliding fair splitting rule** This is a splitting rule that is designed to combine the strengths of

both fair-split with sliding midpoint split. Sliding fair-split is based on the theory that there are two types of splits that are good: balanced splits that produce fat boxes, and unbalanced splits provided the cell with fewer points is fat.

This splitting rule operates by first computing the longest side of the current bounding box. Then it asks which sides could be split and still satisfy the aspect ratio bound with respect to this side. Among these, it selects the side with the largest spread (just as fair-split would). It then considers the most extreme cuts that would be allowed by the aspect ratio bound. This is done by dividing the longest side of the box by the aspect ratio bound. If the median cut lies between these extreme cuts, then we use the median cut. If not, then consider the extreme cut that is closer to the median. If all the points lie to one side of this cut, then we slide the cut until it hits the first point. This may violate the aspect ratio bound, but will never generate empty cells. However the sibling of every such skinny cell is fat, as with sliding midpoint split.

2.1.2 Adding elements

One adds a new point to a KD tree in the same way as one adds an element to any other search tree. First, traverse the tree, starting from the root and moving to either the left or the right child depending on whether the point to be inserted is on the "left" or "right" side of the splitting plane. Once you get to the node under which the child should be located, add the new point as either the left or right child of the leaf node, again depending on which side of the node's splitting plane contains the new node.

Adding points in this manner can cause the tree to become unbalanced, leading to decreased tree performance. The rate of tree performance degradation is dependent upon the spatial distribution of tree points being added, and the number of points added in relation to the tree size. If a tree becomes too unbalanced, it may need to be re-balanced to restore the performance of queries that rely on the tree balancing, such as nearest neighbour searching.

2.1.3 Removing elements

To remove a point from an existing k-d tree, without breaking the invariant, the easiest way is to form the set of all nodes and leaves from the children of the target node, and recreate that part of the tree.

2.1.4 Nearest neighbour search

The nearest neighbour search (NN) algorithm aims to find the point in the tree that is nearest to a given input point. This search can be done efficiently by using the tree properties to quickly eliminate large portions of the search space.

Searching for a nearest neighbour in a k-d tree proceeds as follows:

Starting with the root node, the algorithm moves down the tree recursively, in the same way that it would if the search point were being inserted (i.e. it goes left or right depending on whether the point is less than or greater than the current node in the split dimension). Once the algorithm reaches

a leaf node, it saves that node point as the "current best" The algorithm unwinds the recursion of the tree, performing the following steps at each node: If the current node is closer than the current best, then it becomes the current best. The algorithm checks whether there could be any points on the other side of the splitting plane that are closer to the search point than the current best. In concept, this is done by intersecting the splitting hyperplane with a hypersphere around the search point that has a radius equal to the current nearest distance. Since the hyperplanes are all axis-aligned this is implemented as a simple comparison to see whether the difference between the splitting coordinate of the search point and current node is less than the distance (overall coordinates) from the search point to the current best. If the hypersphere crosses the plane, there could be nearer points on the other side of the plane, so the algorithm must move down the other branch of the tree from the current node looking for closer points, following the same recursive process as the entire search. If the hypersphere doesn't intersect the splitting plane, then the algorithm continues walking up the tree, and the entire branch on the other side of that node is eliminated. When the algorithm finishes this process for the root node, then the search is complete. Generally the algorithm uses squared distances for comparison to avoid computing square roots. Additionally, it can save computation by holding the squared current best distance in a variable for comparison.

Finding the nearest point is an O(log N) operation in the case of randomly distributed points. Analyses of binary search trees has found that the worst case search time for a k-dimensional KD tree containing N nodes is given by the following equation.

In very high dimensional spaces, the curse of dimensionality causes the algorithm to need to visit many more branches than in lower dimensional spaces. In particular, when the number of points is only slightly higher than the number of dimensions, the algorithm is only slightly better than a linear search of all of the points.

The algorithm can be extended in several ways by simple modifications. It can provide the k-Nearest Neighbours to a point by maintaining k current bests instead of just one. Branches are only eliminated when they can't have points closer than any of the k current bests.

It can also be converted to an approximation algorithm to run faster. For example, approximate nearest neighbour searching can be achieved by simply setting an upper bound on the number points to examine in the tree, or by interrupting the search process based upon a real time clock (which may be more appropriate in hardware implementations). Nearest neighbour for points that are in the tree already can be achieved by not updating the refinement for nodes that give zero distance as the result, this has the downside of discarding points that are not unique, but are co-located with the original search point.

2.2 BBD tree

The BBD tree improves upon the KD tree. It is based on a hierarchical decomposition of space. The tree has guaranteed height $O(\log n)$ and subdivides space into regions of $O(d)$ complexity defined by axis-aligned hyperrectangles that are fat, that is the ratio between longest and shortest side is bounded. Space is recursively subdivided into a collection of cells, each of which is either a d-dimensional rectangle or the set-theoretic difference of two rectangles, one enclosed within the other. Each node of

the tree is associated with a cell and hence it is implicitlye associated with the set of data points lying within this cell. Each leaf cell is associated with the set of data points lying within this cell. Each leaf cell is associated with the a single point lying within the bounding rectangle for the cell. The leaves of the tree define a subdivision of space. The tree has $O(n)$ nodes and can be built in $O(dn \log n)$ time. Query time depends on a constant $c_{d,\varepsilon} \leq d\lceil 1 + 6d/\varepsilon \rceil^d$ and is $O(c_{d,e} \log n)$ for single query points.

2.2.1 Construction

The BBD tree is constructed through the repeated application of two operation: splits (like in KD tree) and shrinks (unique to the BBD tree). These are two different ways to divide a cell into smaller cells (children). A split partitions the cell along an axis-orthogonal hyperplane.A shrink partitions a cell into disjoin subcells, but uses a box rather than a hyperplane to do the splitting. For both operations, the following invariants hold:

- All boxes satisfy an aspect ratio bound (no box is either too thin or fat)
- If the parent has an inner box, then this box lies entirely within one of the two children. If the operation is a shrink, then this inner box lies within the inner child of the shrink.
- Inner boxes are sticky for their enclosing outer boxes, that is each of their faces is either sufficiently far from or else touching the outer box's corresponding face.

It is important to note that when only splits are performed, it is not possible to guarantee that the points are evenly distributed. This is remedied using shrinks. Note that a split is really a special case of shrink, where the shrinking box has $2d - 1$ sides in common with the outer box. There are two reasons for making the distinction. The first is that splitting will be necessary for technical reasons in maintaining the above inariants. The other reason is largely practical. Determining whether a point lies within a shrinking box requires $2d$ comparions in general. On the other hand, determining the side of a pliting hyperplane on which a point lies requires only one comparison. This provides huge savings when dimension increases. The BBD tree is constructed through a combination of split and shrink operations We begin with a set S of n data points in R^d. Let C denote a hypercube containing all the points in S. The root of the BBD tree is a node whose associated cell is C and whose associated set is the entire set S. The recursive construction algorithm is given a cell and a subset of data points associated with this cell. Each stage of the algorithm determines how to subdivide the current cell either through splitting or shrinking and then partitions the points among the child nodes. This is repeated until the number of associated points is at most one (or more, according to bucket size), upon which the node is made a leaf of the tree.

Given a node with more than one data points, we first consider the question of whether we should apply splitting or shrinking. As mentioned before, splitting is preferred because it is simpler, but can't guarantee that the final tree is balanced. A simple strategy is to apply splits and shrinks alternately. Experience shows that good results can be obtained when shrinking is only occasionally performed.

As with splits, there are various ways we can perform shrinks:

- **BD Simple**: This is called a simple shrink. It depends on two constants: BD_GAP_THRESH whose value is 0.5, and BD_CT_THRESH whose value is 2. It first computes the tight bounding box of the points, and computes the 2d gaps, that is, the distances between each side of the tight enclosing rectangle for the data points R(S) and the corresponding side of the cell's bounding rectangle $R(C)$. If at least BD_CT_THRESH of the gaps are larger than the length the longest side of $R(S)$ times BD_GAP_THRESH, then it shrinks all sides whose gaps are at least this large. After the shrink has been performed, all of the points of S lie in the inner box of the shrink, and the outer box contains no points. If none of the gaps is large enough, then no shrinking is performed, and splitting is performed instead.

- **BD Centroid**: This rule is called a centroid shrink. Its operation depends on two constants: $BD_MAX_SPLIT_FAC$ and $BD_FRACTION$, both of whose value are 0.5. It repeatedly applies the current splitting rule (without actually generating new cells in the tree). In each case we see which half of the partition contains the larger number of points, and we repeat the splitting on this part. This is repeated until the number of points falls below a fraction of $BD_FRACTION$ of the original size of S. If it takes more than $dim{\cdot}BD_MAX_SPLIT_FAC$ splits for this to happen, then we shrink to the inner box. All the other points of S are placed in the outer box. Otherwise, shrinking is not performed, and splitting is performed instead.

