Difficulty-sensitive point location

van Mierlo, B.C.J.

Award date:
2013

Technische Universiteit
Eindhoven
University of Technology

Technical University Eindhoven

Master thesis

Difficulty-Sensitive Point Location

Author:
Bart van Mierlo

Supervisor:
prof. dr. Mark de Berg

April 15, 2013

Abstract

In this thesis we study the point-location problem in 1-dimensional and 2-dimensional space. We show how to construct data structures such that queries with points far from the boundary of the region containing them are answered faster than queries with points close to the region boundary. More precisely, we present point-location structures that have $O(\log(\frac{1}{\delta_q}))$ query time, where δ_q is the distance from the query point to the nearest region boundary relative to the size of the domain of the space.

Contents

1 Introduction

Point location is one of the most fundamental problems in computational geometry and found in many applications. When car navigation software receives gps-coordinates of your current location it uses point location to find the street you are driving through. When clicking with your mouse on a computer screen point location is used to find out on what button, window, etcetera you clicked. The point-location problem is defined as follows. Let S be a subdivision of a 2- or higher dimensional space into regions. A point-location query is to find, for given query point q, in which region the point q lies. To answer such queries quickly, one can pre-process the subdivision into a suitable point-location data structure. In this thesis we restrict ourselves to 2-dimensional subdivisions with polygonal regions. The standard point-location structures [7, 8, 10] in this case need $O(n)$ storage and have $O(\log n)$ query time, where n is the number of edges of the subdivision. Intuitively, it should be easier to answer a point-location query if the query point q lies inside a large region, or when it lies far from the boundary of any region. The goal of this thesis is to answer the following two research questions:

1. Can we get faster query times if a query point is inside a large region?
2. Can we get faster query times if a query point is far away from the nearest region boundary?

These questions are closely related to so-called *entropy-sensitive* point location. Here each region of the subdivision has a probability associated to it (the probability that a query point falls into that region) and the goal is to minimize the expected query time. More precisely, let S be a subdivision with regions $R_1, \ldots, R_m$ and query point q. Let p_i be the probability that $q \in R_i$. Then $\text{entropy}(S) = \sum_{i=1}^{m}(p_i \log(\frac{1}{p_i}))$. Let T be a point-location structure for a subdivision S whose domain has unit area. If T has an expected query time of $O(\text{entropy}(S))$ it is entropy sensitive. Now if T has a query time of $O(\log(\frac{1}{\text{area}(R_i)}))$, where R_i is the region in which q lies, then it has faster query times if the query point is inside a large region. If we take $p_i = \text{area}(R_i)$, then the query time is $O(\log(\frac{1}{p_i}))$. The expected query time is then $O(\mathit{entropy}(S))$.

Chapter 2 provides some background information on all point-location structures discussed in this thesis, except for the MDBST and MDBST-2D point-location structures which will be introduced in this thesis.

In Chapter 3 we look at 1-dimensional point location. Analysis of the OBST shows the query time for a point q is $O(\log(\frac{1}{p_q}))$, where p_q is the probability of the region in which the query point lies. Therefore we can say that using an OBST we get faster query times when the query point is inside a large region. Note that this implies that if we define the probability of a region to be equal to its relative size, then it has faster query times if a query point is inside a large region and the expected query time is $O(\text{entropy}(S))$. It is also evident that it has faster query times if a query point is far away from the nearest region boundary, because a query point far away from the nearest region boundary implies it is inside a large region. So the OBST has all the desired properties. However its construction time is $O(n^3)$. To improve this, we introduce a 1-dimensional point-location structure called MDBST. This structure has $O(n \log n)$ construction time and like the OBST it has $O(\log(\frac{1}{p_q}))$ query time. This again

means you have faster query times when the query point is either inside a large region or far away from a region boundary. And of course it also has a $O(\text{entropy}(S))$ expected query time. We compare the OBST and MDBST in an experiment which shows, as expected, that MDBST is constructed much faster and has an only slightly worse expected query time. The relative difference in expected query time even seems to decrease when increasing the number of regions in the subdivision.

In Chapter 4 all the 2-dimensional point-location structures mentioned in Chapter 2 are analysed with respect to the research questions. When we make the probability of a region equal to its relative size, we achieve $O(\text{entropy}(S))$ using either *Weighted triangulation refinement* or *Weighted Trapezoidal decomposition* if S has only regions of constant complexity. With a simple example we show that for a 2-dimensional subdivision S which is not restricted to regions with constant complexity it is impossible to get a query time of $O(\log(\frac{1}{p_q}))$. Hence, the answer to research question 1 is negative for 2-dimensional point location. We therefore turn our attention to research question 2. We show that it is possible to have faster query times for query points far away from the region boundary using an adapted version of a so called star quadtree. Then the 1-dimensional MDBST is adapted for 2-dimensional subdivisions. The 2-dimensional version is called MDBST-2D. At the time of writing it can handle subdivisions where the edges are either vertical or horizontal. For these subdivisions we show that MDBST-2D has $O(\log(\frac{1}{\delta_q}))$ query time, where δ_q is the distance from the query point to the nearest region boundary. Finally we experimentally compare the star quadtree and MDBST-2D using rectilinear subdivisions. In this experiment the expected query time of the MDBST-2D is always less than the expected query time of the star quadtree. We believe that this is not only the case in this experiment but that this is always the case, except for some degenerate cases.

2 Background on point-location structures

This chapter contains related work and serves as background information for the rest of this thesis. First the OBST is described which can be used as a 1-dimensional point-location structure and has an optimal expected query time. The OBST is analysed in Section 3. Then the 2-dimensional point-location structures *Triangulation refinement*, *Trapezoidal decomposition* and 2 kinds quadtrees are described. These 2-dimensional structures are analysed in Chapter 4.

2.1 1-dimensional point location

In 1-dimensional point location the subdivision is an interval $[a, b]$ that is partioned into regions (intervals) $R_1, R_2, \ldots R_n$ defined by boundary points $x_0, x_1, \ldots x_n$ (see Figure 1). For the sake of convenience we assume that $[a, b] = [0, 1]$. A subdivision of any other interval can easily be translated and scaled such that its domain becomes $[0, 1]$ without changing any of the ratios between the regions. One could use an ordinary balanced seach tree as point-location structure. The tree would have a depth of $O(\log n)$ and therefore can answer any point-location query in $O(\log n)$ time. If one knows for each region R_i its probability (that a query point lies in that region) then one can optimize the point-location structure for its expected query time. The expected query time of a tree T is $O(cost(T))$, where cost is

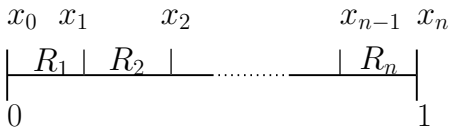


Figure 1: 1-dimensional point location.

defined in Definition 2.1, where we assume that for each region R_i there is a unique leaf node u_i where the search with a query point $p \in R_i$ ends.

Definition 2.1. *Let T be a tree and p_i be the probability of a query to end in the leaf node $u_i \in T$ corresponding to region R_i. Then* $cost(T) = \sum_{u_i \in T} ((depth(u_i) + 1)p_i)$.

The *OBST* (Optimal Binary Search Tree) balances the tree such that it has minimal cost and thus optimal expected query time. The OBST tree is constructed using dynamic programming [4]. The general form of the subproblem to be solved by the dynamic-programming algorithm is as follows. Given indices i, j with $0 \leqslant i < j \leqslant n$, compute an optimal tree for the subdivision of $[x_i, x_j]$ into regions induced by $x_i, \ldots, x_j$.

First three tables are filled with information about these subproblems and then the OBST is constructed using the last table. The first table $w[0 \ldots n-1, 1 \ldots n]$ contains the weight of the subproblems which is the sum of the probabilities of the regions inside the interval of the subproblem. The second table, $cost[0 \ldots n-1, 1 \ldots n]$, contains the cost of the subproblems that is, cost$[i, j]$ is the minimum cost of a tree on the subdivision of $[x_i, x_j]$. Then $cost(i, j) = cost(i, r) + cost(r, j) + w(i, j)$, where r is the point contained in the root node. Unfortunately it is not known which r results in the least cost and the cost for each possible r needs to be calculated.

$$\text{Thus } cost(i,j) = \begin{cases} p_j & \text{if } i = j-1 \\ \min\limits_{i<r<j} cost(i,r) + cost(r,j) + weight(i,j) & \text{if } i < j-1 \end{cases}$$

After calculating the cost in previous table, one can fill the third table. The third table, $root[1..n-1, 2..n]$, contains the index of the point in the root node of the subproblem which is. This table has one row less than the other two since the root of an OBST of a single region is a leaf which does not contain a point. The OBST is constructed using the third table. Start with the root of the complete problem which is located at the top of the table. Divide the range in two using this root and get the roots of the subtrees. Repeat this until the interval contains a single region which becomes a leaf.

The tables consists of $O(n^2)$ entries so the space needed for construction is $O(n^2)$. Filling an entry in this table takes $O(n)$ time since the cost for each possible root needs to be calculated. This results in $O(n^3)$ construction time. The book "Introduction to algorithms" [4] contains a more elaborate description of the OBST, with i.a. pseudo-code.

2.2 Triangulation refinement

The first standard 2-dimensional point-location structure that we discuss [7] is based on triangles. Therefore the non-triangular regions in the subdivision S need to be triangulated

first. Also S may contain at most three outer vertices, which form a triangular bounding box around the subdivision. In case it contains more than three outer vertices then at most two extra vertices need to be added to meet this requirement. The general idea is to build a hierarchy of triangles. The structure consists of a series of triangulations T_0, T_1, ...,T_m. Triangulation T_0 consists of only one triangle made of the three outer vertices, then in each following triangulation the previous one gets refined, until the final triangulation T_m which is the triangulated version of the original subdivision. Each triangle Δ in T_k contains links to those triangles in T_{k+1} that intersect it. We call these triangles the children of Δ. A *terminal triangle* is a triangle that is (also) found in the final triangulation T_m. Quering this point-location structure with a query point q is done by going down the hierarchy, starting in the single triangle of T_0 and always proceeding to the child containing q, until a terminal triangle is reached.

The data structure is constructed bottom up. Triangulation T_k is derived from T_{k+1} by removing an independent set of vertices (not containing the three exterior vertices) and re-triangulating the resultant non-triangle polygons. The links to intersecting triangles need to be added and the terminal triangles need to be marked (the ones that have just one link which is to another terminal triangle). A greedy algorithm can pick a big enough independent set each time such that the number of triangulations $m = O(\log n)$ and at the same time ensures that triangles intersect not more than a constant number of triangles from the next triangulation. This results in a query time of $O(\log n)$ and a space requirement of $O(n)$. The construction time of this point-location structure is $O(n)$ (after triangulating the non-triangular region of the initial subdivision). Figure 2 shows an example of the construction

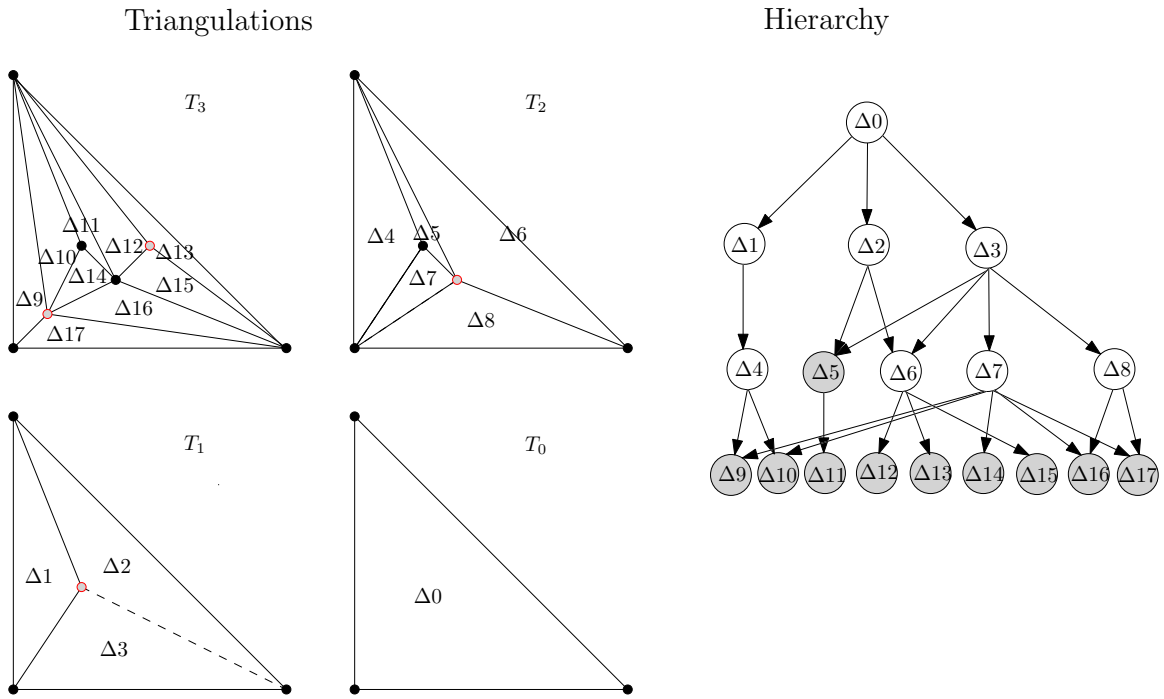


Figure 2: Triangulation refinement example.

of a triangulation hierarchy as described above. T_3 is the original triangulation. The hollow

vertices with a red edge form the independent set that will be removed in T_{k-1}. The dashed edges are created after re-triangulating a non-triangular polygon. An example of a terminal triangle that is not in the final triangulation T_3 is Δ_5. Note that Δ_5 becomes Δ_{11} in T_3 but has the same shape and position.

2.2.1 Weighted Triangulation refinement

Now suppose each region R_i of our initial subdivision has a probability p_i associated to it. Iacono [6] presented an algorithm to adapt the triangulation-refinement point-location structure [7] such that it becomes entropy-sensitive. The trick is choosing the independent sets of vertices that are removed each step in a certain way.

In the paper [7] it is assumed that the subdivision is a triangulation. But in practice the subdivision is often non-triangular. Therefore the regions need to be triangulated first. You can then distribute the probability of the region evenly over the triangles. This step can disturb the $O(entropy)$ expected query time for subdivisions with non-constant complexity regions since the triangles may get a much lower probability than the region it is in.

Let T be a triangular subdivision and n the number of vertices. Then the algorithm chooses an independent set of size $(n-3)/50$ each time, where each vertex has a degree of at most 24 and each vertex is not incident to a triangle with a probability of more than $24/n$. The subtraction of 3 out of n is caused by the fact that the outer 3 vertices may not be removed. Using Eulers formula Iacono proves that there is always such a set. Since the removed vertices have a degree of at most 24 the new triangles intersect at most 24 triangles of triangulation T_{k+1}. The size of the independent set ensures that the number of triangulations, m, is $O(\log n)$. A greedy algorithm can find the independent set of vertices in $O(n)$ time. Hence this structure can still be built in $O(n)$ time and space. The query time of a single query is O(min ($\log(n)$, $\log(1/p_i)$), where R_i is the region containing the query point q. When you multiply for each region the $O(\log(1/p_i))$ bound by its probability you get $entropy(S) = O(\sum_{R_i \in S} p_i \log(1/p_i))$ as a bound for the expected query time for subdivisions of regions with constant complexity.

