I/O efficient Cutting Trees

Jeppe Vroegindeweij

Supervisors: Frank Staals, Marc Kreveld, Sarita de Berg

A thesis presented for the degree of
Game and Media Technology

Utrecht University
Netherlands
June 2024

Contents

1 Introduction

Range searching is one of the most important and well-researched problems in the field of computational geometry. It tackles the following problem: We have a set of points N of size n in a plane and a query shape q. Store set N in a data structure, so we can efficiently find or count the points inside shape q. An example: The city of Utrecht wants to build a new park and has decided to place it in the neighbourhood with the biggest number of dog owners. They have a list of the locations of all households with registered dogs, which we use to create a data structure. After the building process, we can query each neighbourhood for the number of dog owners to find the best location to build the park.

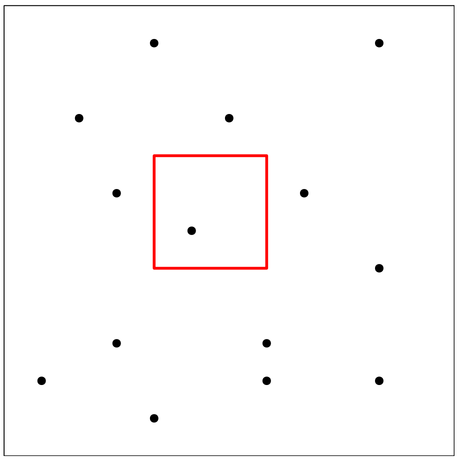
The main use of the cutting tree data structure is triangular-shaped range searching, which can be expanded to polygon shapes if it is combined with a triangulation algorithm. The structure has polylogarithmic query time and for any value of $\varepsilon > 0$ and dimension d, it has a space complexity of $O(n^{d+\varepsilon})$ [12]. This memory complexity makes a cutting tree too expensive for practical applications. However, this is the theoretical cost, so it is worthwhile to investigate this cost further in a practical setting. Another question is to see how the data structure fares when stored in external memory, which allows for a bigger storage capacity.

For the analysis of algorithms and data structures in an external memory setting, we can use the I/O model. The I/O model is used to investigate the cost of storing data in external memory for algorithms and data structures. This is done by calculating how many read and write operations are done between internal and external memory. This is relevant because the current bottleneck for external memory applications is communication between internal and external memory.

In this thesis, we will analyse the external memory efficiency of the cutting tree data structure using the I/O model. With this analysis, we intend to answer the question: How well does the cutting tree data structure perform when stored in external memory? We will also implement a cutting tree, which we will use to investigate the practical limitations of the data structure. With this experiment, we intend to answer the question: What are the benefits and bottlenecks of using the cutting tree data structure in a practical setting?

2 Range searching

Range searching, as the name suggests, is the problem of finding a subset of data points that fits into a certain query range. A query can be a square or a polygon, but also a half-space. Algorithms to solve range searching problems store the set of data points N in a data structure to handle query ranges efficiently. One of the main applications of range searching is for databases. In databases, data entries can be seen as a point in d-dimensional space and a database query is modeled as a range searching query.


(a) Orthogonal range searching

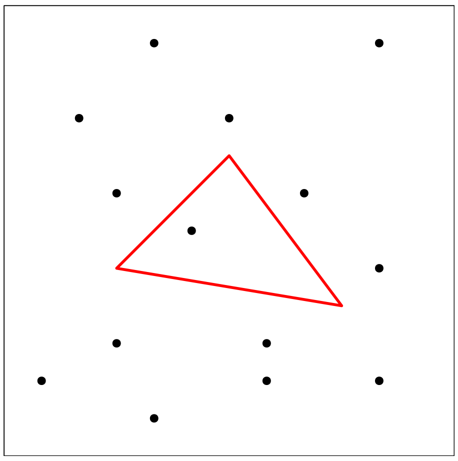

(b) Simplex range searching.