2.2.2 Nearest neighbour search

Given the query point q, we begin by locating the leaf cell containing the query point in $O(\log n)$ time by a simple descent through the tree. Next, we begin enumerating the leaf cells in increasing order of distance from the query point. We call this priority search. When a cell is visited, the distance from q to the point associated with this cell is computed. We keep track of the closest point seen so far.

Let p denote the closest point seen so far. As soon as the distance from q to the current leaf cell exceeds $dist(q,p)/(1+\varepsilon)$, it follows that the search can be terminated, and p can be reported as an approxiamte nearest neighbor to q. The reason is that any point located in a subsequently visited cell cannot be close enough to q to violate p's claim to be an approximate nearest neighbor. It can be shown that, by using an auxiliary heap, priority search can be performed in time $O(d \cdot \log n)$ times the number of leaf cells that are visited.

The number of cells visited in the search depends on d and ε, but not on n. Consider the last leaf to be visited that did not cause the algorithm to terminate. If we let r denote the distance from q to this cell, and let p denote the closest data point encountered so far, then because we do not terminate, we know that the distance from q to p is at least $r \cdot (1+\varepsilon)$. We could not have seen a leaf cell of diameter less than $r \cdot \varepsilon$ up to now, since the associated data point would necessarily be closer to q than p. This provides a lower bound on the sizes of the leaf cells seen. The fact that the cells are fat and a simple packing argument provide an upper bound on the number of cells encountered.

It is easy to extend this algorithm to enumerate data points in "approximately" increasing distance from the query point. In particular, we will show that a simple generalization to this search strategy allows us to enumerate a sequence of k approximate nearest neighbors of q in additional $O(k{\cdot}d{\cdot}\log n)$

time. We will also show that, as a consequence, the data structure can be generalied to handle point insertions and deletions in $O(\log n)$ time per update.

2.2.3 Comparison with KD tree

The empirical running times on most kinds of distributions suggest that there is little or no significant practical advantage to using the BBD tree over the KD tree. Indeed, a KD tree, enhanced with many of the recent improvements (allowing approximation errors, incremental distance calculations, and priority search) is a very good data structure for nearest neighbor searching on most data sets. However, it can perform very badly in some circumstances, especially when the data distribution is clustered in low-dimensional subspaces, as in the clustered segments distribution. Low-dimensional clustering is not uncommon in practice. An inspection of some of the other program statistics (not shown here) explains why. For this distribution, the KD tree produced a large number of cells with very high aspect ratios. Because the optimized KD tree cuts along the dimension of greatest spread, it can produce cells that are very skinny along the dimensions in which the data are well distributed, and very long in the remaining dimensions. These skinny cells violate the packing constraint, which is critical to our analysis. If the query point distribution differs from the data point distribution, then many such skinny cells may be visited by the search. This is why uniformly distributed query points were chosen. In contrast, we could have forced bounded aspect ratios by using the midpoint splitting rule, but by not allowing shrinking. The result is a sort of binary form of a quadtree. For highly clustered distributions, like clustered segments, this results in trees that are at least an order of magnitude larger than the BBD tree in both size and depth. Both variants of BBD trees took advantage of shrinking to produce reasonably small trees with cells of bounded aspect ratio. The running times are significantly better than those for the KD tree for this distribution.

Chapter 3

The Main Algorithm

We will first present the algorithm by Arya, Malamatos and Mount [3] succintly.

Given a point set $S \subseteq R^d$ consisting of n points, an approximation factor $\varepsilon \in (0, 0.5]$ and an integer $t \geq 1$ then the (t,ε) approximate Voronoi Diagram is a subdivision of space into cells such that each cell w is associated with a subset of at most t points from S, called representatives. The representatives are candidate $\varepsilon - approximate$ nearest neighbors to any point $q \in w$ and at least one of them is guaranteed to be an $\varepsilon - approximate$ nearest neighbor of $q \in S$. We also define a space-time tradeoff parameter $\gamma \in [2, \frac{1}{\varepsilon}]$.

3.1 Step 1: Preprocessing the input points

The algorithm assumes that all points are scaled and translated to lie within a sphere of radius $\frac{\varepsilon}{11}$ centered inside the unit hypercube $[0, 1]^d$. This ensures that any point $p \in S$ is an $\varepsilon - approximate$ nearest neighbor to a query point that falls outside the unit hypercube. We are going to focus on answering queries that fall inside the unit hypercube.

3.2 Step 2: WSPD

Our goal is to subdivide the unit hypercube into quadtree boxes which represent the AVD cells. These cells store no geometric information but are instead a rasterization of the traditional Voronoi Diagram. Naturally the cells need to be smaller in areas that fall near the bisector of the line segment that connects 2 input points, whereas they can be larger around individual points. To obtain this classification, we use the algorithm by Callahan and Kosaraju [6]. See Fig. 9.1 for an example.

Definition: Given a set of n points $S \in R^d$ and two subsets X, Y from S we are going to call (X,Y) a well separated pair if X, Y can be enclosed in d-dimensional balls so that the distance of the centers of the two balls is at least σr where σ is an arbitrary number called the separation factor and $r = max(r_X, r_Y)$. For our application, $\sigma = 4$ is proved to be the minimum value that still

holds the desired properties. DEFINITION: A Well Separated Pair Decomposition (WSPD) of S is the set $P_{S,\sigma} = \{(X_1, Y_1), \ldots, (X_m, Y_m)\}$ of pairs of subsets of S such that

1. $\forall i \in [1, m]$, X_i and Y_i are well separated.

2. $\forall x, y$ with $x \neq y$ $(\exists i)[(x \in X_i \wedge y \in Y_i) \vee (x \in Y_i \wedge y \in X_i)]$.

Notice that 2 single points are always well separated. The process calculates a number of dumbells, so called because joining each pair of balls with a line resembles that shape. The time complexity is $O(n \log n + \sigma^d n)$ and produces $O(\sigma^d n)$ dumbells. Notice that for a fixed dimension d the number of dumbells is linear, whereas the geometric Voronoi Diagram needs to consider $\binom{n}{2} = \frac{n!}{2!(n-2)!} = O(n^2)$ pairs. The process of computing the set of dumbells demands the construction of a Fair Split Tree (FST). According to [6], there are two algorithms to construct the tree, a simple one that is quadratic in the worst case and a more complicated one that runs in nlogn. Once the FST is available, the computation of the pairs takes time $O(\sigma^d n)$ and produces $O(\sigma^d n)$ pairs.

3.3 Step 3: Finding the overlapping quadtree boxes

We will use this list of dumbells to construct a tree structure consisting of quadtree boxes in d dimensions. A quadtree box is defined recursively to be either the unit hypercube or a hypercube obtained by splitting any quadtree box into 2^d equal parts. The size of a quadtree box is the length of its edge. The leaves of this tree structure are the cells of the AVD. We produce the quadtree boxes: For each dumbell we denote the midpoint of the line segment that connects the centers of the two enclosing spheres with z and its length with l. We draw a number of concentric spheres centered on z with radius $r_i = 2^i l$ where $i \in [0, \log(\frac{1}{\varepsilon})]$ and for each such circle we find all the overlaping quadtree boxes of side $\frac{r_i}{c_2 \gamma}$ with $c_2 \geq 20d$. Notice that the ratio of circle radius to square side remains constant. In the original paper there is a proof that dictates that this specific size should be used for boxes. See Fig. 9.3 for an example of a quadtree.

Each such quadtree box is inserted in a BBD tree like structure. This is going to be the final structure on which queries will be performed. A quadtree box can be implicitly represented in such a tree by inserting the coordinates of its 2^d vertices. Notice that this tree structure simplifies our query stage: we just need to go down the tree to determine which leaf contains our query point. Our final construction stage is to determine a number of representatives for each leaf. The final stage of the query process is thus to check the distance of the query point to all representatives of the leaf and return the nearest one.

To determine the construction time of this step, we need to calculate the number of quadtree boxes and their insertion time in the BBD tree. If d is considered a constant, then there are $O(n)$ dumbells from WSPD. The ratio of volumes of a sphere to a cube is $O(\gamma^d)$. We make about $\log(1/\varepsilon)$ circles per dumbell. By multiplying these three results we deduce that the number of quadtree boxes is $O(n\gamma^d \log(1/\varepsilon))$. The BBD tree ensures an insertion time of $O(n \log n)$ with $O(n)$ nodes. So the construction time for this stage is $O(m \log m)$ where $m = n\gamma^d \log(1/\varepsilon)$ with $O(m)$ nodes.