2.3 Trapezoidal decomposition

The second standard point location structure we discuss [8, 10] is based on a so-called trapezoidal decomposition. (A more elaborate description is found in book [2]). To obtain a trapezoidal decomposition some extra edges need to be added to the subdivision. To this end a bounding box is created around the subdivision if it doesn't already exist. Then for each vertex a vertical edge is added, going both upward and downwards until it encounters another edge; see the lower left of Figure 3. Notice that the subdivision now consists of only trapezoids where the vertical edges are parallel. It is possible some triangles are created which can be seen as trapezoids with one edge of length zero. For every vertex only two extra edges are added so the number of edges stays asymptotically the same.

The constructed point-location structure is an acyclic directed graph with one root and out-degree two of the internal nodes. Each leaf represents one of the trapezoids in the subdivision. The point-location structure is used to find for a query point the trapezoid it lies in. Each

node contains an edge of the trapezoidal map which is used to navigate the structure.

The point-location structure is constructed simultaneously with the trapezoidal decomposition, by a randomized incremental algorithm. Each time a random edge of the subdivision is inserted into the point-location structure. First the vertical edges through its endpoints are added, followed by the edge itself. The search graph of the previous step is used to locate the trapezoids the endpoints of the new edge fall in. And the trapezoidal decomposition is used to find out which of the other trapezoids the edge intersects. Then the trapezoidal decomposition is updated. Constructing them simultaneously is done to reduce the construction time. Figure 3 shows an example of the trapezoidal map and point-location structure after inserting edge s_1 followed by edge s_2. The construction time is expected to be $O(n \log n)$, the size is expected to be $O(n)$ and the query time is expected to be $O(\log n)$. In the worst case the construction time and space used could be $O(n^2)$ and the query time linear. In case a bad structure is constructed with bad query times the algorithm can be repeated until a good one is found.

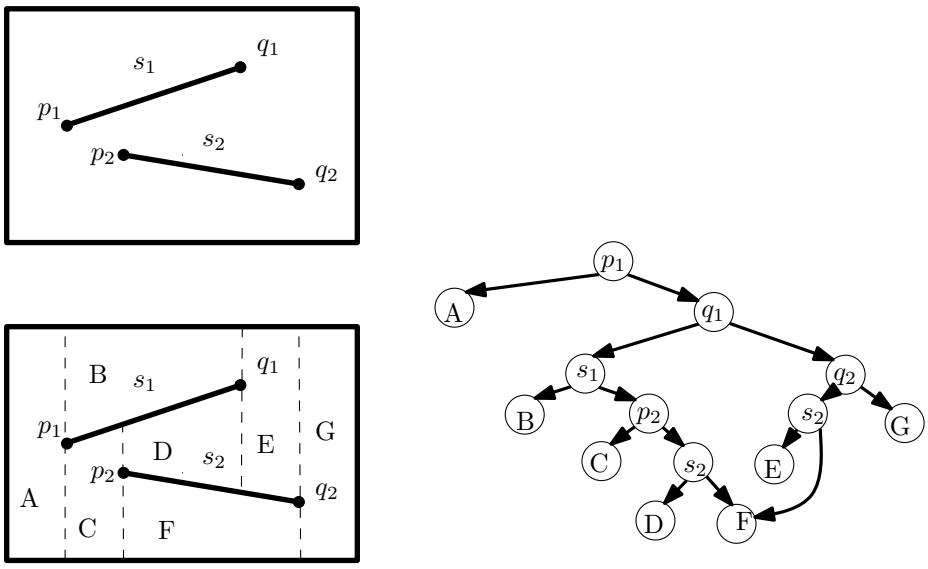


Figure 3: Trapezoidal example from [2].

2.3.1 Weighted Trapezoidal decomposition

There is also a weighted variant of *trapezoidal decomposition* [1]. The probability of each region in the subdivision is distributed evenly over the number of trapezoids it includes (not proportionately to their size). Then each of the four edges of a trapezoid gets $\frac{1}{4}$ of the probability of the trapezoid. The probability each edge gets from their adjacent trapezoids is being added up. Then a constant $K > 0$ is introduced and each edge gets a weight of $max(\lceil K \cdot p_e \cdot n \rceil, 1)$, were p_e is the probability of the edge and n the number of edges in the subdivision. Assigning the weight this way will prevent the difference between weights being more then a factor $O(n)$. The choice of K is a trade off between space requirements and query time but will not effect them asymptotically. A bigger K will decrease the expected query time while increasing the expected required space at the same time. The edges are still

randomly chosen for insertion but the probability that an edge is chosen is not equal for all edges any more, but now depends on its weight.

The expected construction time of $O(n \log n)$ and expected required space of $O(n)$ remain the same, but the expected query time is now $O(entropy)$ for subdivisions with constant-complexity regions. Like with Triangulation refinement the distribution of the probability of regions over the trapezoids prevents the $O(entropy)$ expected query time for subdivisions with non-constant-complexity regions. Though *Weighted Triangulation refinement* and *Weighted trapezoidal decomposition* both have an expected query time of $O(entropy)$, the constants used in the *Weighted trapezoidal decomposition* variant are smaller and hence a bit better [1].

2.4 Quadtrees

Now suppose that the subdivision S is a subdivision of a square. Then we can also use a *quadtree* for point location. A quadtree recursively divides the square into four equal parts until some stopping rule applies. What the stopping rule is depends on the kind of quadtree. Creating four equal parts is accomplished by splitting each square both horizontally and vertically through the middle, resulting into four equal non-overlapping new squares. These squares have sides with a length of $\frac{1}{2^t}$ the length of the sides of the bounding box, where t is the number of recursions.

When the stopping rule applies it is still possible that there is more than one region intersecting the square which means the query cannot be answered yet. In this case another point-location structure is added to these squares. Which structure depends, like the stopping rule, on the kind of quadtree. The stopping rule causes the interior of the squares to have certain properties which can be exploited. Originally quadtrees were used as an indexing structure of points in the plane and the stopping rule could be something like: stop when a square contains at most a constant number of points. Several kinds of point-location quadtrees exist. In the next subsections a short description for two of them is added.

2.4.1 Star quadtrees

A star quadtree [3] is a linear quadtree [5] designed for fat triangulations. A fat triangulation is a triangulation where the triangles have angles of at least α for some fixed constant $\alpha > 0$; the greater the minimum angle the fatter the triangulation. This α is used in the bounds given for the query time etcetera.

The star quadtree has as stopping rule: stop dividing when all edges intersecting the square are incident on a common vertex v. The vertex v can be either inside or outside the square, see Figure 4. When the stopping rule applies to a square it can intersect with at most $\frac{2\pi}{\alpha}$ regions. The point-location structure for squares with multiple possible regions remaining is unspecified. The star quadtree has $O(\log(\frac{n}{\alpha}))$ query time and requires $O(\frac{n}{\alpha^2} \log \frac{n}{\alpha^2})$ construction time and $O(\frac{n}{\alpha^2})$ space, where n is the number of edges in the subdivision.

2.4.2 Guard quadtrees

The guard quadtree [3] is meant for storing low-density subdivisions. When a subdivision S has density λ this means that any disk D is intersected by at most λ edges $o \in S$ where

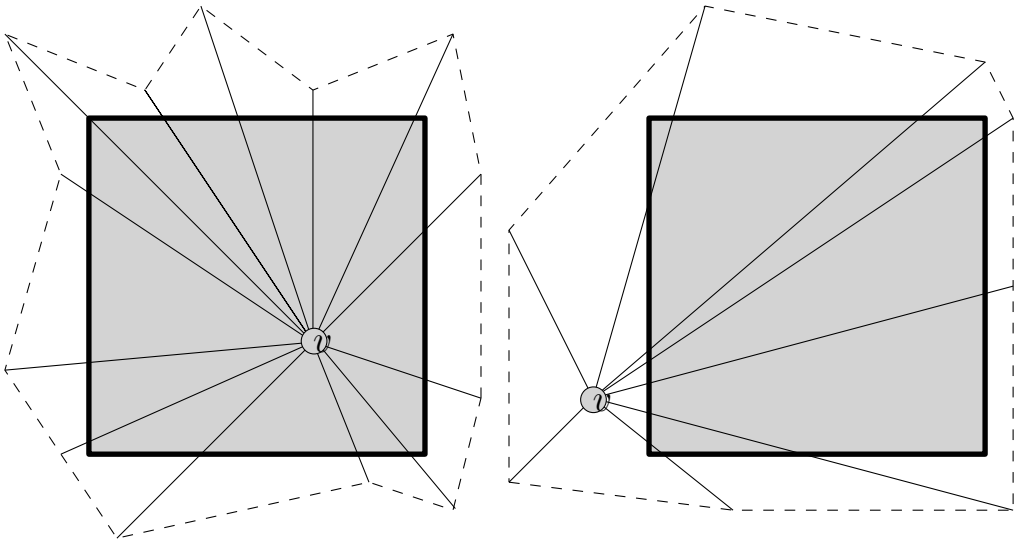


Figure 4: Two squares for which the star quadtree stopping-rule applies.

$length(o) \geqslant diameter(D)$. Every edge has four so-called *guards* which are used in the stopping rule. The guards are located at the corners of the bounding box of the edge. Note that two of those guards are located at the endpoints of the edge.

The stopping rule of the guard-quadtree is: stop dividing when the number of relevant guards inside the square is at most λ. Relevant guards are guards whose corresponding edge intersects the square. The maximum number of remaining edges and thus possible regions in such a square is $O(\lambda)$. The point-location structure for squares with multiple possible regions remaining is unspecified. The guard quadtree is stored as a compressed quadtree [9] were paths of nodes with only one non-empty child are compressed to single edges. The guard quadtree requires $O(n)$ space, $O(n \log n)$ construction time and has a query time of $O(\log(\lambda) + \log(n))$, where n is the number of edges in the subdivision.

3 1-Dimensional difficulty-sensitive point location

Before looking at 2-dimensional point location we take a look at 1-dimensional point location. Recall that our goal is to have faster query times when the query point is far away from the region boundary or in a large region. As explained earlier, this relates to entropy-sensitive point location. First we will prove that an OBST has $O(entropy(S))$ expected query time. Hence, for an OBST the research questions get answered positively. (Recall that for the analysis of the structures we make the probability that a query point is inside a region equal to the size of the region which is in 1-dimensional subdivisions the length.) However, the construction time of an OBST is very large, namely $O(n^3)$. Then we introduce a 1-dimensional point-location structure we call MDBST. It is shown that an MDBST also has $O(entropy(S))$ query time. Finally the OBST and MDBST are experimentally compared, where we see that the MDBST is much faster to construct and has an almost optimal query time.

3.1 Entropy-based analysis of the query time of an OBST

In this section we show that the OBST of a 1-dimensional subdivision S has a $O(entropy(S))$ expected query time. To do this we first prove two lemmas.

Lemma 3.1. *Let T be a BST with a subtree $T' \subseteq T$. When the depth of the whole subtree T' is increased by one, then the cost of T increases by the sum of the probabilities of the regions in T'.*

Proof. Recall that the cost of a tree (Definition 2.1) is calculated by summing up for every node its depth + 1 times the probability of the query ending up there. For the part outside the subtree T', which is $T \setminus T'$, nothing changes, since there both the depth and probability of the nodes will stay unchanged. The cost for a node $u \in T'$ changes though. Its probability stays the same but its depth is increased by one. The difference in cost that a node u contributes is $(depth_u + 2)p_u - (depth_u + 1)p_u = p_u$, where p_u is the probability that a query ends in node u. Since queries end only in leaf nodes which have a probability equal to the probability of the region it contains and a region can occur only once in T, we know that the cost of T increases by the sum of the probabilities of the regions in T'. □

Note that by reverse reasoning the opposite is also true; decreasing the depth of T' decreases the cost of T by the sum of the probability of the regions in T'.

The following lemma implies that the OBST has faster query times if the query point is inside a large region, if we take the probability of a region equal to its size.

Lemma 3.2. *Let T be an OBST and let $u \in T$ be the leaf corresponding to region R_i. Then the depth of u is at most $O(\log(\frac{1}{p_i}))$, where p_i is the probability of region R_i.*

Proof. Lemma 3.2 will be proven by showing that in T always after a constant number of splits, the maximum length of an interval and thus probability of its corresponding subtree is divided by a constant factor greater then one. The number of splits is three and the constant dividing factor is $\frac{3}{2}$. This means that the maximum probability of a node after each three steps is the multiplied with a factor $\frac{2}{3}$.

We prove it by contradiction. Assume $T' \subseteq T$ is a subtree which after three splits contains a subtree $T'' \subseteq T'$ that has a probability $p_{T''} > \frac{2}{3}p_{T'}$. If T'' is in the left branch of the root of T' then Figure 5 shows for each possible position of T'' an example of a tree with the same regions but with less cost and thus a better expected query time. The left most tree is T'. The other trees are the examples of trees with less cost then T', were T'' is one of its grey subtrees. Since T'' has a probability greater then $\frac{2}{3}$ the rest of the subtrees have a combined probability of less then $\frac{1}{3}$. And in all examples the depth of T'' decreases by one while the depths of all other subtrees increases by at most two. In total the cost of the given trees decrease compared to T'. For examples with T'' in the right simply mirror the trees in Figure 5. This shows us that T' is not optimal qua expected query time and therefore can not be an OBST after all. Since every subtree of an OBST should also be an OBST we know that T cannot be an OBST either. This contradiction proves Lemma 3.2. □

If the query point is far away from a region boundary then it has to be in a large region too. Therefore the OBST gets also faster query times if the query point is far away from a region boundary. With Lemma 3.2 we can now prove the following theorem.

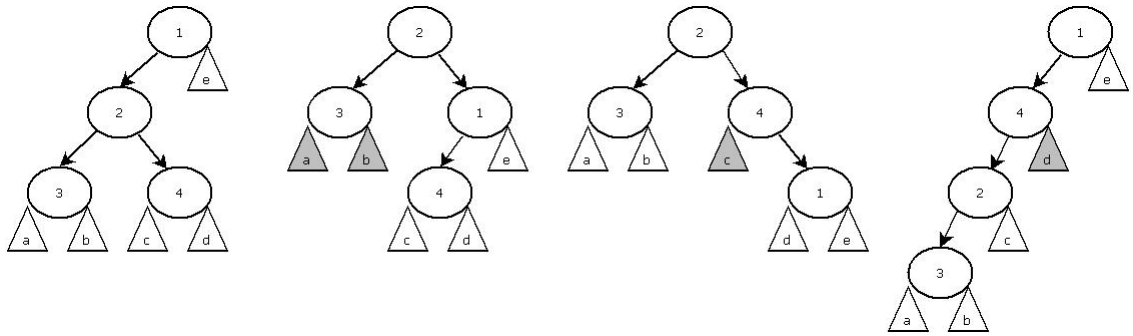


Figure 5: The left tree is T'. The other trees are rotated versions of T', containing the same nodes and subtrees. The rotated trees have less cost than T' if one of its grey subtrees (T'') has a probability of at least $\frac{2}{3}$ of the probability of T'.