Figure 1: Range searching classes

Range searching algorithms can be divided into multiple categories of which orthogonal range searching and simplex range searching are the most notable. The difference between the two categories lies in the query ranges used by the algorithms. Orthogonal range searching algorithms restrict the query range to d-dimensional rectangles, so rectangles for $d = 2$ and cuboids for $d = 3$. Simplex range searching, as the name suggests, uses simplices as a query shape. A simplex is a generalization of a triangle and tetrahedron to higher dimensions. Figure 1 shows the difference between the two query types. Other query shapes are a half-space and a circle or sphere. Since the simplex is not bound by axis alignment, it is a more general descriptor, but this freedom also makes it more difficult to work with. Another important aspect of range searching algorithms is the tradeoff between query time and space complexity. A range searching algorithm can only optimise for one of these aspects. So, a data structure that optimizes query time, like the cutting tree, will do so at the cost of more memory. On the other hand, a data structure that optimizes for memory, like the partition tree, will have a low memory cost, but this comes at the cost of a slower query time.

2.1 Orthogonal range searching

Orthogonal range searching was one of the central problems in geometric algorithms in the 70's and 80's. At that time two techniques became prominent for solving such problems, first kd-trees and second range trees. Both algorithms were introduced by Jon Louis Bentley in the 70's. We will use his papers and the book [9] for descriptions of the data structures and algorithms.

2.1.1 Kd-trees

Bentley introduced the kd-tree [6]. The main idea of the technique is to adapt a 1-dimensional balanced binary tree into a d-dimension tree. In a balanced binary tree, a dataset is recursively split into two sides using the median value as a divider. This recursive process creates a tree structure, where the right subtree of a node contains the data points bigger than the divider and smaller points are stored in the left child. An example is shown in Figure 2.

A kd-tree takes this concept and applies it to higher dimensions. This means that the space in which the points sit is recursively divided by hyperplanes into hyperrectangles. To take the 2-dimensional case, we start with all points and sort them on a single axis, for example, the x-axis. From this sorted list we

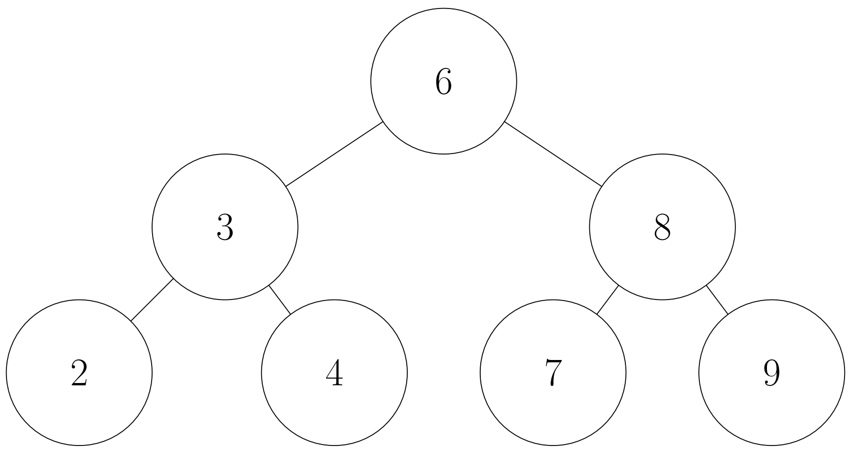


Figure 2: Example of a Binary Search Tree

take the median value and split the dataset into two sides, points with an x-value below the median on one side and points with an x-value above the median on the other. We then recursively go into those child nodes and further divide the dataset. An important note is that the dividing axis changes each time so nodes in the second tier will be divided on the y-axis. An example of how the space is divided for a kd-tree is shown in Figure 3.