3.3.1 Step 3′

There is also a faster alternative which calls for inserting just the boxes that overlap the hyperplane of the bisector of the line instead of the whole sphere. This reduces the number of boxes to $O(n\gamma^{d-1}\log(1/\varepsilon))$.

3.4 Step 4: Deciding the representatives

For each leaf of the tree structure we need to calculate $t = \frac{1}{(\gamma\varepsilon)^{\frac{d-1}{2}}}$ representatives, that is a subset of input points who are candidate ε-ANN for any query point that falls inside the particular cell. To determine the subset, we calculate random helper points on a circle centered on the center of the box and of radius find the NN of a number of random helper points that lie on a circle around the particular quadtree. In case there are input points inside this circle, then one of them is added to the list of representatives. Notice that for $\varepsilon = \frac{1}{2}$, only one representative per leaf needs to be calculated.

Chapter 4

Our Variant

We define a number of abbreviations:

1. d denotes the dimension we are working in.

2. r is the maximum number of representatives a leaf is allowed to have.

4.1 Quadtree boxes that cover a sphere

Our initial task is to implement the algorithm as described in the previous section by filling in the missing details. This mainly involves finding a way to calculate the quadtree boxes that overlap a d-dimensional sphere.

To this end we use an algorithm from graphics, the Bresenham Midpoint Circle algorithm [15]. This rasterization algorithm was originally used to calculate the pixels that need to be colored to produce the circumference of a circle on screen. It doesn't require expensive angle calculations, but instead uses integer arithmetic to produce 1/8th of a circle. The rest of the circumference is created by mirroring the coordinates of each pixel. See Fig. 9.4.

For example, in order to draw the circle $x^2 + y^2 = r^2$, it first fills the pixel at $(r, 0)$ and then proceeds to the 45 degrees angle in steps that increase y by one and only occasionally decrease x. The algorithm was modified to produce the centers of quadtree boxes of filled circles and rings in any dimension. Producing filled circles requires that on each step we don't just produce the box with center (x, y), but all the boxes of the column, $(x, 0), (x, side), \ldots, (x, y)$ Producing rings demands drawing both an outer and an inner circle and filling their difference.

Extending this algorithm to cover hyperspheres of any dimension is more demanding. The idea is to divide the d-dimensional hypersphere into circle slices of different radiuses which the Bresenham algorithm can draw. During this process we can take advantage of mirroring. For example, to cover a sphere in $\mathbb{R}^3$, we need to draw circles of decreasing radius from $z = 0$ to $z = r$ and mirror every quadtree box to the negative value $-z$.

A hypersphere with center $(x_0, y_0, z_{1,0}, z_{2,0}, \ldots, z_{d-2,0})$ is described by $(x-x_0)^2+(y-y_0)^2 = r^2 - \sum_{i=1}^{d-2}(z_i - z_{i,0})^2$ First we produce set $A = \{0, side, 2 \cdot side, \ldots, r\}$. This describes possible values for each z_i variable. We calculate the cartesian product and receive a set of tuples:

$$\underbrace{A \times A \times \ldots \times A}_{\text{d-2}}$$

$$T = \{(\underbrace{0, 0, \ldots, 0}_{\text{d-2}}), (0, 0, \ldots, side), \ldots, (r, r, \ldots, r)\}$$

For each $t \in T$ so that $r'^2 = r^2 - \sum_{i=1}^{d-2} t_i^2 > 0$, we can produce a set of quadtree boxes by:

1. Calculating the x and y coordinates by running Bresenham's midpoint circle algorithm for center (x_0, y_0) and radius r'.
2. Calculating the other $d-2$ variables by mirroring each tuple value around the center of the hypersphere.

For the first step we have already mentioned that the Midpoint Circle algorithm is easy to adapt to drawing filled circles or rings. For the second step we must produce sets $B_1, B_2, \ldots, B_{d-2}$ where $B_i = \{z_{i,0} + t_i, z_{i,0} - t_i\}$ and take the cartesian product $B_1 \times B_2 \times \ldots \times B_{d-2}$. We combine every result of this step with every result of the first step to get the coordinates of a unique quadtree box.

Finally, in order to implement Step 3' we simply filter the results of the above algorithm, keeping only the ones that belong to the hyperplane of the bisector. Their characteristic property is that the vector they define is almost perpendicular to the vector defined by the dumbell. Therefore for every box of Step 3 we calculate the inner product of its vector and the dumbell's vector and only insert it if the result is close to zero.

4.2 Dealing with space requirements

After implementing the original algorithm using the generalized Bresenham, it was evident that the space required was not practical. Indeed, as little as 128 points in $d = 3$ required gigabytes of memory to build. We will first point out the reasons why this happens and then describe our method to circumvent the memory bottleneck.

Revisiting the complexity calculation of the previous section, if d is not considered a constant and $\gamma = 2$ (minimum space), then the number of quadtree boxes is exponential in d , as follows:

$$\frac{V sphere}{V cube} = \frac{\frac{\pi^{d/2}}{(d/2)!}R^d}{\frac{R^d}{(20\gamma d)^d}} \approx (700d)^d$$

If we also take into account that the number of dumbells depends on $\sigma^d = 4^d$, then, for a given number of points, increasing $d = 3$ to $d = 4$ involves calculating 26000 times more quadtree boxes

and from $d = 4$ to $d = 5$, 34000 times more quadtree boxes. So the number of boxes that need to be inserted has a very big constant which is exponential in d.

We propose a number of methods to circumvent this problem

1. We limit our approach to $\varepsilon = \frac{1}{2}$ and, as a result $\gamma = 2$ and $t = 1$. This provides the slowest query time, but also the smallest data structure. We will not bother with lower values of ε because ,as we will see in the experimental results, our data structure is extremely accurate.

2. The original algorithm recommends a BBD tree like structure to save the quadtree boxes. Instead we use a simple d-dimensional quadtree, as in [11] and the demo implementation [1]. The suggested method for coercing a BBD/KDtree to represent a single quadtree box demands the insertion of 2^d points and produces a structure that is in our experience much larger than the equivalent quadtree. A small experiment revealed that in practice the equivalent KD tree is 4 times as big as our quadtree. An initial concern was that the quadtree might have $O(n)$ depth in the worst case, but this is a problem experienced in point quadtrees, not region quadtrees, as in our case. In all cases, the maximum depth of our quadtree was measured as part of our experimental results. In Section 5 we see that this property grows

These changes are not enough to solve the memory issue. We will also introduce leaf merging: If the union of representatives of all leaves belonging to an inner node has cardinality less than or equal to r, then we can delete the children and turn the parent into a leaf. This process can be applied recursively if we traverse the tree depth first. See Fig. 9.5 for an example. Leaf merging effectively minimizes our tree structure.

We proceed to describe 2 different methods to create the quadtree. In both cases we create the tree by insertion, repeatedly putting in boxes calculated by the generalized Bresenham algorithm in Section 3.1.

Method A: Merge Step 3 and 4 into a single step

For this method we load a single representative to every box before inserting it. To insert such a box, traverse the quadtree starting from the root and in each iteration go down to the child node that contains it. This loop continues until a leaf node is found, then:

- If the leaf has no representatives, create new children for it and continue travelling down the tree the nex box is inserted.

- If it already contains representatives, check whether or not the new box's representative is already present among the leaf's representatives. If that's the case, there is nothing further to do, since we already consider the new box's representative as a candidate $\varepsilon - ANN$ for this area. Otherwise, create new children and assign to each one the representatives of their father. It's easy to show that this algorithm guarantees that we lose no representatives after

[1] http://valis.cs.uiuc.edu/~sariel/progs/avoronoi/

consecutive insertions and leaf merges, see Fig. 9.6. Finally, the new representative is added to the representatives of some leaf.

In case this addition causes the leaf to contain more than r representatives, we select the one that is further away from the center of the leaf and remove it.

After a number of insertions and with the purpose of managing the memory consuption, we apply leaf merging as described above.

This method, as we'll see in Section 5, is slow because it calculates a representative for each new box.

4.2.1 Method B: Keep Step 3 and 4 separate

In order to speed up the process, we attempted to keep step 3 and 4 separate. We add a new height variable h to each quadtree node. The meaning of $h = 3$ in a leaf node is that the leaf is the root of a complete subtree with height 3.

This time we insert quadtree boxes without representatives. Again we periodically apply a variant of leaf merging that supresses full subtrees by changing the height variable of their root and deleting the rest of the tree. Once this step is completed, we can continue to step 4. We visit all leaves of the tree and assign a representative to each one. If a leaf with height greater than zero is found, we construct the corresponding subtree. As leaves are filled with representatives, we apply leaf merging to keep memory consumption low.