Theorem 3.3. *Let T be an OBST for a 1-dimensional subdivision S. Then the expected query time of T is $O(entropy(S))$.*

Proof. We know that the expected query time is measured by the cost of the OBST and only leaf nodes with regions contribute to the cost and these regions occur exactly once. Therefore the expected query time is the sum of the probability of the regions times the depth of the node in which they are located. Let T be the OBST of subdivision S and p_i the probability of a region then the expected cost is $\sum_{R_i \in S} (p_i \cdot \log(1/p_i)) = O(entropy)$. □

The OBST does not have $O(\log n)$ query time like for example Weighted triangulation, where n is the number of regions. We show this by giving a counter example with $O(n)$ query time. Suppose S is a subdivision with an interval $[0, 1]$ that has n regions. If S would have a boundary point halfway its interval, and then each time recursively another boundary point halfway the interval from 0 to the previous boundary point until it has n regions. Then the OBST of S would give you the boundary points $x_0..x_n$, where $x_0 = 0$, $x_n = 1$ and other points are $x_i = \frac{1}{2^{n-i}}$. The OBST of S would then be a left-going chain with length n. See Figure 6 for an example with $n = 5$. The smallest regions would have a $O(n)$ query time. More then $O(n)$ query time for a OBST is not possible since it uses only boundary points for its internal nodes and exactly once.

3.2 Another difficulty-sensitive 1-dimensional point-location structure: MDBST

The construction of an OBST takes $O(n^3)$ time, where n is the number of edges in the subdivision. For many applications this is to slow. Here we introduce another point-location structure called MDBST (Middle Dividing Binary Search Tree) that can be constructed in $O(n \log(n))$ time, though it is not always optimal in terms of the expected query time. But when it is not optimal it is close.

This MDBST algorithm works as follows. It is a recursive algorithm that at each step has as input a subdivision $[x_i, x_j]$ which is a part of the original subdivision. More precisely, x_i and x_j are boundary points with $i < j$. Each time take the boundary point that divides $[x_i, x_j]$ most equally in terms of length. This is the point closest to the middle point. Take this point as a split point for the root of the (sub)tree starting with the whole interval. Then split

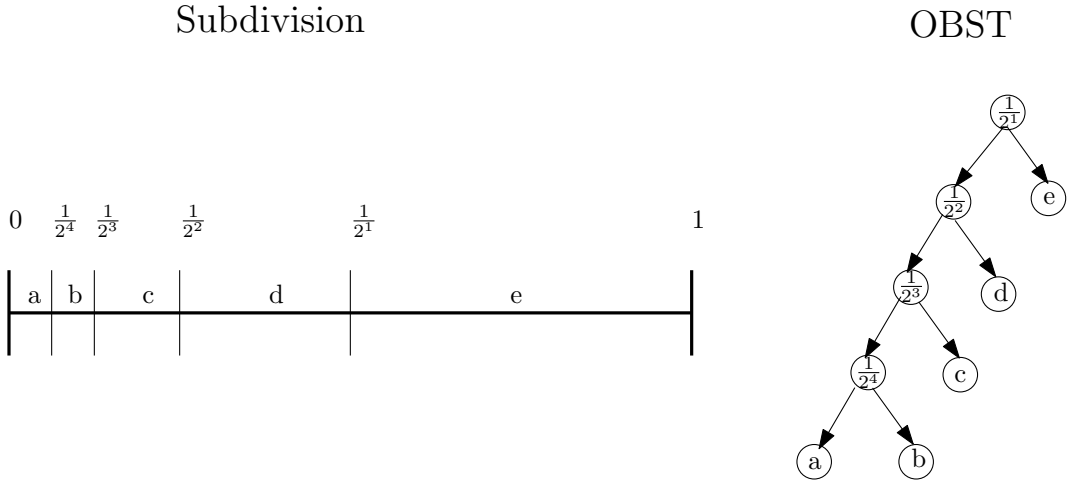


Figure 6: OBST with $O(n)$ query time were $n = 5$.

subdivision in two at that split point and recursively add the (sub)trees from the two smaller subdivisions until they consist of a single region and become a leaf.

As mentioned before this 1-dimensional point-location structure MDBST can be constructed in $O(n \log n)$ time. If there are n regions then there are $O(n)$ points between regions. Each point represents exactly one node of the MDBST. Together with the n leaf nodes containing a region there are $O(n)$ nodes. For each recursive step the point closest to the middle needs to be chosen. Calculating the middle point takes $O(1)$ time and finding the closest point can be done in $O(\log n)$ time since you can order them beforehand in $O(n \log n)$ time. Together this results in $O(n \log n)$ construction time.

In Algorithm 8 pseudocode for constructing an MDBST is given.

Algorithm *MDBST*$(X[0..n], i, j)$
$(*$ $X[0..n]$ is an array with boundary points. $*)$
$(*$ $X[i] = x_i$ $*)$
$(*$ Indices i, j with $0 \leqslant i \leqslant j \leqslant n$ define part of the subdivision to be handled which has interval $[x_i, x_j]$. $*)$

1. mp $\leftarrow \frac{X[j]-X[i]}{2} + X[i]$
2. Let c be the index of the boundary point closest to mp.
3. $root(x) \leftarrow X[c]$
4. **if** $c - 1 \geqslant i$
5. $\quad left(x) \leftarrow$MDBST(X, i, c)
6. **if** $c + 1 \leqslant j$
7. $\quad right(x) \leftarrow$MDBST$(X, c, j)$
8. **return** x

Figure 7 shows an example of a 1-dimensional subdivision and the constructed MDBST. Note that the choice for x_1 could have also been for x_2 (since both were as far away from the middle point resulting in a slightly different tree depending on the implementation). Figure 8 shows another example where the result of MDBST is not an optimal in terms of the expected query

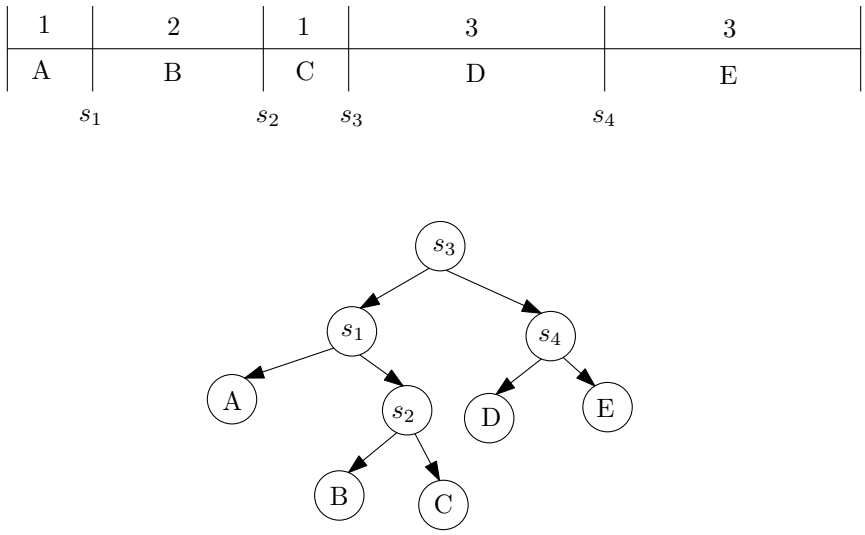


Figure 7: Example subdivision with resulting MDBST tree.

time. In the example ϵ has a value $0 < \epsilon < \frac{1}{6}$. The cost of the MDBST tree is $(2+1) \cdot 1 = 3$. The cost of an OBST tree is $(1+1)(\frac{1}{2} - \epsilon) + (2+1)(\frac{1}{2} - \epsilon) + (3+1)2\epsilon = \frac{5}{2} + 3\epsilon$. So in this case the MDBST tree is at most $\frac{6}{5}OPT$ since $\frac{3}{5/2+3\epsilon} \leqslant \frac{6}{5}$.

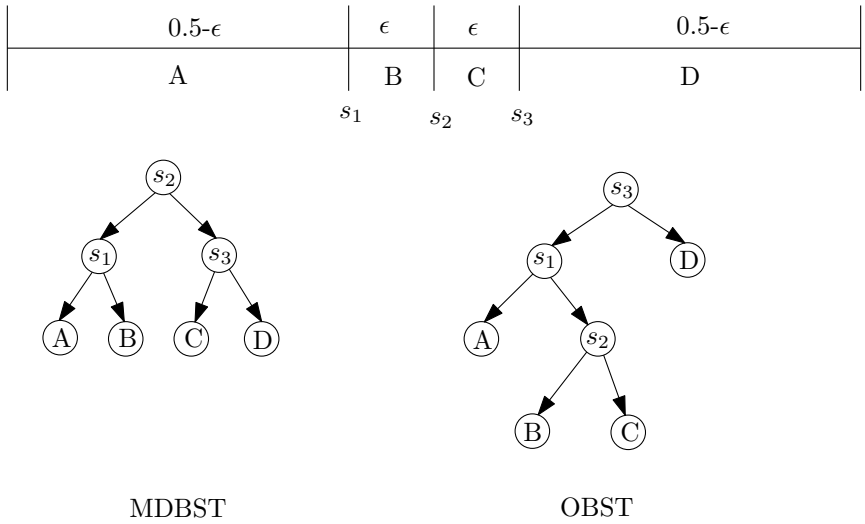


Figure 8: Example subdivision for which a MDBST does not have optimal query time.

3.3 Entropy-based analysis of the query time of an MDBST

In this section we show that the MDBST of a subdivision S has a $O(\log(\frac{1}{p_i}))$ query time, where p_i is the probability of the region where the query point lies. Hence, it has faster query times for query points inside large regions, faster query times for query points far away from a boundary, and an expected query time of $O(entropy(S))$, if the probability of a region is equal to its length.

Theorem 3.4. *Let T be an MDBST of subdivision S. Then a query with a point q takes $O(\log(\frac{1}{length(R_i)}))$ time, where R_i is the region containing q.*

Proof. Similar to the proof of Lemma 3.2 for the OBST, we will show that always in a constant number of splits the interval of a subtree is divided by at least a constant factor greater then one. The number of splits is three and the division is at least $\frac{3}{2}$. This means after each three splits the maximum length of an interval and thus probability of the tree is at most $\frac{2}{3}$ of what it was.

Let subtree $T' \subseteq T$ and x_i boundary point closest to the middlepoint. *If T contains just a single region it will not have a point p' but then the theorem is true anyway since it becomes a leaf which will not be split any further.* Since T' is an MDBST its root contains p' as split point. Then we make a case distinction as seen in Figure 9. We made the interval of T' start at 0 and the distance be relative to the length of the interval of T'. This will not affect the ratios between the regions. In case A the distance from point p' to the middlepoint of T' is greater then $\frac{1}{6}$ and else it is case B.

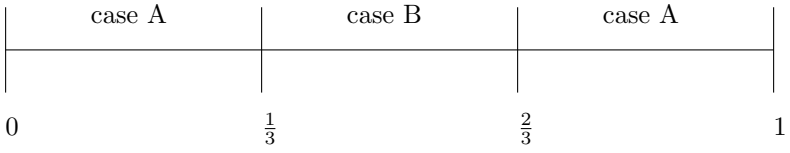


Figure 9: Cases A and B.

Case A:
In case A, point p' is located between 0 and $\frac{1}{3}$ or $\frac{2}{3}$ and 1 in the interval. Point p' will split the interval of T' into an interval bigger than $\frac{2}{3}$ and a small interval at most $\frac{1}{3}$. Let $T_1 \subset T'$ be the subtree corresponding to the small interval and $T_2 \subset T'$ be the subtree corresponding to the big interval. If T_2 contains more than one region it will be split. Let p_2 be the point between regions closest to the middlepoint of T_2 which splits it. Since T_2 contains the bigger interval of the two it will also contain the middle point of T'. Let $r_{2,1}$ be the region containing the middle point of T'. Point p' is a boundary of $r_{2,1}$ with a distance to the middle point of at least $\frac{1}{6}$. The other boundary is located at least as far away from the middle point to the other side. This means $r_{2,1}$ will include at least the interval from $\frac{1}{3}$ to $\frac{2}{3}$ and has a length greater then $\frac{1}{3}$. To either sides of $r_{2,1}$ the remaining intervals can now only be smaller then $\frac{1}{3}$. Therefore the interval of $r_{2,1}$ is greater then half of the interval of T_2, meaning that the middlepoint of T_2 is in $r_{2,1}$ and p_2 is the other boundary of $r_{2,1}$. Then p_2 will split T_2 in two. Let subtree $T_{2,1} \subset T_2$ be the one containing $r_{2,1}$ then that we the only region in it which means it becomes a leaf and will not be split any more. Let $T_{2,2} \subset T_2$ be the other subtree with an interval with the length smaller then $\frac{1}{3}$. Now T' is divided into subtrees T_1 and $T_{2,2}$ with interval smaller then $\frac{1}{3}$ after at most two splits and subtree $T_{2,1}$ which is not split any more after one split. In Figure 10 you can see the tree for case A.

Case B:
Since in case B point p' has a distance of at most $\frac{1}{6}$ to the middle point, it will split the interval of T' in intervals with lengths of at most $\frac{1}{2} + \frac{1}{6} = \frac{2}{3}$. So after one split the subtrees have intervals with lengths of at most $\frac{2}{3}$ of the interval of T'.

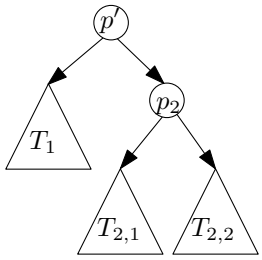


Figure 10: MDBST for case A.

Both in case A and B we see that in every tree splits the the maximum interval length will be divided by at least $\frac{2}{3}$. Then the query time for T is at most $\log_{3/2}(\frac{1}{length(R_i)}) = O(\log(\frac{1}{length(R_i)}))$. This proves Theorem 3.4. □

The MDBST can have in the worst case $O(n)$ query time. The same reasoning as for the OBST applies here, which is given at the end of Section 3.1.

3.4 Experimental comparison of the MDBST and the OBST

Two experiments were conducted to compare the two point-location structures. Although the MDBST has not always the minimal possible cost it is expected to be constructed much faster than the OBST. In experiment 1 the scaling behaviour with respect to the number of regions is investigated. We look at the construction time, expected query time and depth of the point-location structures. In experiment 2 we look at the expected query time of the point-location structures with respect to the proportion of big regions compared to small regions.