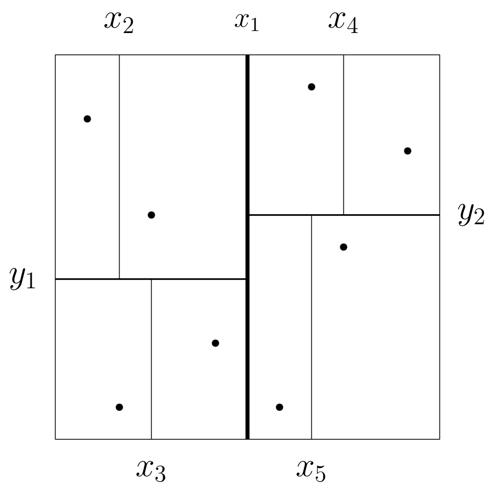


Figure 3: An example of a kd-tree

In higher dimensions, the dividing line will be a hyperplane, but the process stays the same. For a set of points N of size n, a kd-tree can be constructed in $O(n \log(n))$ time and will have a space complexity of $O(n)$. A rectangular range query can be performed in $O(n^{1-1/d} + k)$ time, where k is the number of data points reported [8]. From this summary the strength of kd-trees is clear, the linear space cost is ideal, but the query time is a big constraint.

2.1.2 Range trees

A range tree is optimized for query time at the cost of extra needed memory and was first introduced by Bentley [7]. The idea of a range tree is to create a multi-dimensional tree structure. Again we begin with a binary search tree. This structure allows us to answer 1-dimensional range queries. To adapt the data structure for 2-dimensional range queries, we add a sub-tree to each node of the 1-dimensional range tree. This subtree handles range queries on another axis than the parent tree. So, for a 3-dimensional query, we need three layers of trees. In the first range tree, we handle the x-axis attribute of a query. In each node of the x-axis tree, we store a subtree that handles the y-axis of a query. Nodes in the second layer of the tree contain a further subtree to handle the z-axis. It is important to note that these subtrees only contain the set of points that are inside the range of the parent tree. An example scheme of the range tree structure is shown in Figure 4.

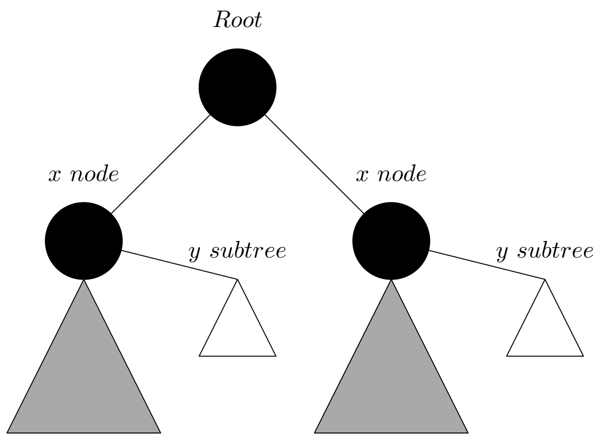


Figure 4: Structure of a range tree

This method uses more space, since we have to store points multiple times in different subtrees. The size of the tree is $O(n\log^{d-1}(n))$ and the tree is built in $O(n\log^{d-1}(n))$ time. Range counting queries cost $O(\log^{d-1}(n) + k)$ time, where k is the number of data points reported. This clearly shows the tradeoff present in range-searching structures, since the range tree has a significantly faster query time compared to the kd-tree, but at the cost of a higher memory cost.

2.2 Simplex range searching

Simplex range searching became a popular problem for geometric algorithms research in the 90's. Simplex-shaped queries give more freedom to the user, but this also means that the data structures handle more complex calculations. Just like with orthogonal range searching, there are two main algorithms in simplex range searching: partition trees and cutting trees. Both of these algorithms are explained in [1], [2] and in the book [9].

2.2.1 Partition trees

The basic idea of a partitioning tree is to divide the space into partitions, such that each partition contains roughly the same number of data points. This idea was first used by Willard [22], who proposed to divide the space up into four simplices, using two lines, such that the points were spread equally among them. This of course can be repeated for each of the simplices, creating a tree structure. A query on this structure checks whether the query shape intersects a partition, if that is the case the partition is further recursed upon. The resulting structure has $O(n)$ memory complexity, since each point is stored only once. The structure can be constructed in $O(n \log(n))$ time. For the query time, an important observation is the fact that any half-line can only intersect three of the four simplex spaces, thus for each node, the structure has to check at most three child nodes. This results in a query time of $O(n^{\alpha})$, where $\alpha = \log_3(4) \leq 0.7925$.