This method proved to be much faster than the previous one and in our final version of the application we maxed out performance by implementing Step 3'.

Chapter 5

Our Implementation

In this section we will describe the main steps of our implementation. The input consists of n points in d and a maximum number of representatives per leaf, r.

1. Scale and translate the input points into a circle of radius $\frac{\varepsilon}{11}$ centered inside the unit hypercube $[0,1]^d$. This process demands the computation of the minimum enclosing sphere of the original points. To this end we use Miniball v2.0 [10] and [1], a library that finds the smallest enclosing ball of a set of points in linear time.
2. Find the well separated pairs. We refer to the original paper [6] for the actual algorithm and our description in Section 3. The simple of algorithm we mentioned in Section 2.2 is used for the construction of the fair split tree, a choice which doesn't incur a time bottleneck since this calculation is extremely fast compared to the rest of the process.
3. Construct the quadtree. We calculate the overlapping quadtree boxes by using the generalized Bresenham algorithm we described in Section 3.1.

5.1 Method A

For this method every box is assigned a single representative. This representative is decided by running an exact search on the input points on an ANN KD tree using the center of the quadtree as the query point.

Initially, the program gave every box a representative before trying to insert it. An improvement of 20% in construction time was achieved by making this decision lazy: the calculation of the new box's representative is done only if a leaf node containing representatives is reached, and thus this decision needs to be made, according

In an effort to make this step faster, we parallelized it. Our initial approach was to dynamically allocate dumbells to threads and to allow them to insert quadtree boxes in a common quadtree.

[1] http://www.inf.ethz.ch/personal/gaertner/miniball.html

In this solution, each thread constructs a local KD tree and once it has computed the representatives of a number of new boxes, it locks the whole quadtree and runs the insertion algorithm. This parallelization yielded a 40% speedup over the serial instance of the program with just 4 threads and didn't consume any extra memory. Still this is not the best parallelization method. Our final version removes the global quadtree and the lock, and instead allows each thread to have a local quadtree. As a result, all threads are working at 100% all of the time, but memory consumption is also multiplied, roughly by the number of threads. Once all threads have completed, the individual quadtrees are merged into one final global quadtree. The merging algorithm is very intuitive: we keep the first quadtree as an accumulator and merge the other ones in it. For each quadtree, we traverse both itself and the accumulator with DFS.

(a) If both nodes are inner nodes, we recurse to the children

(b) If the accumulator has an inner node and the quadtree has a leaf, there is nothing to do

(c) If the accumulator has a leaf and the quadtree has an inner node, we copy the subtree from the quadtree to the accumulator

(d) If both nodes are leaves, we copy the quadtree leaf's representatives to the accumulator's representatives and remove the ones that are further away from the center of the quadtree box in case the cardinality of the union is bigger than r

The merging phase takes negligible time (less than 1% for 4 threads) compared to the overall construction time and its use leads to a 50% improvement in construction time for 4 threads compared to the previous parallel version.

Every so often we apply our merging function. Generally, the frequency of application acts as a tradeoff parameter between maximum space required and construction time. We call the merging function every time the quadtree becomes too large for the memory of the current machine to handle without swapping: since we know the exact footprint of one node of the tree, we have a specific limit of nodes that can be handled in a given amount of memory. The number of nodes in the tree is periodically calculated and the merging function is called as necessary.

5.2 Method B

For this method boxes are inserted without representatives.

In the first phase, which corresponds to Step 3 of the algorithm, we insert blank boxes in quadtrees. This step is parallelized, each thread is dynamically assigned a dumbell and computes all the boxes that cover the spheres associated with it. The variant of leaf merging described in Section 3.2-Method B is applied when the tree gets too big for the computers main memory. When all threads are done, we merge the individual trees as in Section 4.1-Method A. We thus end up with a quadtree that contains no representatives and in which complete subtrees have been compressed to their root.

In the second phase, which corresponds to Step 4 of the algorithm, we calculate the representatives of the quadtree. Again we parallelize the computation, assigning a number of children of the quadtree root to every thread. Each thread visits every leaf of each child, expands the compressed subtrees into a complete subtrees of height h and calculates representatives for the empty leaves. We apply leaf merging as before to manage the memory consumption.

Chapter 6

Code Snippets

It is interesting to see specific parts of the code.

1. This is the QNode class that makes up our quad trees

   ```
   class QNode {
   public:
     QNode** children;
     myset* representatives;
     int depth;
   };
   ```

   It is brought down to its essentials. We only keep a pointer to an array of children and a pointer to the set of representatives. This essentialy means that our node costs only 16 bytes of memory on a 64bit system. There are certain reasons that led us to implement it this way:

   (a) We can't just keep the children in a QNode* children because that causes all the children nodes to lie next to their parent in memory, actually nested between their parent and their uncle. However, it is crucial we are able to delete and create nodes anywhere in the tree and the only way to achieve that is by ...prekageo...

   (b) We keep each unique set of representatives only once and make a pointer to it. This is the most sane way to avoid using excess space.

   (c) We keep no information about the coordinates of the center of the square or the size of the side of the square. As we will see, these are not necessary, we can just compute them as we go down the tree from the root, where center coordinates and size are known.

2. C++ doesn't have a library that calculates cartesian products. We give an example of an iterative method of computing the cartesian product of a vector of vectors of integers. We use a vector of iterators, each one iterates over an individual vector of integers. Each iterator knows it's current position, it's starting position and it's final position.

```
typedef std::vector<int> Vi;
typedef std::vector<Vi> Vvi;

struct Digits {
  Vi::const_iterator begin;
  Vi::const_iterator end;
  Vi::const_iterator me;
};
typedef std::vector<Digits> Vd;
```

Then initialize all the iterators at the beginning. Make a copy of the current state of iterators, then increase the rightmost one. Continue making copies until the rightmost one reaches the end, reset it and increase the next to last one by one. Continue in this fashion until all iterators have reached the end.

```
void cart_product(Vvi& out,Vvi& in) {
  Vd vd;

  // Start all of the iterators at the beginning.
  for(Vvi::const_iterator it = in.begin();
    it != in.end();
    ++it) {
    Digits d = {(*it).begin(), (*it).end(), (*it).begin()};
    vd.push_back(d);
  }

  while(1) {
    Vi result;
    for(Vd::const_iterator it = vd.begin();
      it != vd.end();
      it++) {
      result.push_back(*(it->me));
    }
    out.push_back(result);

    for(Vd::iterator it = vd.begin(); ; ) {
      ++(it->me);
      if(it->me == it->end) {
        if(it+1 == vd.end()) {
          return;
        } else {
          it->me = it->begin;
          ++it;
        }
```

```
      } else {
        break;
      }
    }
  }
}
```

3. We present the basic Bresenham midpoint circle function It takes the center coordinates (x_0, y_0) and the radius as input. The ddF_x variable denotes the error on the x axis and the ddF_y variable denotes the error on the y axis. Also, f is the total error.

```
void rasterCircle(int x0, int y0, int radius) {
  int f = 1 - radius;
  int ddF_x = 1;
  int ddF_y = -2 * radius;
  int x = 0;
  int y = radius;

  setPixel(x0, y0 + radius);
  setPixel(x0, y0 - radius);
  setPixel(x0 + radius, y0);
  setPixel(x0 - radius, y0);

  while(x < y) {
    if(f >= 0) {
      y--;
      ddF_y += 2;
      f += ddF_y;
    }
    x++;
    ddF_x += 2;
    f += ddF_x;
    insertBox(x0 + x, y0 + y);
    insertBox(x0 - x, y0 + y);
    insertBox(x0 + x, y0 - y);
    insertBox(x0 - x, y0 - y);
    insertBox(x0 + y, y0 + x);
    insertBox(x0 - y, y0 + x);
    insertBox(x0 + y, y0 - x);
    insertBox(x0 - y, y0 - x);
  }
}
```

4. This is how we retrieve this information when we go down the tree. qb points to the current

node, qb_center is its center coordinates and qb_side is its side. Finally leaf_center is an array of the center coordinates of the input quadtree box (the one we want to descend to) and child_addr is the address (number of array cell) that we select among the children of the current node to descend to. Initially qb points to the root, qb_center=0.5 $\forall i \in [0, dim]$ and qb_side is 1.0.

```
child_addr=0;
it=1;
for (i=0; i<dim; i++) {
  if (leaf_center[i] > qb_center[i]) {
    child_addr = child_addr | it;
    qb_center[i] += qb_side;
  } else {
    qb_center[i] -= qb_side;
  }
  it = it << 1;
}
qb=&(qb->children[0][child_addr]);
qb_side = qb_side/2.0;
```

As we see, bitwise operations are employed to make this step as fast as possible. For example, if we are at the root then qb_center is $(0.5, 0.5, 0.5)$ for 2D and we want to to $(0.75, 0.25, 0.75)$. For the x coordinate the if clause is activated and child_addr gets the value $0|1 = 1$. Then $it = 1 << 1 = 2$ and when the loop checks the y coordinate the else clause is activated which leaves child_addr the same. Finally, $it = 2 << 1 = 4$ and the if clause is activated so child_addr gets a final value of $1|4 = 5$. We can also check the correctness of the array boundaries: For input $(0.25, 0.25, 0.25)$ obviously all checks will fail so child_addr will point to 0. For input $(0.75, 0.75, 0.75)$ all checks will succeed so child_addr will point to $(((0|1)|2)|4) = 7 = 2^3 - 1$.