Java is used as programming language. The computer used for conducting the experiments is a "Lenovo T60p" laptop with the following properties:

Microsoft Windows XP
Professional
Service Pack 3

Intel(R) Core (TM)2 CPU
T7200 @ 2.00GHz
2.00 GHz, 2,00 GB of RAM
Physical Address Extension

3.4.1 Experiment 1: Scaling behaviour for uniformly distributed boundary points

Data sets.
13 sets of 10,000 subdivisions each were created. A set consists of subdivisions with either 20, 40, 60, 80, 100, 120, 140, 160, 180 or 200 regions. Each region is assigned a random width [0,1). At the end the width of each region is divided by the sum of the widths of all region,

such that the interval of the subdivision becomes $[0, 1]$. Each region has the precision of a double in Java. For each subdivision an OBST and a MDBST are constructed.

Construction time.
In the second and third column of Table 1 the average construction times of the OBST and MDBST are shown in milliseconds (All tables are in the appendix). Despite the somewhat inaccurate measurements because of short construction times you see that the MDBST is as expected constructed much faster. We expect the construction times to be $O(n^3)$ for the OBST and $O(n \log n)$ for the MDBST, were n is the number of regions. In column 4 and 5 the construction times are divided by n^3 and $n \log n$ respectively. Here the numbers settle around a constant number, indicating that the bounds for the construction time are tight. The numbers of the OBST are more steady than the ones of the MDBST. Constructing an OBST always requires the same number of comparisons for subdivisions with the same number of regions. This is not true for the MDBST since the number of comparisons needed for searching the boundary point closest to the middle of an interval varies. In Figure 11 most of the data is shown again in a graph which gives a good impression of the big difference of construction time.

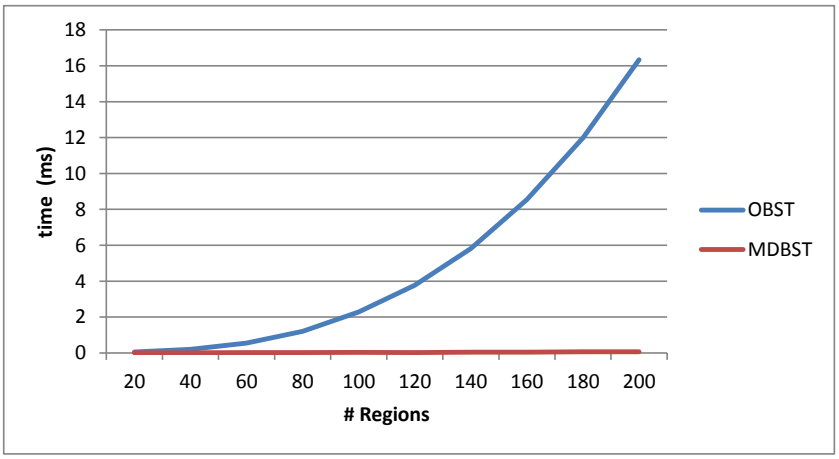


Figure 11: average construction times in milliseconds experiment 1.

Expected query time.
Recall that the expected query time is $O(cost(T))$. In Table 2 and Figure 12 both the average and maximum relative cost of the MDBST compared to the cost of the OBST is shown. It shows a declining trend in both the average and maximum. And the difference in cost is not big to start with. When the MDBST has the same cost as an OBST it means that it is optimal too. Surprisingly no optimal MDBST was created in this experiment for subdivisions with 80 or more regions.

Depth.
In Figure 13 and Table 3 the minimum, average and maximum depth of both the OBST and MDBST is shown. The two structures do not differ much in depth but usually the MDBST seems to have slightly smaller depth.

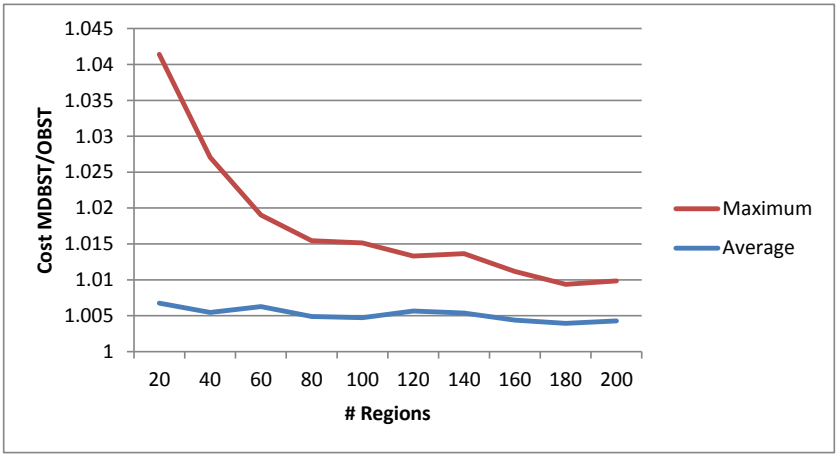


Figure 12: Cost MDBST/OBST, experiment 1.

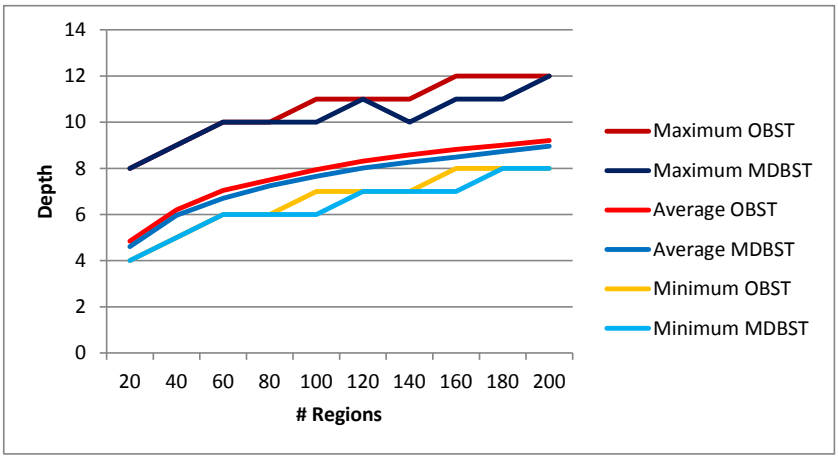


Figure 13: Depth, experiment 1.

3.4.2 Experiment 2: Large and small regions

For the second experiment subdivisions are created with small and large regions, as follows. The small regions are the same as the regions in the previous experiment which have lengths of [0,1). The large regions vary per set and can have lengths of either $[5, 10)$, $[50, 100)$, $[500, 1000)$ or $[5000, 10000)$. The portion of large regions starts at zero and is increased by $\frac{3}{100}$ each time. Each combination forms a set and each set contains 10,000 subdivisions of 200 regions. For each subdivision an OBST and an MDBST is created. The goal of this experiment is to find out if there is a relation between the relative cost of the MDBST with respect to the fraction of large regions.

Table 4 contains the average relative cost of the MDBST per set. The first column shows which portion of the regions is a large region. The first row shows how large these large regions are. In Figure 14 this information is shown in a graph.

When a subdivision has no large regions then it is similar to the subdivisions in exper-

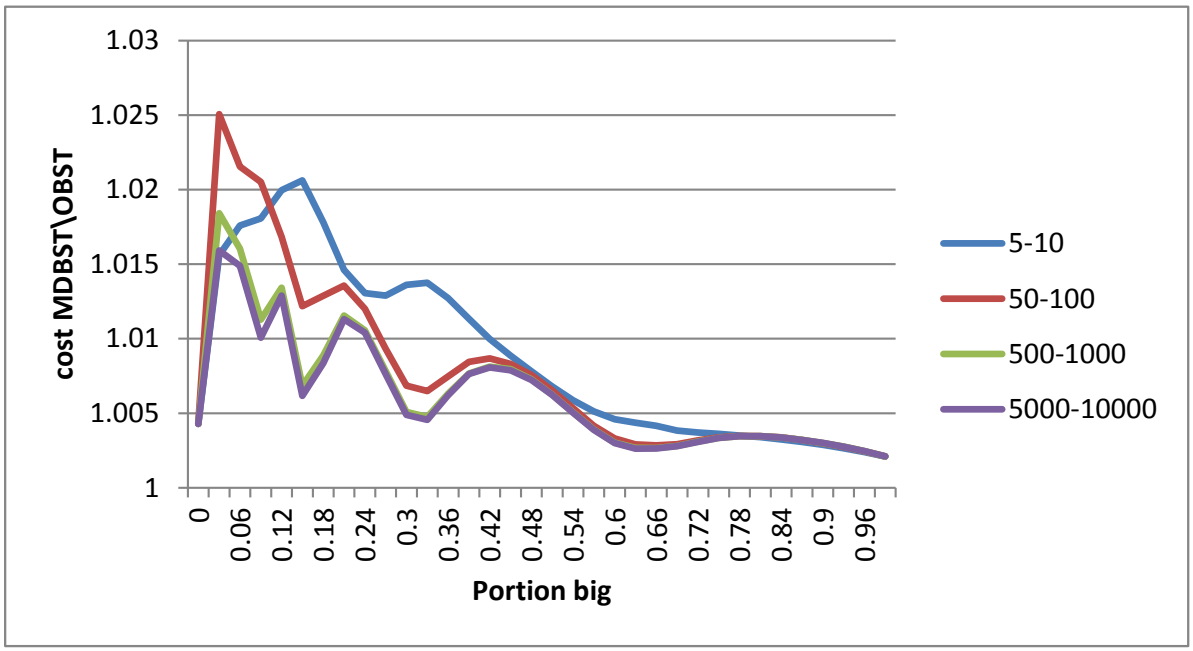


Figure 14: Average relative cost MDBST, experiment 2.

iment 1 (for 200 regions). When it contains a few large regions then the MDBST performs relatively bad although its average cost is still not more than 2% more then for the OBST. After that you'll see an overall decreasing line. After some point (around $\frac{70}{100}$ large regions) the MDBST performs even better than with no large regions. Even though the line is overall decreasing we see some fluctuation in it and this fluctuation seems to be similar for all four lines where each line represents subdivisions with different large regions sizes. Looking at the four lines we see that the difference in cost is smaller when the large regions are larger.

That the difference in cost is smaller when the large regions are larger could be explained by the fact that when the large regions become larger the relative size of the small regions compared to the whole length of the subdivision becomes smaller. Therefore the chance of having a small region containing the middlepoint becomes smaller and it is less likely to have a situation like in Section 3.1 high up in the tree resulting in a "big" difference in cost. Apparently the likelihood for such cases to happen is more important for the average difference in cost then the size of the big region (the bigger the big region the bigger the difference in cost can be).

Increasing the number of big regions also decreases the relative length of the small regions. Also the relative size of a single big region becomes less. This explains the decreasing line.

That after some point the MDBST performs even better then when there were zero big regions can be explained by the fact that the size difference between the smallest and largest big region is at most a factor two and for smaller regions it can be much more.

When investigating the fluctuation in the decreasing line of the relative cost we noticed that there is also a fluctuation in the depth of the OBST but not in the MDBST. The OBST has

an average depth which is greater then the MDBST but when the total number of big regions is close to a power of two then depth of the OBST is close to the depth of MDBST. And when at a point where the difference in depth is small we see minima in the decreasing line of the relative cost. In Figure 15 you can see the depths of the OBST and MDBST for the case of big regions having a length [5000,10000).

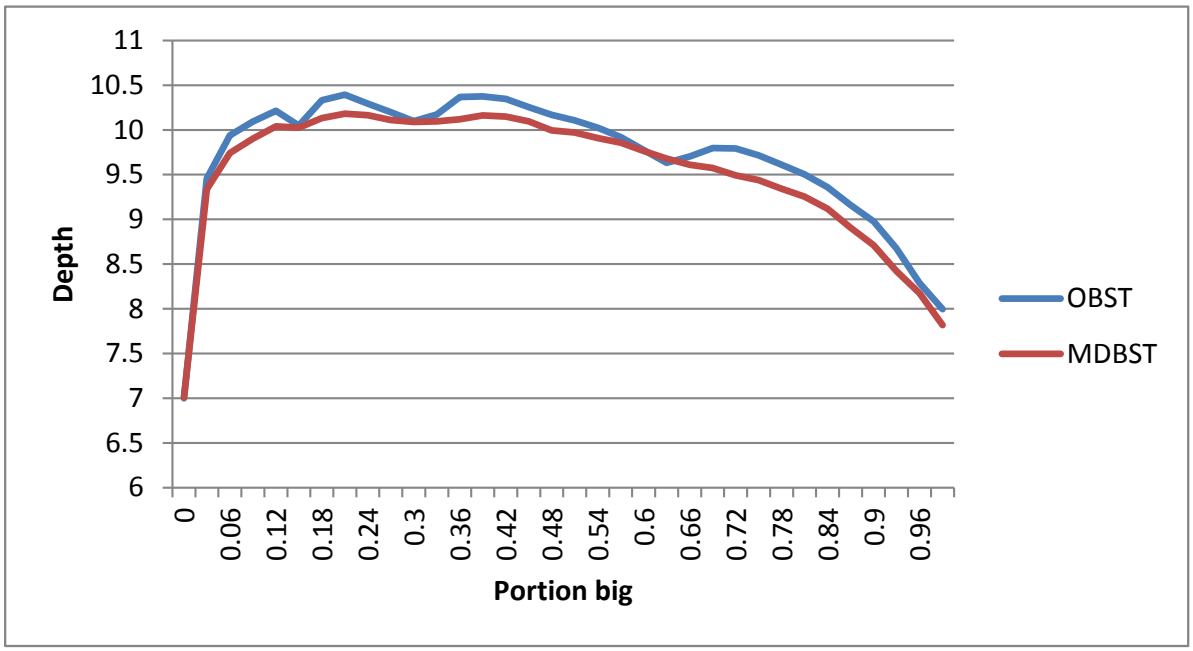


Figure 15: Average depth big regions [5000,10000), experiment 2

4 2-Dimensional point location

Now that we know more of 1-dimensional point location in relation with the two research questions, we try to answer these questions for 2-dimensional point location. In 2-dimensional point location the subdivision S is a partitioning of a 2-dimensional domain into regions (faces) $R(S)$ defined by a set of edges $E(S)$. The domain of S is the unit square. The point-location structure T for such a subdivision S is usually a tree or a DAG (directed acyclic graph) in which each node $u_i \in T$ represents a region $region(u_i)$ of the domain of S (not to be confused with the regions $R(S)$). The region of the root of T is the whole domain, and in each internal node the region is partitioned into multiple smaller regions that are represented by its child nodes. Often the region is split in two over a dividing line which is stored in the node. This line is often defined by an edge from S. It turns out that it is not always possible to create a point-location structure for a 2-dimensional subdivision that has faster query times if the query point is inside a large region. It is possible though to get faster query times if the query point is far away from the nearest region boundary. An adaptation for the star quadtree is given for which this holds. Also the MDBST for 1-dimensional subdivisions is adapted such that it can handle rectilinear 2-dimensional subdivisions (which are subdivisions that have only horizontal and vertical edges) without dangling edges. We call this version MDBST-2D. For this limited set of subdivisions we prove that it has faster query times if the query point is far away from the nearest region boundary.