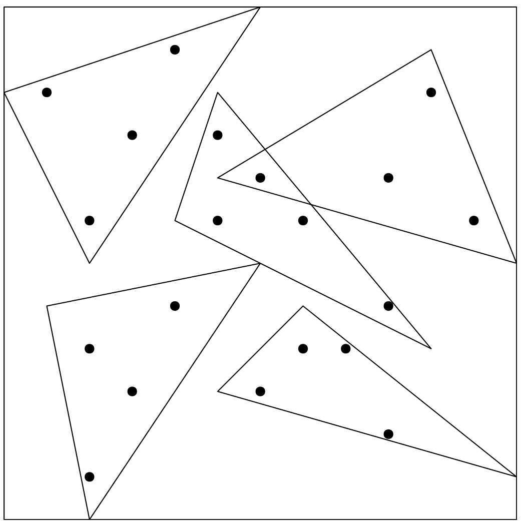


Figure 5: Example of a partitioning

In general improvements on this structure aimed to reduce the crossing number or minimal number of simplices intersected by a halfspace query. The best solution was found by Matoušek [18]. He proved that for a set of points N and a given parameter $1 < r \leq n/2$, the set N can be divided into subsets $N_V \subset N$, such that the size of N_V is bounded by $n/r \leq |N_v| \leq 2n/r$. Each subset is then fit into a triangle t_i, while the crossing number of any hyperplane h is at most $\alpha r^{1-1/d}$, where α is a constant. This allows him to create a partition tree that works the same as the one created by Willard, but the query time improves to $O(n^{1-1/d})$. He did this by focusing on partitioning the points instead of focusing on partitioning the space. The resulting partition tree thus is a structure that has a linear memory complexity at the cost of query time, just like the kd-tree.

2.2.2 Cutting trees

The idea of using cuttings for simplex range searching was first introduced by Clarkson [12]. The idea of a cutting is to divide the space into disjoined triangular regions or simplices, such that the data is evenly spread among them. An example of this is shown in Figure 6.

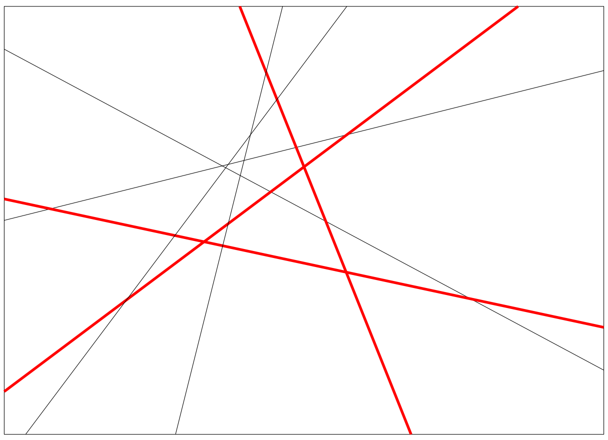
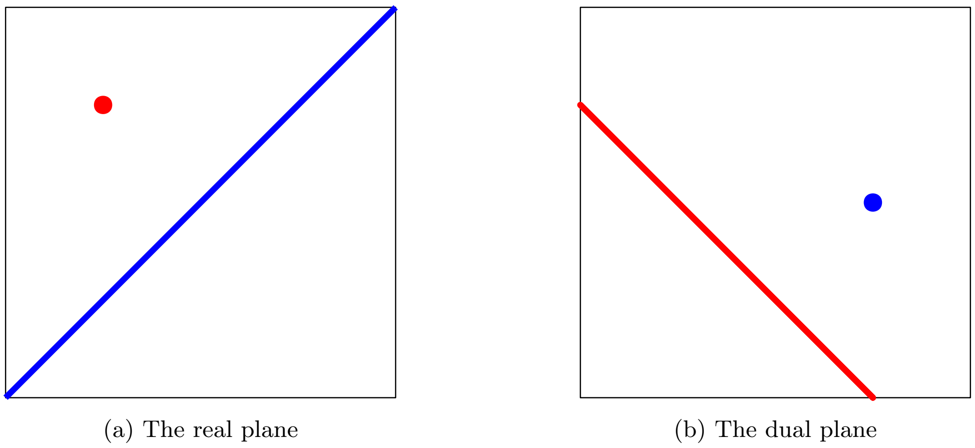