5. Parallelization is achieved through the use of boost threads. The use of boost threads is very easy. The code that needs to be executed as a single thread is written inside a class. The class should include:

   - A constructor, a function which is executed once on thread startup.
   - An operator() function, executed right after the constructor.
   - A destructor, a function which is executed once on thread completion.

   Outside the class, we just need to initialize the right number of threads, commence them and then wait for their completion. Communication between the threads and the rest of the code is achieved by using global variables.

```
struct Worker {
  Worker(){allocate resources};
```

```
  operator()(){run functions};
  ~Worker(){destroy resources}
}
boost::thread* threads[num_threads];
for (int i=0; i<num_threads; ++i)
  threads[i] = new boost::thread(Worker(i));
for (int i=0; i<num_threads; ++i) {
  threads[i]->join();
  delete thr[i];
}
```

6. This is how we dynamically allocate WSPD pairs to threads. All WSPD pairs are stored in a global list and each thread must dynamically get the index of the next WSPD it's going to work on. We keep a global wspd index and a global boost::mutex. When a thread needs a new WSPD pair, it first locks the mutex, gets the index number, increases the index and releases the lock. If the number it acquires is greater than the length of the list, that is the number of WSPD pairs, then the thread terminates.

```
int global_index=0;
boost::mutex mutex;

struct Worker {
  int get_index() {
    boost::mutex::scoped_lock lock(mutex);
    global_index++;
    if (global_index < Globals::wspd_centers.size()) {
      return global_index;
    } else {
      return -1;
    }
  }

  operator()() {
   for (int priv_ind=get_index(); priv_ind!=-1; priv_ind=get_index();) {
      //work on Globals::wspd_centers[priv_index]
    }
  }
}
```

Chapter 7

Experimental Results

We recall our notation:

1. n is the number of input points
2. d denotes the dimension we are working in.
3. r is the maximum number of representatives a leaf is allowed to have.

We will initially present our testing methodology. Once the final quadtree is constructed, we make a number of queries to a KD tree, asking for an exact and an approximate answer, and to our AVD structure. Each time we call the corresponding search functions by providing a single query point and request a single nearest neighbor. We measure the aggregate *query time* in each case and also compute the *query exactness* of each of the two approximate methods. This is done by comparing the exact answer of the ANN KD tree to the two approximate answers by calculating $\frac{approx}{exact} - 1$.

For each quadtree, we make 10^9 queries, consisting of random points equally distributed inside each leaf of our quadtree. Thus, not only are our queries all inside the unit hypercube, but for a bigger quadtree, our queries are more concentrated near the input points, where most AVD cells lie, than near the edges of the hypercube. For smaller quadtrees, each leaf covers a bigger area and our queries are closer to being random over the whole hypercube. As we will see this affects the behaviour of the approximate KD tree queries.

In our results we are interested in the construction time, the maximum memory required by each thread, the size of the final data structure, its maximum height, query time and query exactness.

There are 3 main KD tree implementations:

1. ANN, a mature implementation, previously mentioned in Section 1.
2. CGAL, a computational geometry library that also includes a KD tree implementation [1].

[1] **CGAL**,Computational Geometry Algorithms Library, http://www.cgal.org

3. Libnabo, a new, faster implementation of KD trees based on the same algorithms as ANN, but different data structures [2].

We should justify our choice to compare with ANN. Using the methodology described above we found that the ANN KD tree is faster than the ones in CGAL and Libnabo. This is in accordance with the comparison done by the authors of Libnabo [9]. Libnabo is $10-20\%$ faster than ANN in the special case where a large number of queries is given to the search function all at once, not one by one as in our experiments. In all cases, we'll see that AVD is at least 50% faster than the ANN KD tree and should therefore be faster than Libnabo as well. See Fig. 9.7.

Note that our comparison to KD trees instead of BBD trees should not affect our experimental results. BBD trees are faster than KD trees in specific distributions, when data is clustered in low-dimensional subspaces [4]. We experiment with distributions that are uniform over all dimensions and therefore no significant difference between the two.

Lastly, our testing of the efficiency of parallelization was limited to 4 threads because of hardware constraints. All experiments were conducted using this number of threads.

7.0.1 Method A

Initially we test our program with input points in 3 dimensions and for various values of r. We see that an increase in r results in lower construction time, lower max space, a smaller data structure and slightly increased query exactness. On the other hand, query time is also slightly increased, since there are more points to check sequentially for each search. Also, query time in KD trees increases more rapidly with an increase in the number of input points. If query points are more uniformly distributed inside the unit hypercube, KD trees gives answers faster, but lose in exactness. Our conclusion from the above is that in no case should $r=1$ be used. Otherwise, the choice of r is a tradeoff between query time and the rest of the characteristics that we measure. See Fig. 9.8.

We also examine the speedup achieved by using multiple threads. We run the same test ($d=3$,$n=128$,$r=4$) for $t \in [1,4]$ and obtain Fig. 9.9. Using Amdahl's law [3], $S(N)=\frac{1}{(1-P)+\frac{P}{N})}$, and substituting $S=2.8$ and $N=4$ we calculate that our implementation is 70% parallelizable and a maximum speedup of 3.3 can be achieved.

For the rest of the experiments we decide to use $r=4$.

7.0.2 Method A′

There are numerous applications for which a bigger value than the maximum $\varepsilon=0.5$ covered in the theoretical analysis is adequate. At the same time we want to speed up the construction process. We proceed to describe a practical approach that makes construction feasible both for more points and bigger dimensions, but at the loss of exactness.

[2] **Libnabo**, https://github.com/ethz-asl/libnabo

[3] http://en.wikipedia.org/wiki/Amdahl's_law

1. We define the variable s which is the factor by which the size of the quadtree boxes is multiplied by. For $s = 1$, we insert boxes as big as the original algorithm proposes, for $s = 2$ we calculate boxes that are twice as large etc.
2. Once the construction of the individual quadtrees and their merging in the final quadtree has been completed, we will examine every leaf and make sure it contains exactly r representatives. This is easy to do by searching a KD tree for the r nearest neighbors to the center of the box and adding them one by one until the leaf is filled.

We run our tests as before. As we see in Fig. 9.10, construction is greatly accelerated, enabling us to test for many points. As before, max space and tree size are very well contained. It is difficult to find a pattern for the maximum error, however we find that 99.9% of all answers are at most 10% wrong. The maximum depth of our tree seems to be about $O(logn)$, much less than linear.

We continue our experiments with dimensions 4 and 5 in Fig. 9.11 and Fig. 9.12. These are dimensions that we wouldn't be able to test for without big values of s. The results that we obtain for dimension 4 lead us to the same conclusions as with dimension 3. However, for $d = 5$ it becomes evident that as the dimension increases, AVD is not that much faster than the ANN KD tree, but it is much more exact.

7.0.3 Method B

For this method we can use bigger boxes to cover spheres without losing the upper bound of the maximum error like we did in Method A'. We insert bigger boxes (again, using the notion of s) but each leaf is now assigned a height $h = s - 1$. For example instead of inserting boxes that cover a sphere ($s = 1$) and mark the leaves with $h = 0$, we could cover the sphere with boxes twice as big ($s = 2$) and mark the leaves with $h = 1$. In Step 4 of this Method, these leaves are expanded to complete trees of height h, therefore we calculate representatives for boxes of the correct size and no information is lost.

In Fig. 9.13 we experiment with different values of s for $d = 3$, $n = 256$, $r = 4$. Based on the figure, we should select between $s = 4$ and $s = 8$ for our further experiments. Total time is less for $s = 4$ but time spent on Step 3' (Total time minus Step 4) is longer compared to $s = 8$. Bearing in mind that Step 3' grows faster than Step 4, we select $s = 8$ for the rest of our experiments.