4.1 Faster query times when in large region?

The first research question is:"Can we get faster query times if a query point is inside a large region?". The query time of the Weighted triangulation hierarchy and *Weighted trapezoidal decomposition* is $O(\log(\frac{1}{p_i}))$, were p_i is the probability of the region $R_i \in R(S)$ in which the query point lies. When we make the probability of a region equal to its size we get faster query times if the query point is inside a large region. However this holds only for subdivisions with regions of constant complexity. We will show with a simple example that no point-location structure can guarantee $O(\log(\frac{1}{\text{size}(R_i)}))$ query time, where R_i is the region containing the query point, when the regions in the subdivision do not all have constant complexity. Imagine a subdivision with two regions of the same size, which are separated by a zigzag line of n edges like in Figure 16. Although the subdivision has only two regions, which each occupy about half of the space each, it is obvious that a point-location structure cannot answer a query in constant time in the worst case.

4.2 Faster query times when far away from boundary?

The second research question is: "Can we get faster query times if a query point is far away from the nearest region boundary?". Let δ_q be the distance between the query point q and the nearest region boundary. If the query time of a point-location structure is $O(\log(\frac{1}{\delta_q}))$, then we would have faster query times if the query point is far away from the nearest region boundary. If a query point is far away from the nearest region boundary then the query point has to be inside a large region. Hence, *Weighted Triangulation refinement* and *Weighted trapezoidal decomposition*, for subdivisions with regions of constant complexity, have faster query times if the query point is far away from the nearest region boundary. But for subdivisions that have a region with non-constant complexity this cannot be guaranteed by these point-location

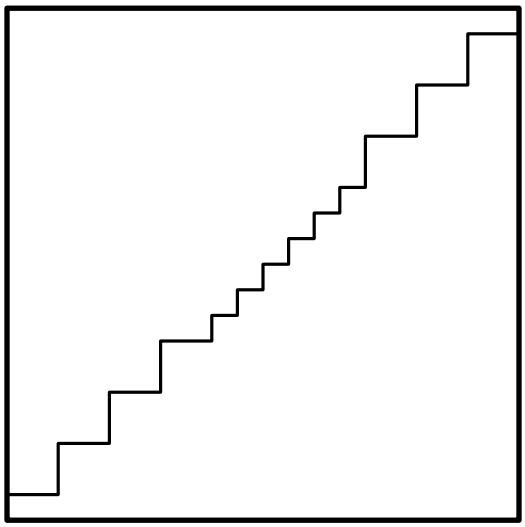


Figure 16: Two regions divided by a zigzag line.

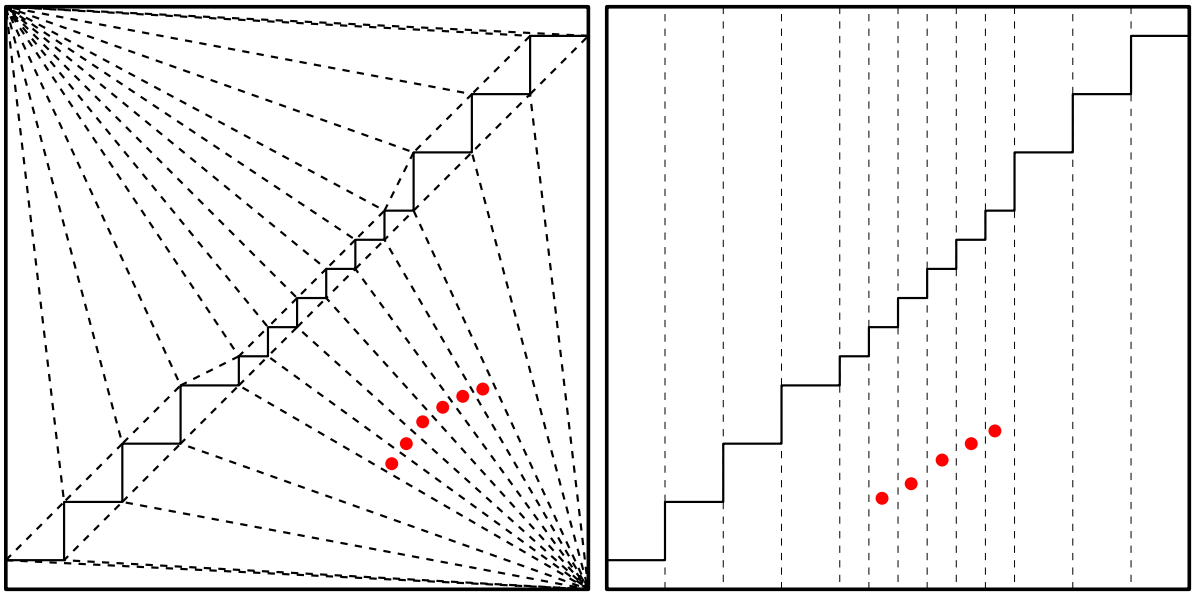
structures. The subdivision from the previous section, with a zigzag (Figure 16), is an example for which it is obvious that both *Weighted triangulation refinement* and *Weighted trapezoidal decomposition* will not produce a point-location structure which has faster query times for all query points far away from the nearest region boundary. In Figure 17 you can find a triangulation and a trapezoidal map of that subdivision, with some red points. All the red points are far way from the nearest region boundary, but is impossible to have fast query times for all of them. Quadtrees seem more suited when one wants to get faster query times if the query point is far away from the nearest border. The star quadtree and the guard quadtree are analysed next.

4.2.1 A star quadtree with $O(\log(\frac{1}{\delta_q}))$ query time

Recall that a star quadtree is designed for fat triangulations. A fat triangulation is a triangulation where the triangles have angles of at least α for some fixed constant $\alpha > 0$. Even though it is designed for triangulations, its stopping rule is valid for any kind of polygonal subdivision. So it is not necessary to triangulate the subdivision first. But the proven bounds for the query time, construction time and required space do not apply for non-triangular subdivisions. Of course we can first triangulate the subdivision. This would not increase the number of edges asymptotically, but the minimum angle α can become much smaller. Nevertheless, the bounds on the query time we will prove below also hold for non-triangular subdivisions. First we will show that the star quadtree has $O(\log(\frac{1}{\delta_q}) + \log(\frac{1}{\alpha}))$ query time, where α is the minimum angle. Then an adaptation for the star quadtree is given which achieves $O(\log(\frac{1}{\delta_q}))$ query time.

Theorem 4.1. *Let T be a star quadtree for subdivision S with angles of at least α for some fixed constant $\alpha > 0$. Then a query on T with a point q takes $O(\log(\frac{1}{\delta_q}) + \log(\frac{1}{\alpha}))$ query time, where δ_q is the distance between q and its nearest region boundary.*

Proof. First we analyse the query time up until the stopping rule is applied. Recall that the stopping rule for the star quadtree is: stop dividing when all edges intersecting the square



(a) Triangulation (b) Trapezoidal map

Figure 17: Triangulation and a trapezoidal map of a subdivision with a zigzag, each containing some red points. The solid lines are region boundaries, the dashed lines are edged added to create the triangulation or trapezoidal map.

are incident on a common vertex v. Note that the stopping rule holds for an empty square. Since we know that δ_q is the distance from q to its nearest edge, we know that the circle with radius δ_q and query point q at its center is empty inside. Any square with diameter δ_q or smaller that contains q, is always in that circle and therefore also empty, see Figure 18. Until the stopping rule is applied the square subdivision is partitioned recursively in four equal squares. The diameter of a child square is half the diameter of its parent. The square corresponding to a node at depth i has a diameter of $\frac{\sqrt{2}}{2^i}$. Hence, a node corresponding to a square with diameter δ_q, has depth $\log(\frac{\sqrt{2}}{\delta_q})$. The node representing the largest square which contains q and with a diameter equal or smaller then δ_q has a depth less then $\log(\frac{2\sqrt{2}}{\delta_q}) + 1$. The stopping rule will be applied to this empty square or one of its ancestors. Hence the query time up until the stopping rule is applied is $O(\log(\frac{1}{\delta_q}))$. The part after the stopping rule is unspecified for the star quadtree. At that point the square contains at most $\frac{2\pi}{\alpha}$ edges which are incident on a common vertex v. By storing these edges in a balanced search tree the query can then be answered in $O(\log(\frac{1}{\alpha}))$ time. The total query time is then $O(\log(\frac{1}{\delta_q}) + \log(\frac{1}{\alpha}))$. □

Now an adaptation is given for the star quadtree that has $O(\log(\frac{1}{\delta_q}))$ query time. We call this adapted version *δ_q-star quadtree*. The δ_q-star quadtree is a quadtree with the same stopping rule as the star quadtree. The difference is in the part after the stopping rule applies. At that moment the remaining part of the subdivision is a square with edges that are all incident on a common vertex. Hence all regions are also incident to the common vertex. If the square contains zero edges it will contain a single region, and a leaf can be added. Otherwise the produced point-location structure is a binary search tree containing edges of the subdivision

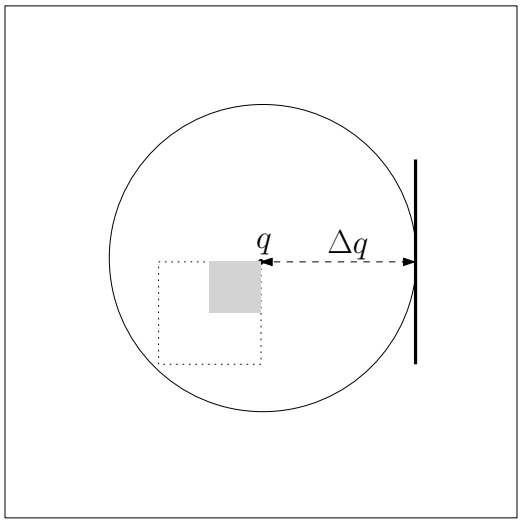


Figure 18: The gray square represents the minimum size of a square containing q.

in its internal nodes. However, we will not use a regular balanced binary search tree to store these edges. The idea is to convert the 2-dimensional subdivision to a 1-dimensional subdivision and construct an MDBST for it, and then replace the boundary points of the MDBST with their corresponding edges in the 2-dimensional subdivision. First it is explained how the point-location structure for after the stopping rule can be constructed for the case that the common vertex lies outside or on the boundary of the square. Afterwards it is explained for the case that the common vertex lies inside the square.

Let σ be a final square in the quadtree subdivision, and let $E(\sigma)$ be the set of edges intersecting σ. Let m be the number of edges in $E(\sigma)$. Let v be the vertex on which all edges in $E(\sigma)$ are incident. Pick an arbitrary edge $e_* \in E(\sigma)$, and define $\angle(e_*, e_i)$ to be the clockwise angle around v, between e_* and another edge e_i, with a value $-\pi \leqslant \angle(e_*, e_i) \leqslant \pi$ (see Figure 19). Let $e_1, \ldots, e_m$ be the edges in $E(\sigma)$ ordered on their angle with e_*. Since all edges are incident on v, all regions are also incident on v. Let $R_a, \ldots, R_m$ (with a either 0 or 1 depending on the case), such that a region R_i has the edges e_i and e_{i+1} on its boundary. We will define a 1-dimensional subdivision σ', based on σ and the edges incident to v, for which an MDBST will be made. The point $x'_1, \ldots, x'_m$ that define σ' will correspond to the edges e_i, and the regions $R'_a, \ldots, R'_m$ will correspond to regions R_i. For the precise definition of σ' we distinguish two cases.

Case 1: Vertex v lies outside or on the boundary of the square.
In this case the number of regions is $m + 1$ and $a = 0$. The regions in σ are $R_0..R_m$ and the regions in σ' are $R'_0..R'_m$ in σ'. An edge e_0 is added from v to the corner of σ, such that all other corners lie on the right side of e_0. And an edge e_{m+1} is added from v to the corner of σ, such that all other corners lie to the left of e_{m+1}. By adding e_0 and e_{m+1} we make sure that each region $R_0..R_m$ has two boundary edges. The interval of σ' becomes $[\angle(e_*, e_o), \angle(e_*, e_{m+1})]$. Every edge $e_i \in E(\sigma)$ corresponds to a boundary point $x'_i \in \sigma' = \angle(e_*, e_i)$. Note that now a region R_i in σ corresponds to a region R'_i in σ'. See Figure 20 for an example. To construct the point-location structure, we make an MDBST for σ' and replace each boundary point x'_i

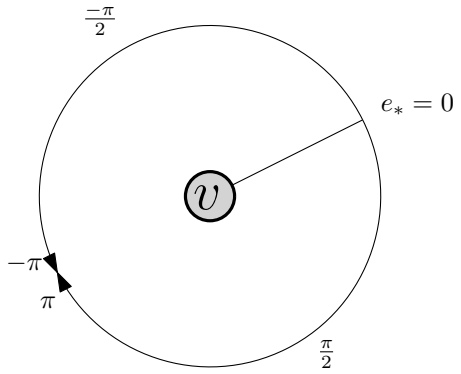


Figure 19: Illustration of the clockwise angle with e_*.

with its corresponding edge e_i.

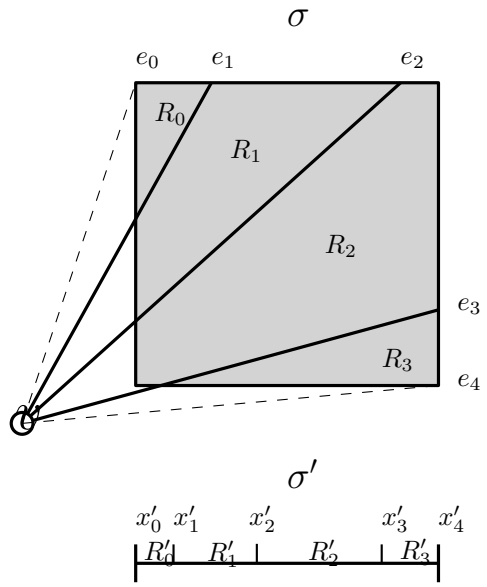


Figure 20: Example σ and σ' of a δ_q-star quadtree (Case 1).

Case 2: Vertex v lies inside the square.
In this case the edges and regions of σ lie in a complete circle around v. The regions in σ are $R_1..R_m$. No edges need to be added, since every region has two boundary edges, because e_0 now also represents e_{m+1} as boundary edge for R_m. The domain of Subdivision σ' is the interval $[-\pi, \pi]$. Similar to the previous case, every edge e_i in σ corresponds to a boundary point $x'_i \in \sigma' = \angle(e_*, e_i)$ except for $x'_0 = -\pi$ and $x'_m = \pi$.

Region R_m is split by the expansion of e_* and corresponds to two regions in σ' we call $R'_{m.1}$ and $R'_{m.2}$. If the expansion of e_* contains an edge then either the length of either $R'_{m.1}$ or $R'_{m.2}$ is zero. See Figure 21 for an example. We construct an MDBST for σ' and replace each boundary point x'_i with its corresponding edge e_i.