We proceed into testing our application for $d = 3$ with our standard testing methodology, see Fig. 9.14. Construction time has been reduced greatly. For example, the construction time for $n = 1024$ has dropped from 5 hours using Method A to 5 minutes. We also notice that most of the time is taken up by Step 4, the calculation of representatives. What is more, Step 3' grows a lot faster with more points compared to Step 4. Maximum required space grows linearly and the tree is much smaller than with Method A. Maximum depth of the final quadtree is again about $O(logn)$. Lastly, the AVD is twice as fast a the approximate KD tree in answering queries, while giving the exact answer 99.9% of the time.

Compared to the previous methods, not only Method B much faster to construct for $d = 3$, it also yields a smaller final data structure. This is manifested in its lower space consumption, smaller depth and results in faster query times.

We also test our data structure with queries that are uniform on the hypercube. As before, we calculate 10^9 query points and check the query time and exactness of answers. The results are shown in Fig. 9.15. The KD tree is much faster in this test, but still slower than AVD (AVD is 35% faster). However it maintains the same level of exactness, whereas the KD tree shows a huge drop, with at most 40% of its answers being exact. It is important to note however that the error of the KD tree is not large and in these tests about 99% of its answers are at most 10% wrong.

Method B is fast enough that we can try building a tree for dimension 4. The results are shown in Fig. 9.16. Although the experimental results in query time and error

7.0.4 Method Summary

We summarize the differences in experimental results between the different methods:

- **Method A**: Requires the most time to construct.
- **Method A'**: Construction is accelerated at the expense of the upper bound of error. Dimensions higher than 3 are feasible.
- **Method B**: Fastest construction and smallest data structure. Also fastest and most accurate queries. Practical for $d = 3$.

Chapter 8

Conclusion

In this paper we presented an implementation of a new algorithm that constructs Approximate Voronoi Diagrams aimed at Nearest Neighbor Searching and compared it to the best current KD tree implementations.

We sum up the great advantages of the AVD data structure. For dimension 3 and Method B, we conclude that AVD is significantly faster and more exact than KD tree. These properties are maintained whether the query points are concentrated near the input points or are random over the hypercube. Depending on the query set, the KD tree is either relatively fast or relatively accurate, but not both at the same time. The size of the final AVD data structure, although much greater than that of a KD tree, is very manageable for modern computers using the methods we proposed. The same applies for construction time. For dimensions 4 and 5 and based on Method A′, the same properties are true, but at the expense of the upper bound on maximum error.

There are a number of disadvantages in our implementation. Even for dimension 3 we believe that construction time could be faster. Currently, a lot of boxes are calculated multiple times and thus slow down the process without enriching the final data structure One way to address this problem would be to merge dumbells that are very close together and have spheres that overlap to a large extent. What is more, using a library that is faster than ANN, such as Libnabo with multiple query points on each call of the search function would also speed up construction. Our Method A′ applied for dimensions 4 and 5 could be combined with the ideas of Method B to yield a fast Method B′.

It should also be mentioned that AVD does not yet support incremental addition or removal of input points. These are methods that are desirable by users and further research is needed for their implementation. The structure itself is more static than a classic KD/BBD tree, since the desired approximation factor ε needs to be known before construction, whereas KD tree supports it as a variable specified in query time.

At the current stage of research we believe that AVD is a competitive option in the following scenario:

1. The input points are 3 dimensional and static.

2. A very large number of queries is going to be performed.
3. Exactness of answers is crucial to the process..

For our future work we want to answer a number of question that our research poses:

- How does the AVD data structure compare to KD tree and other solutions in actual applications that require approximate nearest neighbor searching and specifically for $d = 3$?
- What is the theoretic upper bound of the approximation factor when boxes of bigger size are inserted?
- Are there applications where an upper bound on the exactness of answers is not required as long as almost all results are very close to exact?

Chapter 9

Appendix

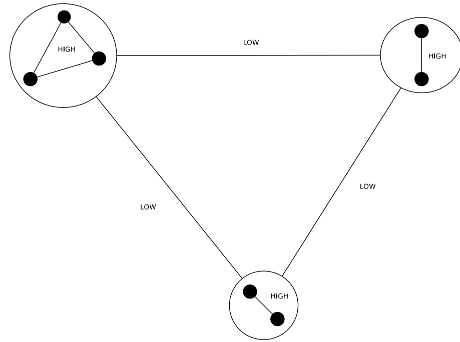

Figure 9.1: high resolution (small squares) inside big subsets

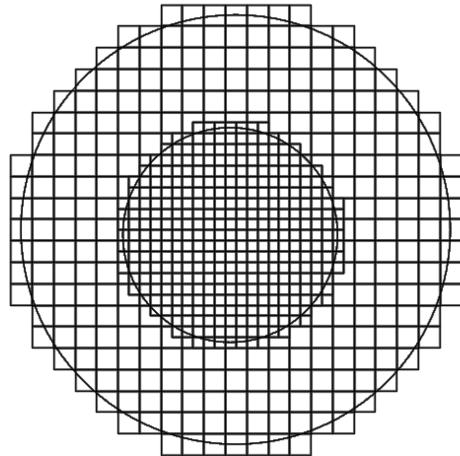

Figure 9.2: An example of quadtree boxes that overlap circles in 2D

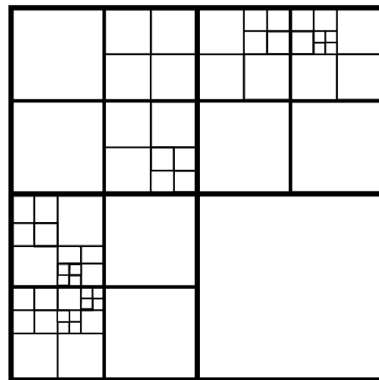

Figure 9.3: An example of a quadtree

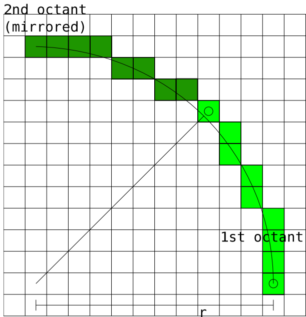

Figure 9.4: The original Midpoint Circle Algorithm (from Wikipedia)

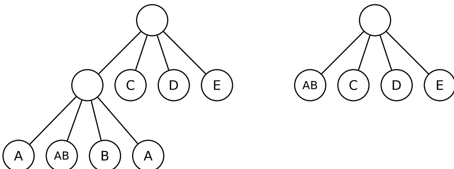

Figure 9.5: An example of merging for $d = 2$ and $r = 2$.The tree on the right can't be merged because there are 5 different representatives.

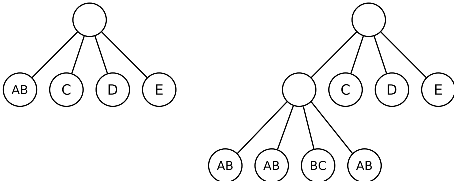

Figure 9.6: We try to insert a small square with repr "C". The new children inherit the parent's representatives, a superset of their original representatives."A" is further away than "B" and is removed.

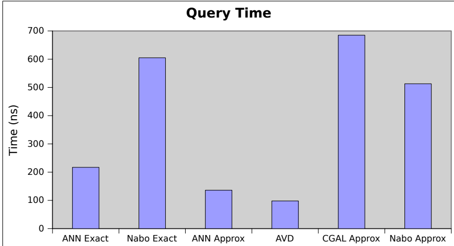

Figure 9.7: Comparison of different KD tree implementations and AVD ($d = 3$,$n = 256$,$r = 4$)

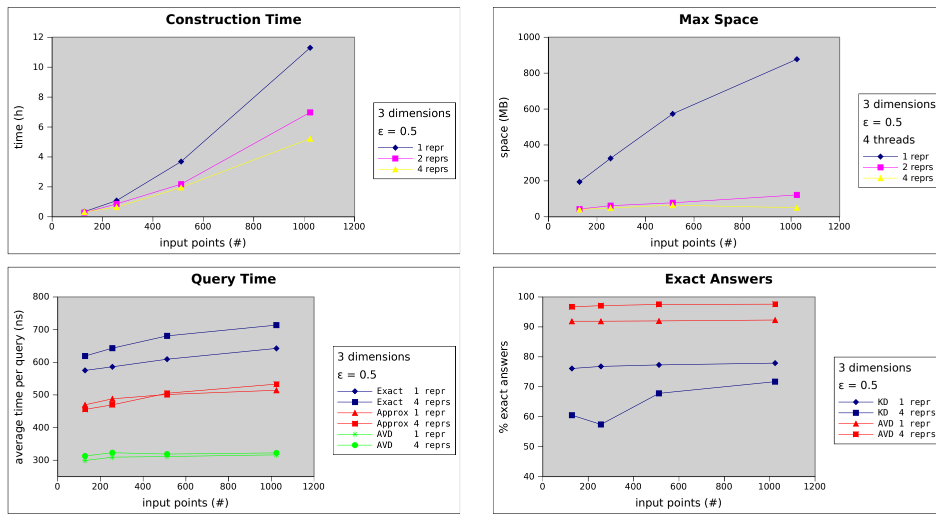

Figure 9.8: Experimental results for Method A, $d = 3$

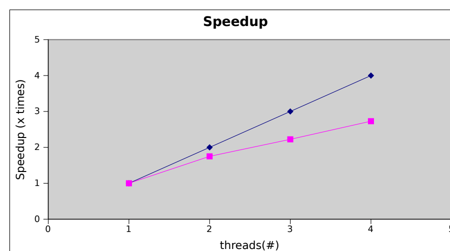

Figure 9.9: Speedup achieved with 1-4 threads against maximum speedup

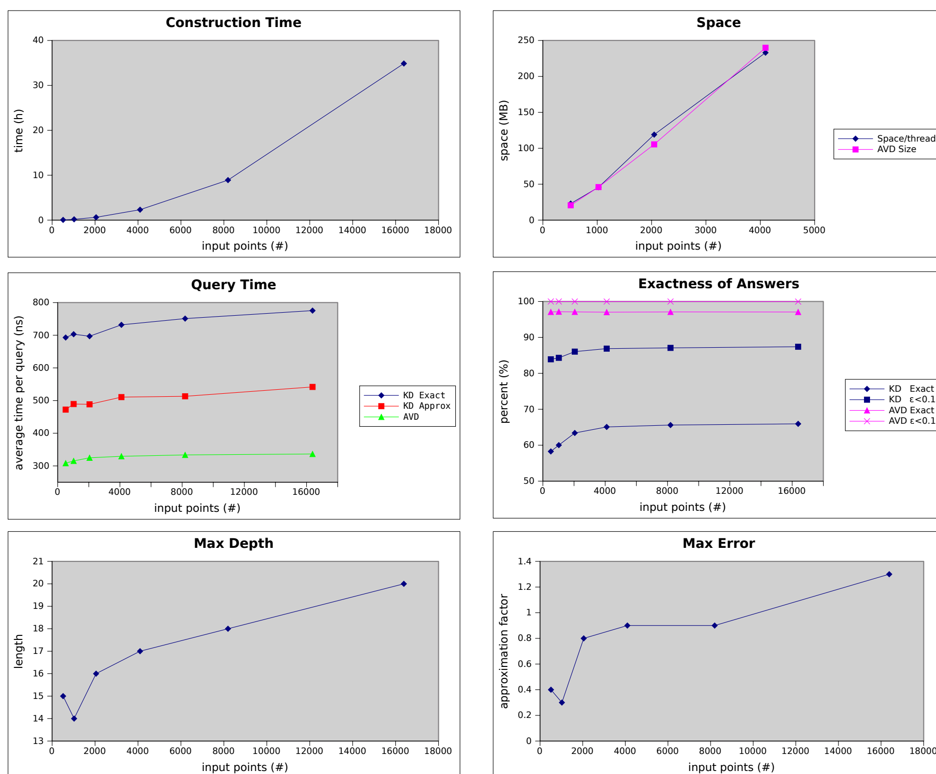

Figure 9.10: Experimental results for Method A′, $d = 3$, $r = 4$, $s = 4$

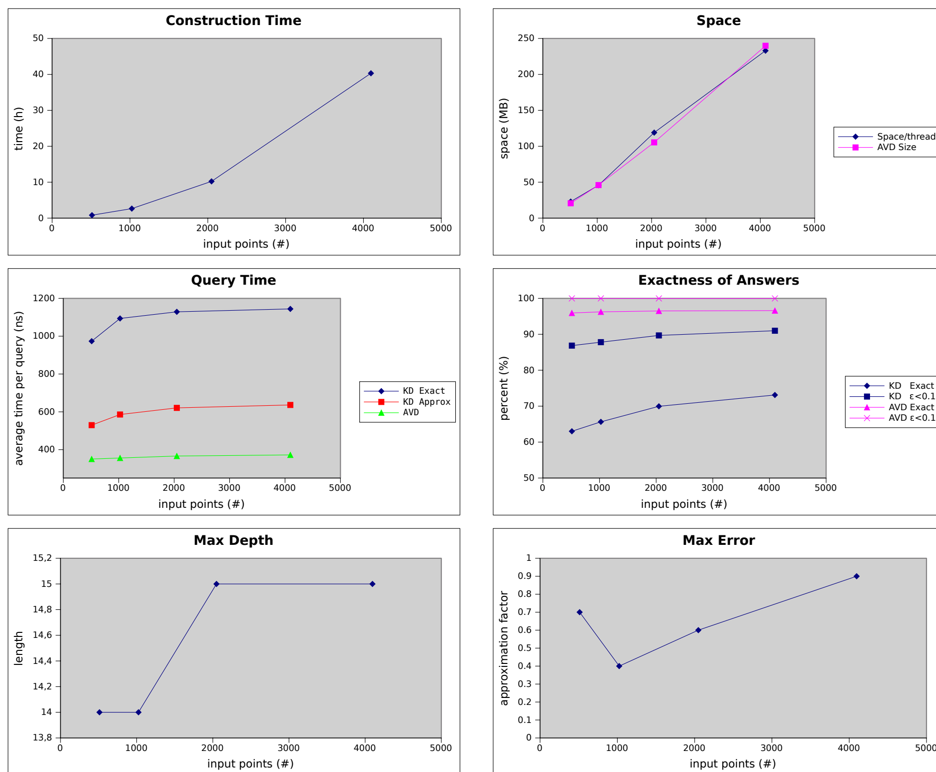

Figure 9.11: Experimental results for Method A′, $d = 4$, $r = 8$, $s = 8$

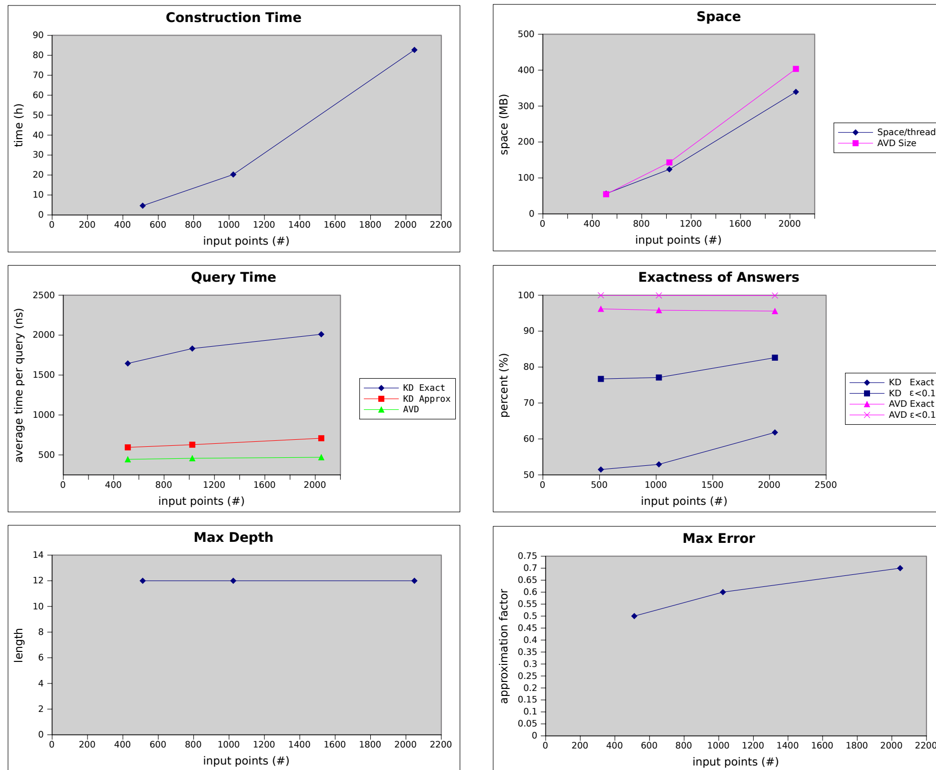

Figure 9.12: Experimental results for Method A′, $d = 5$, $r = 16$, $s = 16$

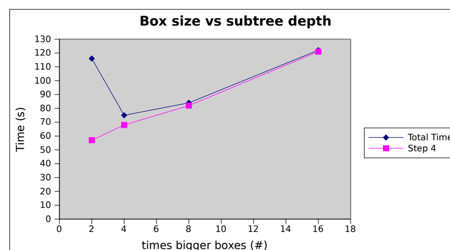

Figure 9.13: Selecting the best value for s in Method B

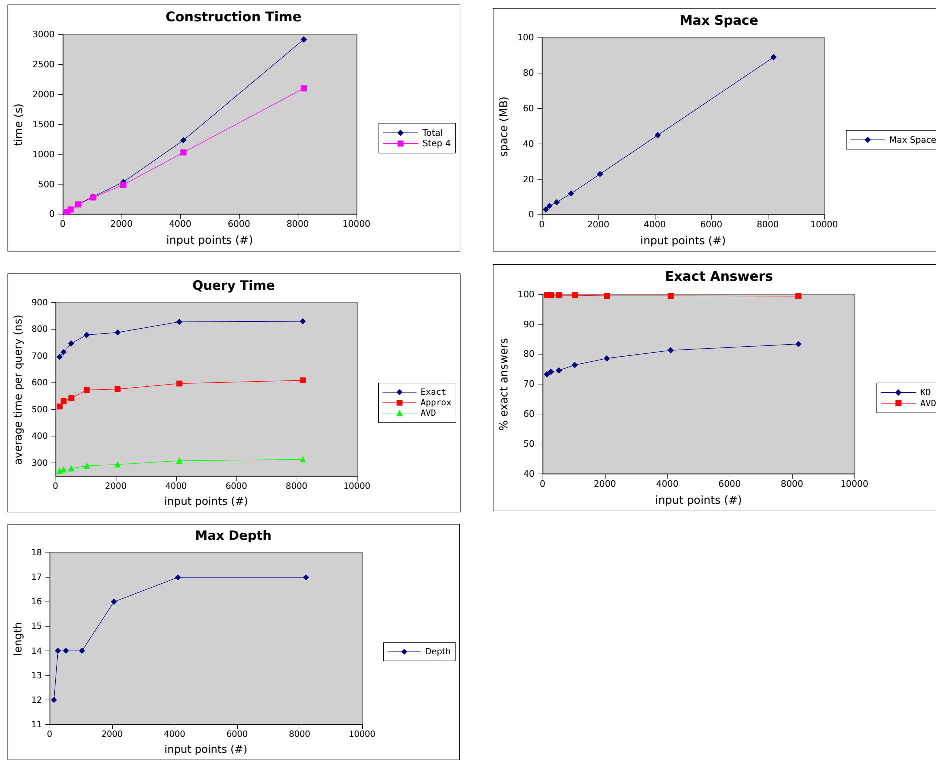

Figure 9.14: Experimental results for Method B, $d = 3$

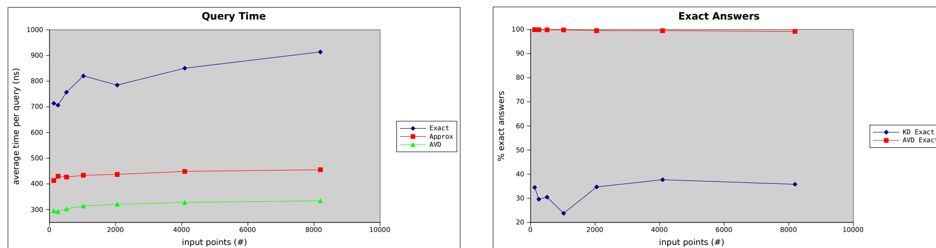

Figure 9.15: Experimental results for Method B, $d = 3$ with random queries over the hypercube

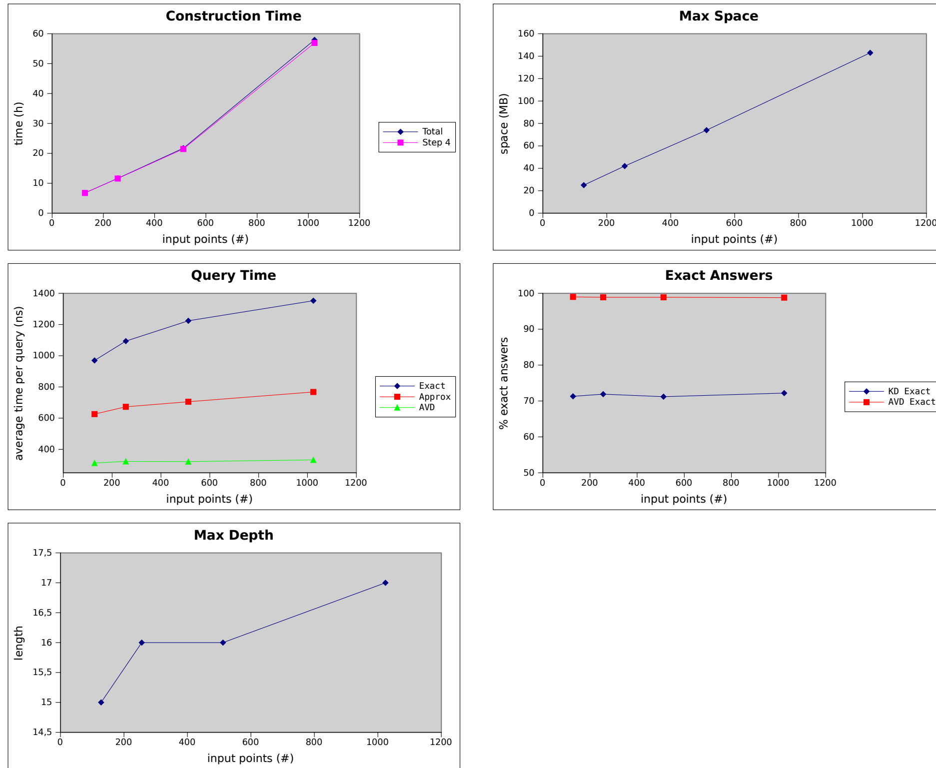

Figure 9.16: Experimental results for Method B, $d = 4$

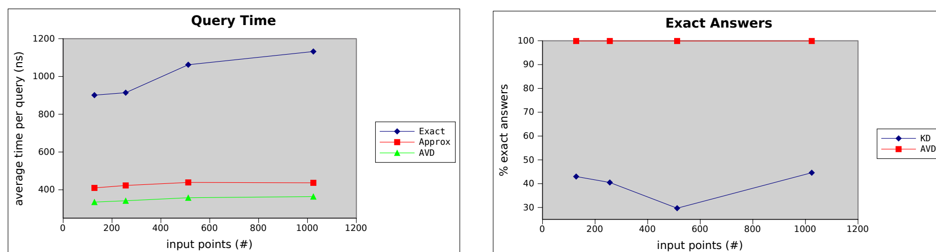

Figure 9.17: Experimental results for Method B, $d = 4$ with random queries over the hypercube

Bibliography

[1] Sunil Arya and Theocharis Malamatos. Linear-size approximate voronoi diagrams. In David Eppstein, editor, *SODA*, pages 147–155. ACM/SIAM, 2002.

[2] Sunil Arya, Theocharis Malamatos, and David M. Mount. Space-efficient approximate voronoi diagrams. In John H. Reif, editor, *STOC*, pages 721–730. ACM, 2002.