Now we know how the δ_q-star quadtree can be constructed, we show that it has faster query times if the query point is far away from the nearest region boundary, by proving the following

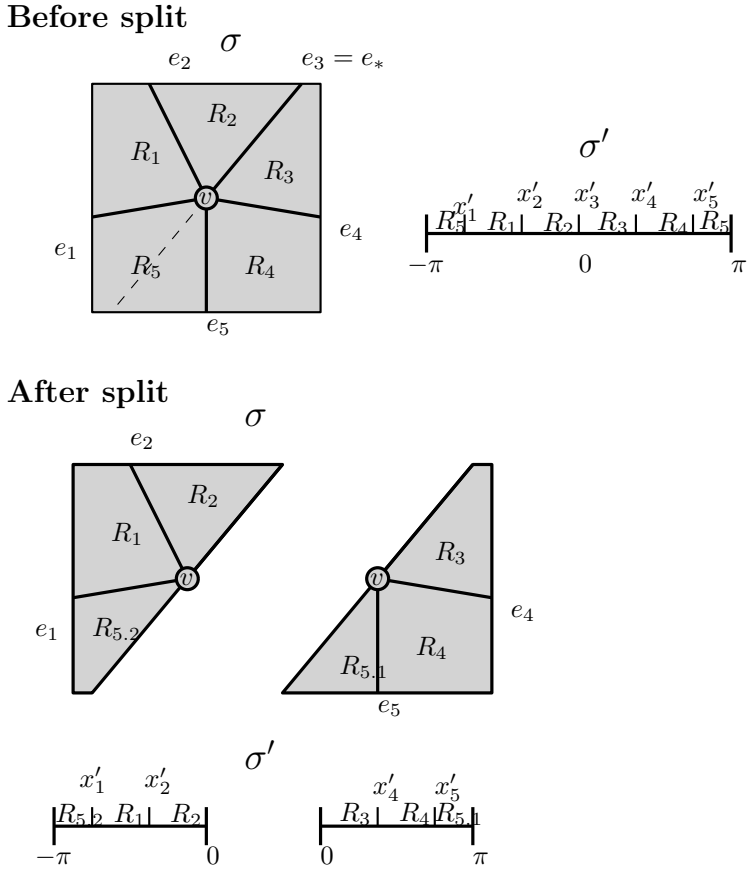


Figure 21: Example σ and σ' of a δ_q-star quadtree before and after the first split (Case 2).

theorem.

Theorem 4.2. *Let T be a δ_q-star quadtree of a 2-dimensional subdivision S. Then a query with point q takes $O(\log(\frac{1}{\delta_q}))$ time, where δ_q is the distance between q and its nearest region boundary.*

Proof. Let σ be the square of the quadtree subdivision that contains the query point q, and let σ' be the 1-dimensional subdivision corresponding to the subdivision within σ. The query time until we reach the node corresponding to σ is the same as for the star quadtree which means it is $O(\log(\frac{1}{\delta_q}))$. Since the part after the stopping rule is a MDBST we know, by Theorem 3.4, that querying it takes $O(\log(\frac{\psi}{\text{length}(R'_i)}))$ time, where R'_i is the region in σ' containing point $\angle(e_*, vq)$ and ψ is the length of the interval of σ'. This ψ is at most 2π. Hence, if we can prove that $\text{length}(R'_i) = \Omega(\delta_q)$ then we know it has $O(\log(\frac{1}{\delta_q}))$ query time.

Let δ_v be the distance between v and q. Let δ_e be the distance between q and the nearest of the two boundary edges of R_i (incident on v), and let γ be the angle between vq and the nearest of the two edges incident on v and R_i. See Figure 22. The bounding box of S is the unit square and its diameter is $\sqrt{2}$. Hence, the maximum distance between two points in σ is $\sqrt{2}$. There is at least one edge ending in v. Hence, $0 \leqslant \delta_q \leqslant \delta_e \leqslant \delta_v \leqslant \sqrt{2}$, and we have $\gamma = \arcsin(\frac{\delta_e}{\delta_v}) \geqslant \arcsin(\frac{\delta_q}{\sqrt{2}})$. It is obvious that the angle between ve and the

edge incident on v and R_i but not nearest is at least γ. Normally $\text{length}(R'_i) = x'_{i+1} - x'_i =$

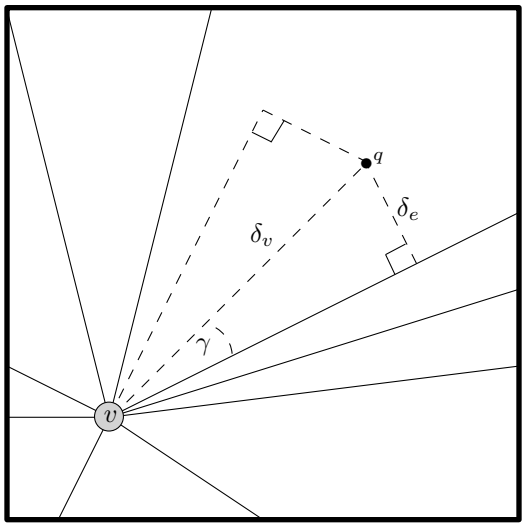


Figure 22: Example for δ_v, δ_e and γ.

$\angle(e_*, e_{i+1}) - \angle(e_*, e_i) \geqslant 2\gamma$. Only if R_i is split in two (Case 2: $i = n$), or if σ contains only one edge of R_i (Case 1: $i = 0 \bigvee i = n$), then this is not the case. In those cases $\text{length}(R_i)$ is at least γ.

Distance δ_q can never be more than $\sqrt{2}$ since it is the maximum distance in S. Since $\sinh(\frac{\delta_q}{\sqrt{2}}) \geqslant \frac{2\delta_q}{\sqrt{2}}$ for $0 \leqslant \delta_q \leqslant \sqrt{2}$, we have $\text{length}(R_i) \geqslant \gamma \geqslant \sinh(\frac{\delta_q}{\sqrt{2}}) \geqslant \frac{2\delta_q}{\sqrt{2}}$.

We already knew that the query time for after the stopping rule is $O(\log(\frac{1}{\text{length}(R_i)}))$. Hence, it is also $O(\log(\frac{1}{\delta_q}))$. Since both of the query times for the parts before and after the stopping rule are $O(\log(\frac{1}{\delta_q}))$, the total query time of the δ_q-star quadtree is also $O(\log(\frac{1}{\delta_q}))$. □

4.2.2 $O(\log(\frac{1}{\delta_q}) + \log(\lambda))$ query time guard quadtree

Recall that the guard quadtree [3] is meant for storing low-density subdivisions. When a subdivision S has density λ this means that any disk D is intersected by at most λ edges $o \in S$ where $\mathit{length}(o) \geqslant \mathit{diameter}(D)$. The following theorem gives a bound on the query time of the guard quadtree.

Theorem 4.3. *Let δ_q be the distance from query point q to the nearest region boundary and let λ be the density of the subdivision. Then the guard quadtree has $O(\log(\frac{1}{\delta_q}) + \log(\lambda))$ query time.*

Proof. Recall that the stopping rule of the guard-quadtree is: stop dividing when the number of relevant guards inside the square is at most λ. This stopping rule applies to an empty square. As shown for the star quadtree, there is a node representing such a square in the quadtree, containing point q, at depth $O(\log(\frac{1}{\delta_q}))$. The maximum number of remaining edges after the stopping rule is $O(\lambda)$. Hence, if we construct a standard point-location structure on

the subdivision within each leaf square of the quadtree, then the query time is $O(\log(\frac{1}{\delta_q}) + \log(\lambda))$. □

Unfortunately we found no way to adapt the guard quadtree such that it has $O(\frac{1}{\delta_q})$ query time, exploiting the property of containing only $O(\lambda)$ edges after the stopping rule. Other than using a point-location structure after the stopping rule, that already achieves this on its own, we see no way to achieve $O(\frac{1}{\delta_q})$ query time.

4.3 MDBST-2D

Here we adapt the 1-dimensional MDBST such that it can be used as a point-location structure for rectilinear 2-dimensional subdivisions (which are subdivisions that have only horizontal and vertical edges) without dangling edges. We call this adapted version MDBST-2D. Later on it is shown that the MDBST-2D has faster query times if the query point is far away from the nearest region boundary. Suggestions for possible ways to adapt to arbitrary polygonal subdivisions are given in Chapter 5.

In Figure 23 an example is given of a rectilinear subdivision without dangling edges. Dangling edges are not allowed since those edges are on both sides incident on the same region and therefore do not separate regions. It is possible though to get edges that on both sides are incident on the same region as seen in Figure 24.

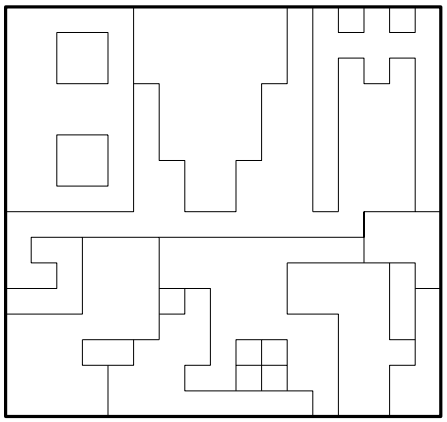


Figure 23: Example of a rectilinear subdivision.

In 1-dimensional subdivisions the MDBST chose the boundary point closest to the middlepoint of the interval, because that was the one dividing the interval most evenly. With the following example we show that in 2-dimensional subdivisions the line containing an edge closest to the middle point will not always partition the region most evenly. Note that if one only partitions the regions with horizontal and vertical lines, then the regions will always be rectangular. Suppose the subdivision is a rectangle, where the horizontal side has length 20 and the vertical side has length 10 meter long. The rectangle is divided into four regions by a horizontal line and a vertical line. The vertical line lies 1 unit to the right of the middlepoint. The horizontal line lies 1 unit below the middlepoint. See Figure 25. Partitioning the subdivision by the vertical line, will result in a part of area 110 and a part of area 90. Partitioning the subdivision by the horizontal line, will result in a part of area 120 and a part of area

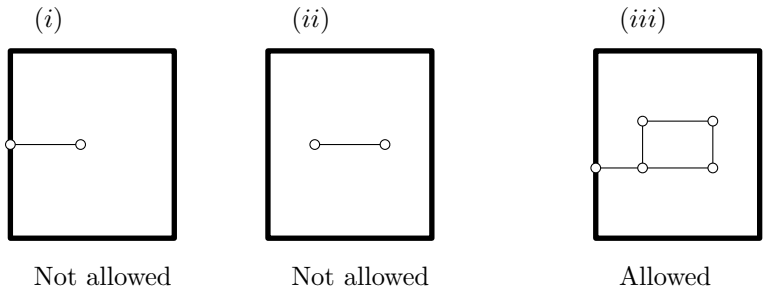


Figure 24: For all three subdivisions there is an edge incident on the same region on both sides. Subdivision (i) and Subdivision (ii) contain a dangling edge and are not allowed. However, Subdivision (iii) is allowed.

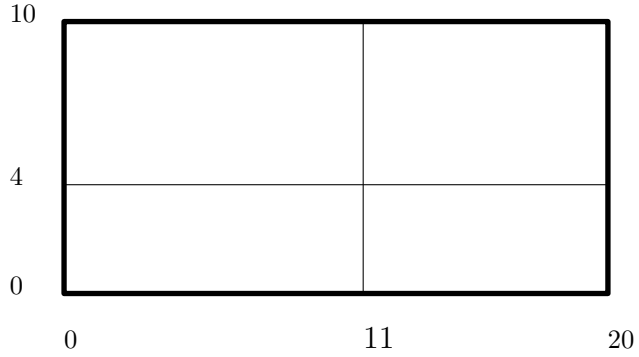


Figure 25: Altough both lines are at an equal distance to the middlepoint, the horizontal line would divide most evenly.

80. Hence, while both lines are equally close to the middlepoint, partitioning by the vertical line will divide the subdivision most evenly. The line which partitions the subdivision most evenly, is the line that has the minimum distance to the middlepoint relative to the length of the sides of the region that are perpendicular to the line. In the given example the relative distance from the vertical line to the middlepoint is $\frac{1}{20}$, and for the horizontal line it is $\frac{1}{10}$. Hence, the vertical line is the closest relative to the relevant side length, and it divides the subdivision most evenly.

Unfortunately this solution will not guarantee faster query times if the query point is far away from the nearest region boundary. While an edge can be far away from the query point, the line containing that edge can go through the interior of a region and come close to the query point. In Figure 26 a subdivision is shown for which the mentioned solution will not result in a fast query time for point q that is far away from the nearest region boundary. Hence, a different approach. One option would be to switch each time between horizontal and vertical edges, when searching for the edge closest to the middle point, until one group runs out of edges. Or one could, when possible, always choose edges perpendicular to the longest side of the region. We pick the last solution. Now we will prove some lemmas which will later on help us prove that our solution works.

Lemma 4.4. *Let T be a MDBST-2D for the rectilinear subdivision of the unit square and let $u \in T$ be a node. Then $depth_h(u) = O(\log(\frac{1}{length_v(u)}))$, where $depth_h(u)$ is the number of nodes*

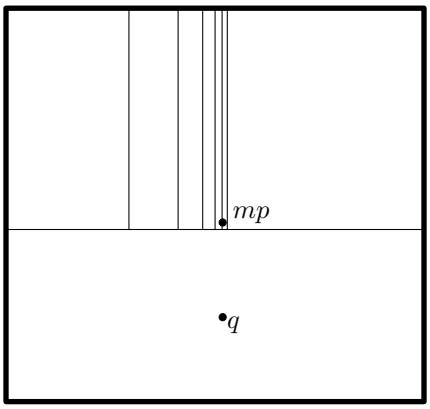


Figure 26: A subdivision for which an MDBST-2D that partitions using the line through an edge closest to the middlepoint relative to the length of the sides of the region perpendicular to the line, does not result in a fast query time for point q far away from the nearest region boundary. The point mp is the middle point of the subdivision. Note that the horizontal line lies slightly below the middlepoint. In the branch which contains q each time a vertical edge is chosen over the horizontal edge because the vertical edge goes exactly through the middle point of the respective region.

on the path from the root to node u that contain horizontal edges, and $length_v(u)$ is the length of the vertical sides of region(u). Similarly, $depth_v(u) = O(\log(\frac{1}{length_h(u)}))$, where $depth_v(u)$ is the number of nodes on the path from the root to node u that contain vertical edges, and $length_h(u)$ is the length of the horizontal sides of region(u).

Proof. We will prove the first part of the lemma. The second part can be proven in a similar way. If a node in T contains a line through a vertical edge, the length of the vertical sides of that node is equal to the the length of the vertical sides of the regions corresponding to its child nodes. Hence, lines through vertical edges will not influence the vertical sides of the region. Hence, we can essentially ignore the vertical splitting lines, and the situation is very similar to the 1-dimensional situation. Hence, according to Theorem 3.4 $\text{depth}_h(p) = O(\log(\frac{1}{\text{length}_v(p)}))$. □

Lemma 4.5. *Let T be a MDBST-2D and let $u \in T$ a node. If the regions corresponding to nodes on the path from the root of T to u always intersect at least one vertical edge if the horizontal side of the region is longest, then $depth(u) = O(\log(\frac{1}{length_h(u)}))$, where $length_h(u)$ is the length of the horizontal sides of region(u). Similarly, if the regions corresponding to nodes on the path from the root of T to u always intersect at least one horizontal edge if the vertical side of the region is longest, then $depth(u) = O(\log(\frac{1}{length_v(u)}))$, where $length_v(u)$ is the length of the horizontal sides of region(u).*

Proof. We will prove the first part of the lemma. The second part can be proven in a similar way. Let $\text{depth}_h(u)$ be the number of nodes on the path from the root to node u that contain horizontal edges, and let $\text{depth}_v(u)$ be the number of nodes on the path from the root to node u that contain vertical edges. Let $\text{length}_v(u)$ be the length of the vertical sides of

region(u).
According to Lemma 4.4, $\text{depth}_v(u) = O(\log(\frac{1}{\text{length}_h(u)}))$. Since on the path from the root of T to u there were always vertical edges intersecting the regions, a line containing a horizontal edge could only have been chosen if the vertical side of the region was longest. It takes at most $O(\log(\frac{1}{\text{length}_h(u)}))$ nodes containing horizontal lines for the length of the vertical sides to be equal or smaller than $\text{length}_h(u)$. After one more node like that the vertical side will be smaller then the horizontal side. Hence, $\text{depth}(u) = \text{depth}_v(u) + \text{depth}_h(u) = O(\log(\frac{1}{\text{length}_h(u)})) + O(\log(\frac{1}{\text{length}_h(u)})) + 1 = Os(\log(\frac{1}{\text{length}_h(u)}))$. □

Lemma 4.6. *Let T be a MDBST-2D, and let leaf(q) be a leaf containing point q, and let δ_q be the distance from q to the nearest region boundary. Let σ be the region corresponding to the parent of leaf(q). Then the region of leaf(q) has at least one side with a length of at least $\frac{\delta_q}{\sqrt{2}}$, and σ contains at least one edge perpendicular to this side.*

Proof. Let e be the edge used to partition σ, and let l be the line containing e. Let δ_e be the distance from q to e, and let δ_l be the distance from q to l. Then $\delta_e \geqslant \delta_q$ since δ_q is the minimum distance from q to an edge.

If $\delta_e = \delta_l$, then the side of leaf(q) perpendicular to l has a length of at least $\delta_e \geqslant \delta_q$, and edge e is perpendicular to this side and in σ (Figure 27, Case 1).
If $\delta_e > \delta_l$ then an endpoint, that we call ζ is the closest point on e to q (Figure 27, case

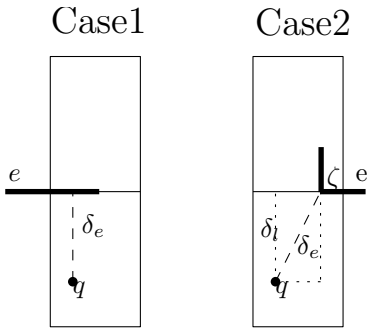


Figure 27: Illustration for the proof of Lemma 4.6.

2). This ζ is inside the region of the parent of leaf(q). The bounding box with q and ζ in its corners lies inside leaf(q), and has a diameter of δ_e. Hence, at least one side of that bounnding box and thus also a side of $leaf(q)$ has a length of at least $\frac{\delta_q}{\sqrt{2}}$. Since ζ lies inside σ and no dangling edges are allowed, there must be another edge for which ζ is an endpoint. This edge is not parallel to e else it would be closer to ζ, so it can be only perpendicular to e. This means that it does not matter which of leaf(q) has a length of at least $\frac{\delta_q}{\sqrt{2}}$, it has an edge parallel to it in σ. □

Now we can prove the following theorem that shows us that the MDBST-2D has faster query times is the query point is far away from the nearest region boundary.

Theorem 4.7. *Let T be an MDBST-2D for a subdivision S of the unit square. Then querying T with a point q takes $O(\log \frac{1}{\delta_q})$ time, where δ_q is the shortest distance from q to an edge of S.*

Proof. From Lemma 4.6 we know that the leaf containing q has a side with a length of at least $\frac{\delta_q}{\sqrt{2}}$, and the parent of leaf containing q did contain an edge perpendicular to that side. Depending on that side, being horizontal or vertical, we can prove with Lemma 4.5, that the depth of the leaf containing q is at most $O(\log(\frac{1}{\delta_q}))$. Hence, a query with point q takes $O(\log(\frac{1}{\delta_q}))$ time. □

4.4 Experimental comparison of the MDBST-2D and the δ_q-star quadtree

Some experiments were conducted to compare the δ_q-star quadtree with the MDBST-2D. We mainly look at the cost of the point-location structures, which is a good measure for the expected query time. But we also look at the number of nodes the point-location structure has, which determines the required space.

Java is used as programming language. The computer used for conducting the experiments is the same as for the 1D-experiments; see Section 3.4.

4.4.1 Data sets

The subdivisions for the experiments were generated as described next. The following parameters can be used to control the complexity and the sparseness of the subdivision:

- x (number of columns)
- y (number of rows)
- ϵ (fraction of edges to be removed, $0 \leqslant \epsilon \leqslant 1$)
- $\delta_{\min}$ (minimum length of the sides of a region)
- $\delta_{\max}$ (maximum length of the sides of a region)

To generate the subdivision, we first create a grid, with x columns and y rows. Each row and each column gets a random width in the range $[\delta_{\min}, \delta_{\max})$. Then the grid is scaled such that it fits exactly in the unit square. A grid consists of xy regions, where each region is a rectangle. Then edges are randomly removed one at a time until in total $\lceil (x(y-1) + y(x-1))\epsilon \rceil$ edges have been removed. See Figure 28 and Figure 29, for two examples of a subdivision.

We want to make sure that we always generate a subdivision of the unit square without dangling edges, that is, each vertex should have degree at least two. We do this as follows. First of all, we never remove edges on the outer boundary (the boundary of the unit square). Secondly, if after an edge is removed there are edges with a degree-1 endpoint, then those edges get removed too. If the removal of these edges results in other such edges, then these get removed too. This continues until none of such edges exist. The removed edges are added to the total number of removed edges, but this process does not necessarily stop when a fraction

of ϵ edges is already removed. Hence, it can happen that more than a fraction of ϵ edges get removed.

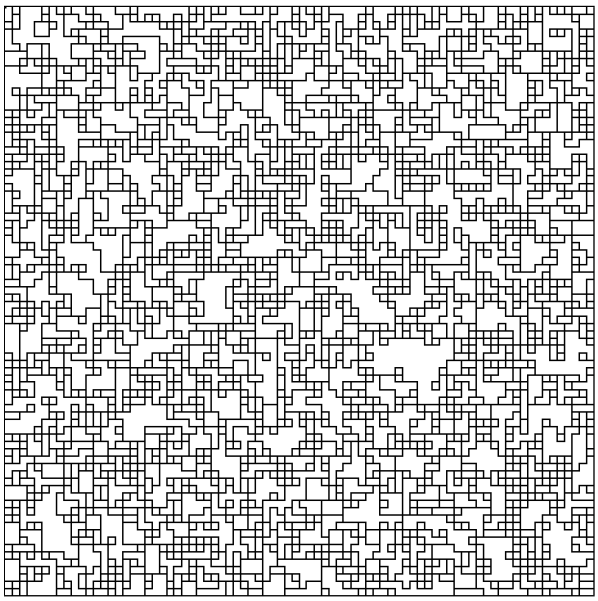


Figure 28: A subdivision with $x = y = 80$, $\delta_{\min} = \delta_{\max} = 1$, $\epsilon = 0.4$.

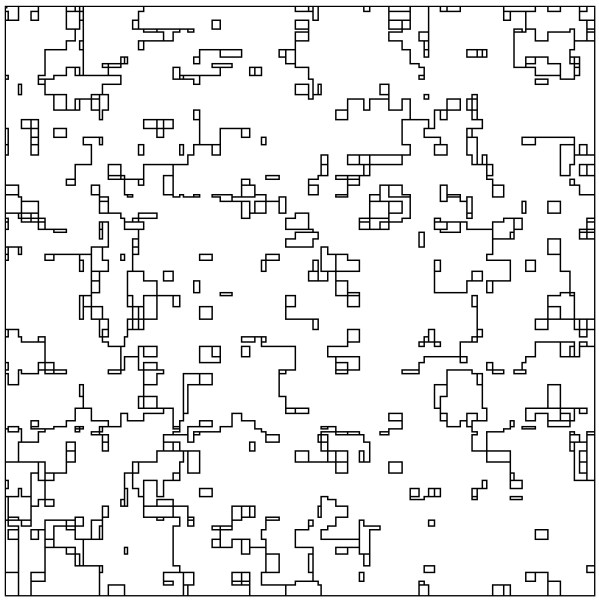


Figure 29: A subdivision with $x = y = 80$, $\delta_{\min} = 0.2$, $\delta_{\max} = 1$, $\epsilon = 0.8$.

4.4.2 Experiments

The following 3 experiments were conducted. A discussion of the outcome of the experiments is given in Section 4.4.3.

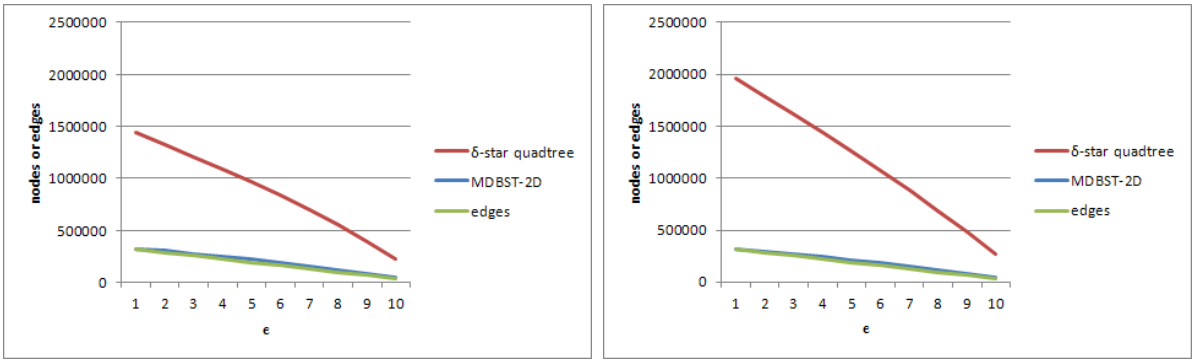
Experiment 1. Every subdivision in the first experiment is generated with the following settings:

- $x = 400$
- $y = 400$
- $\delta_{\min} = 1$
- $\delta_{\max} = 1$

Ten sets were created, with values of ϵ ranging from $\epsilon = 0$ to $\epsilon = 0.9$, with steps of 0.1. Each set consists of 1000 subdivisions. For each subdivision a δ_q-star quadtree and a MDBST-2D are created. An example for such a subdivision only with a slightly smaller x and y is given in Figure 28. Note that all rows and columns have the same width, since $\delta_{\min} = \delta_{\max}$. Hence, we simply have a regular grid when $\epsilon = 0$. For each set, the average number of nodes together with the number of edges in the subdivision (Figure 30a) and the average cost (Figure 31a), are given for both structures. Also the minimum, maximum and average relative cost of the MDBST compared to the δ_q-star quadtree is given (Figure 32a).

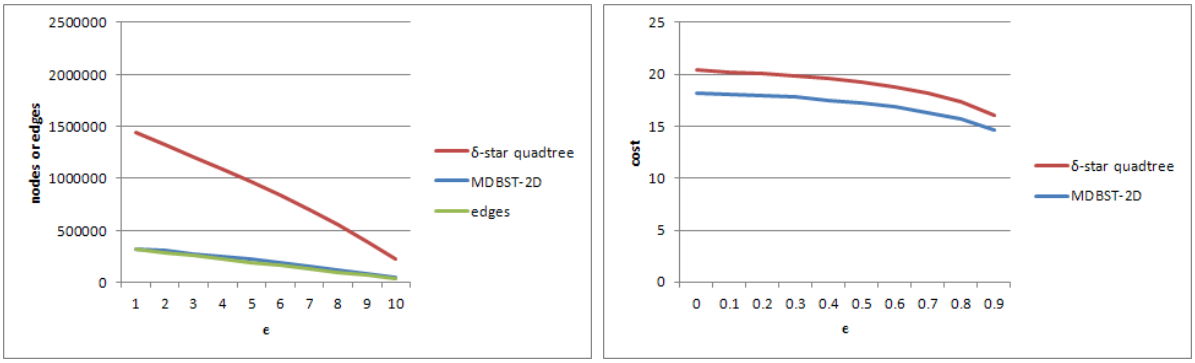
Experiment 2. The second experiment is the same as the first, except now $\delta_{\min} = 0.2$. Hence, the edges are not distributed evenly. An example for such a subdivision only with a slightly smaller x and y is given in Figure 29. The results are shown in Figure 30b, Figure 31b and Figure 32b.

Experiment 3. In the third experiment the scaling behaviour as a function of the number



(a) Experiment 1 (b) Experiment 2

Figure 30: The average number of nodes in the data structure as of a function ϵ. For comparison, the number of edges in the subdivision is shown as well.



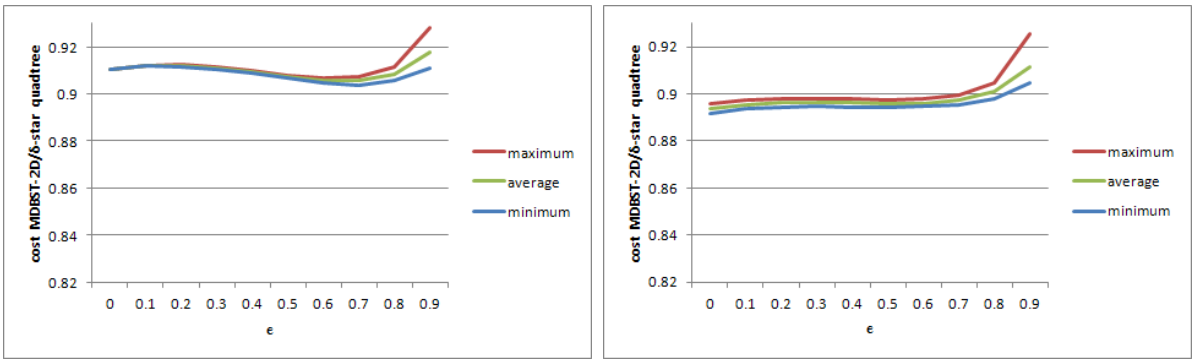
(a) Experiment 1 (b) Experiment 2

Figure 31: The average cost in the data structure as of a function ϵ.

of edges (for fixed ϵ) is investigated. The subdivisions in this experiment are generated with the following settings:

- $x = y$
- $\delta_{\min} = 0.2$
- $\delta_{\max} = 1$
- $\epsilon = 0.8$

Ten set were created with varying number of rows and columns (more rows and columns means more edges when ϵ is the same). For the first set $x = y = 50$. For each following set x and y are increased by 50. The results are shown in Figure 33, Figure 34 and Figure 35.