[3] Sunil Arya, Theocharis Malamatos, and David M. Mount. Space-time tradeoffs for approximate nearest neighbor searching. *J. ACM*, 57(1), 2009.

[4] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[5] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[6] Paul B. Callahan and S. Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42(1):67–90, 1995.

[7] Timothy M. Chan. Approximate nearest neighbor queries revisited. *Discrete & Computational Geometry*, 20(3):359–373, 1998.

[8] Kenneth L. Clarkson. An algorithm for approximate closest-point queries. In *Symposium on Computational Geometry*, pages 160–164, 1994.

[9] Jan Elseberg, Stephane Magnenat, Roland Siegwart, and Andreas Nuchter. Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics*, 2012.

[10] Bernd Gärtner. Fast and robust smallest enclosing balls. In Jaroslav Nesetril, editor, *ESA*, volume 1643 of *Lecture Notes in Computer Science*, pages 325–338. Springer, 1999.

[11] Sariel Har-Peled. A replacement for voronoi diagrams of near linear size. In *FOCS*, pages 94–103. IEEE Computer Society, 2001.

[12] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In Jeffrey Scott Vitter, editor, *STOC*, pages 604–613. ACM, 1998.

[13] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In Alpesh Ranchordas and Helder Araújo, editors, *VISAPP (1)*, pages 331–340. INSTICC Press, 2009.

[14] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *CVPR*. IEEE Computer Society, 2008.

[15] J.R. Van Aken. An efficient ellipse drawing algorithm. In *CG&A*, pages 24–35, September 1984.