(a) Experiment 1

(b) Experiment 2

Figure 32: The average relative cost ($\frac{\text{MDBST-2D}}{\delta_q\text{-star quadtree}}$) in the data structure as of a function ϵ

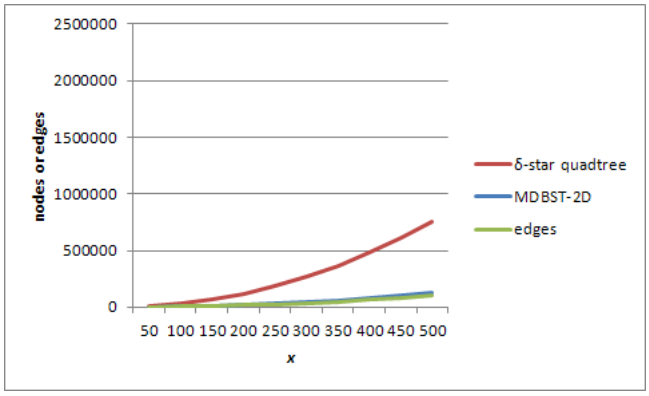


Figure 33: Experiment 3:The average number of nodes in the data structure as of a function ϵ. For comparison, the number of edges in the subdivision is shown as well.

4.4.3 Discussion of the experiments

In the experiments we see that the number of nodes of the MDBST-2D is always at most the number of nodes of the δ_q-star quadtree. This can be explained as follows. In the part of a δ_q-star quadtree before the stopping rule, one recursively partitions a square into four smaller ones, until some stopping rule applies. Partitioning a square into four smaller ones can be seen as twice, dividing the current region exactly in half over its longest side. Both sides of a square are equally long, so one can start with dividing either over the horizontal side or the vertical side, then divide the two parts over their longest side which is the side perpendicular to the last chosen side. Eventually, for every edge in the subdivision, there should exist at least one node in the tree, which contains the line over this edge. Else, one cannot distinguish between the region on one side of the edge and the region on the other side. Hence, if we do not have a split coinciding, with an edge, before the stopping rule applies, we will add it afterwards. The MDBST-2D works in a similar way. It divides the subdivision recursively, over its longest side, into non overlapping regions. But it does not necessary divide it exactly in half, because it divides over the nearest line through an edge, which is perpendicular to the longest side. This looks like an improvement, because now every dividing line coincides with at least one edge. There are no edges parallel to the dividing line between the dividing

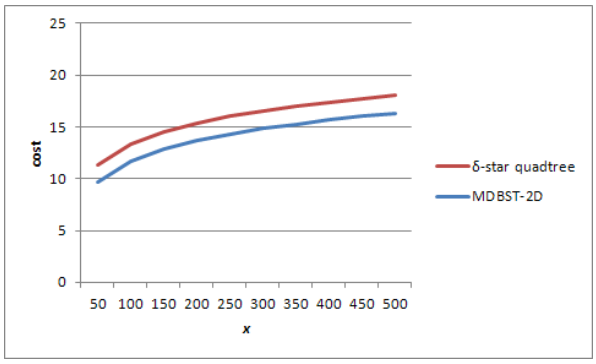


Figure 34: Experiment 3:The average cost in the data structure as of a function x.

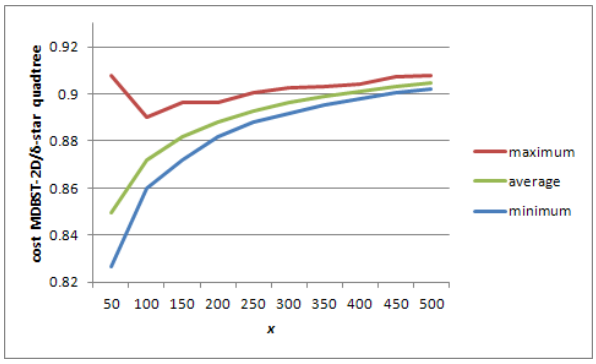


Figure 35: Experiment 3: The average relative cost ($\frac{\text{MDBST-2D}}{\delta_q\text{-star quadtree}}$) in the data structure as of a function x.

line and the line parallel through the middle point. In this part also no perpendicular edge can end since it has to end on an endpoint of another edge. Hence, shifting the dividing line to the nearest line through an parallel edge, will not result in more edges in either of the two regions. Hence, the number of nodes in the MDBST-2D is typically at most the number of nodes in the δ_q-star quadtree. The cost of the MDBST-2D is also always less than the cost of the δ_q-star quadtree. This can be explained in the same way. Hence, the expected query time and required space is less for the MDBST-2D.

Comparing the relative cost in Experiment 1 and Experiment 2 (Figure 32), we see that the relative cost of the MDBST-2D performs is smaller in Experiment 2. This is because in Experiment 1 the column and row widths are the same and the number of both the rows and columns is even. This taken together, makes the chance of an edge coinciding with the side of a square of the quadtree much higher. The advantage over the δ_q-star quadtree would be even greater, if the number of columns and the number of rows would be a power of 2, such as 256 or 512. Looking at relative cost from Experiment 3, in Figure 35, we see that if the number of edges in the subdivision is increased, that the relative difference in cost between the two structures becomes less. However we suspect that the cost of the δ_q-star quadtree will never be less than the MDBST-2D, except for some degenerate cases were the difference is made after the stopping rule of the δ_q-star quadtree. An example of a subdivision for which the δ_q-star quadtree can have less cost, depending on the implementation choices of both structures, is a square with three edges starting in the middle, of which one goes up, one goes down and one goes to the right (Figure 36).

But if we look at the relative cost in Experiment 1 and Experiment 2, we see that when ϵ is large and many edges are removed, that then the relative difference becomes less. This is contradictory to what we have seen in Experiment 3. Probably the advantage the MDBST-2D has when more multiple edges lie on the same line, which is higher when ϵ is small, is more important.

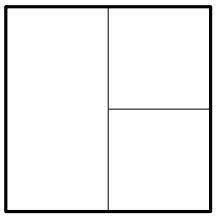


Figure 36: A subdivision for which the δ_q-star quadtree can cost less than the MDBST-2D. For both structures it costs less if it divides the subdivision first vertically, then if one would divide it first horizontally. Both structures can divide the subdivision either first horizontally or first vertically, depending on the implementation of the algorithms.

5 Conclusion

For 1-dimensional subdivisions, the OBST (Optimal Binary Search Tree) is a point-location structure that has faster query times if the query point is inside a large region or far away from the nearest region boundary. Unfortunately the OBST requires $O(n^3)$ construction time. We therefore introduced a point-location structure, called MDBST, that also has faster query times if the query point is inside a large region or far away from the nearest region boundary. This structure can be constructed in $O(n \log n)$ time. The expected query time of the OBST is optimal when the probability of a query point to lie in a region is proportional to the size of the region. The MDBST has no optimal expected query time, but according to some experiments it comes close. In the experiments the expected query time of the MDBST is at most 5% more than for the OBST. When increasing the number of regions we even saw a declining trend in the relative difference of the expected query time of both structures.

For 2-dimensional subdivisions with regions of constant complexity, point-location structures such as *Weighted triangulation refinement* and *Weighted trapezoidal decomposition*, have faster query times if the query point is inside a large region or far away from the nearest region boundary. But there are 2-dimensional subdivisions with non-constant complexity regions for which both structures fail both requirements. We showed that it is in fact impossible, to have a data structure that is guaranteed to have $O(\log(\frac{1}{size(R_i)}))$ query time, where R_i is the region containing the query point. It is possible though, to construct a point-location structure for 2-dimensional subdivisions with non-constant complexity, that has faster query times if the query point is far away from the nearest region boundary. An example is the δ_q-star quadtree, introduced in this thesis, for which the part after the stopping rule is based on the 1-dimensional MDBST. The MDBST itself is also adapted, such that it can be used for 2-dimensional subdivisions containing only horizontal and vertical edges. This structure is called MDBST-2D, and it is shown that it also has faster query times if the query point lies far away from the nearest region boundary. We experimentally compared the δ_q-star quadtree to the MDBST-2D. The MDBST-2D has always a better expected query time, and requires less space.

Unfortunately the MDBST-2D cannot handle diagonal edges yet. There are some ideas of how to do this, but these are not worked out. In the MDBST-2D the horizontal edges and vertical edges were in different groups. One idea is to group edges with similar slope together. Another idea is to first recursively partition the subdivision, with horizontal or vertical lines

through the endpoints of the edges. And then after some stopping rule, which is perhaps the same as for the star quadtree, start dividing also diagonally over the edges. The third idea is to make a hybrid between the second idea and a δ_q-star quadtree. Choose a dividing line through an endpoint if its relative close to the middle, else divide exactly in half. And use the stopping rule, and structure afterwards, as used in the δ_q-star quadtree.

References

[1] S. Arya, T. Malamatos and D. Mount. A simple entropy-based algorithm for planar point Location. *ACM Transactions on Algorithms* 3(2): 17 (2007).

[2] M. de Berg, O. Cheong, M. van Kreveld and M. Overmars. *Computational Geometry Algorithms and Applications*. Springer, 2008.

[3] M. de Berg, H. Haverkort, S. Thite and L. Toma. Star-quadtrees and guard-quadtrees: I/O-efficient indexes for fat triangulations and low-density planar subdivisions. *Computational Geometry: Theory and Applications* 43: 493-513 (2010).

[4] T. Cormen, C. Leiserson, R. Rivest and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[5] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM* 25: 905–910 (1982).

[6] J. Iacono. Expected asymptotically optimal planar point location. *Computational Geometry Theory and Applications* 29: 19–22 (2004).

[7] D. Kirkpatrick. Optimum search in planar subdivisions. *SIAM J. Comput.* 12: 28–35 (1983).

[8] K. Mulmuley. A fast planar partition algorithm, I. *J. Symbol Comput.* 10: 253-280 (1990).

[9] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufman, 2006.

[10] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry Theory and Applications* 1: 51–64 (1991).

Some tables with results of the experiments for 1-dimensional point-location structures.

♯regions	OBST	MDBST	$(OBST/n^3)10^6$	$(\mathsf{MDBST}/(n \log n))10^5$
10	0.043	0.012	43.027	35.731
20	0.055	0.011	6.867	12.344
40	0.203	0.008	3.168	3.689
60	0.552	0.020	2.554	5.758
80	1.208	0.024	2.359	4.647
100	2.281	0.032	2.281	4.756
120	3.770	0.027	2.182	3.197
140	5.825	0.041	2.123	4.088
160	8.550	0.044	2.087	3.713
180	11.989	0.064	2.056	4.761
200	16.340	0.072	2.043	4.703
400	138.109	0.219	2.158	6.334
800	1373.777	0.233	2.683	3.020

Table 1: Average construction times in milliseconds, experiment 1.

♯regions	Average MDBST/OBST	Maximum MDBST/OBST
10	1.010703229	1.072771914
20	1.006746017	1.04139978
40	1.005472646	1.027054879
60	1.006273478	1.019032613
80	1.004895461	1.015458239
100	1.004725213	1.015131395
120	1.005648701	1.013308495
140	1.005378261	1.013648433
160	1.004403343	1.011180706
180	1.003931625	1.00937304
200	1.0042721	1.009862491
400	1.00388304	1.006531437
800	1.003495534	1.00507212

Table 2: Relative difference in cost experiment 1.

♯Regions	Minimum OBST	Average OBST	Maximum OBST	Minimum MDBST	Average MDBST	Maximum MDBST
10	3	3.42	6	3	3.28	6
20	4	4.84	8	4	4.61	8
40	5	6.20	9	5	5.96	9
60	6	7.04	10	6	6.70	10
80	6	7.49	10	6	7.24	10
100	7	7.94	11	6	7.66	10
120	7	8.31	11	7	8.01	11
140	7	8.58	11	7	8.26	10
160	8	8.82	12	7	8.49	11
180	8	9.00	12	8	8.73	11
200	8	9.21	12	8	8.96	12
400	9	10.41	12	9	10.17	12
800	11	11.67	13	10	11.36	13

Table 3: Depth, experiment 1.

Fraction large	$[5, 10)$	$[50, 100)$	$[500, 1000)$	$[5000, 10000)$
0	1.004283983	1.004284913	1.004283309	1.004284665
0.03	1.01560761	1.025067452	1.018414091	1.015919917
0.06	1.017592887	1.021543963	1.016037392	1.014892902
0.09	1.018071169	1.020517294	1.011274517	1.010073094
0.12	1.01996699	1.01682772	1.013412912	1.012888929
0.15	1.020625089	1.012184567	1.006835021	1.006162765
0.18	1.017810622	1.012881529	1.008872291	1.008355721
0.21	1.014618902	1.013562393	1.011545298	1.011306369
0.24	1.01305453	1.011995515	1.010555087	1.01040555
0.27	1.012890639	1.009325947	1.007832987	1.00761598
0.3	1.013608351	1.006851752	1.005083955	1.004896244
0.33	1.013754949	1.006492798	1.004752716	1.004549843
0.36	1.012723379	1.007467773	1.006308313	1.006202419
0.39	1.011326463	1.008449826	1.007663802	1.007631493
0.42	1.009973357	1.008677177	1.008128674	1.008078175
0.45	1.008862729	1.008298669	1.007953728	1.00787666
0.48	1.007812931	1.00756735	1.007276384	1.00723348
0.51	1.006803091	1.006534974	1.006243324	1.006224685
0.54	1.005869089	1.005339649	1.005059076	1.005036425
0.57	1.005126865	1.004160063	1.003917735	1.003880702
0.6	1.004591786	1.003301779	1.003026908	1.002994238
0.63	1.00435852	1.002908787	1.002669939	1.002629955
0.66	1.00415015	1.002843063	1.002657674	1.002631035
0.69	1.003832153	1.002916672	1.002805514	1.002793075
0.72	1.003704303	1.003182341	1.003102322	1.0030808
0.75	1.003609408	1.003385126	1.003344305	1.003340843
0.78	1.003502814	1.003476542	1.003455274	1.003458201
0.81	1.003391456	1.003458096	1.003462252	1.00346497
0.84	1.003240016	1.003380698	1.003374454	1.003367314
0.87	1.003070441	1.003201929	1.003220815	1.003201005
0.9	1.0028805	1.002982474	1.002991151	1.003009617
0.93	1.002655585	1.002743717	1.002767504	1.00275158
0.96	1.002396591	1.00244652	1.002453485	1.002458054
0.99	1.002089521	1.002107413	1.002100112	1.002102185

Table 4: Average cost MDBST/OBST, experiment 2.