Figure 6: Example of a cutting, the red lines divide the space

To do this we first transform the n points in N into lines in the dual-space. A line in 2-dimensional space can be described as $y = ax + b$, so by substituting a and b with the x and $-y$ value of a point n_i, we define a line l_i, which is the dual of n_i. Of course, the opposite is also possible, by substituting the a and b values into the x and $-y$ of a point. This is used to transform a query half-space into its dual form: a point. The dual space has multiple properties of which two are important for us. First, the vertical distances between objects are preserved and second the order of the objects is reversed. This is shown in Figure 7.



(a) The real plane (b) The dual plane

Figure 7: Example of duality

Given a set of lines L and a random sample $R \subset L$, such that $r = |R|$ and $1 \leq r \leq n$. We can use R to create an arrangement of lines $\Delta(A_r)$, which is a way to divide space into cells. Clarkson [12] proved that the arrangement then has a probability of at least 1/2 of having the characteristic that each cell or simplex of the triangulated arrangement is intersected by at most $O(\frac{n \log(r)}{r})$ lines of L if r is a sufficiently large value. We call such an arrangement a cutting. The size of the cutting is $O(r^2)$, which denotes the number of triangles or cells of the cutting. Thus using this proof and the dual-space Clarkson created a new structure to solve simplex range searching queries: the cutting tree.

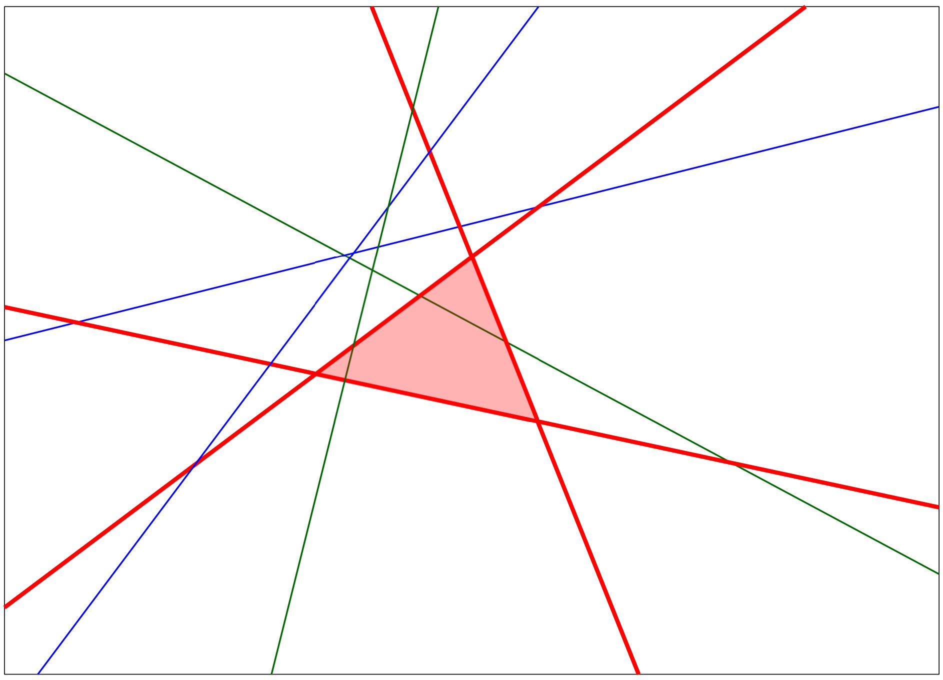


Figure 8: Example of a cutting: The red lines indicate the sample, the blue lines are above the selected cell and the green lines intersect the cell

The cutting tree structure is as follows: Each of the simplices of the created cutting becomes a child node of the root or parent node. In each of these nodes, we store the set of lines that are strictly above the corresponding simplex, fully below the simplex and that intersect the simplex. In each node, the set of intersected lines is then used to create a new cutting and thus child nodes, which is how the tree structure is made.

A half-space range counting query can be translated to a point q in the dual space. For q we want to count all the lines that lie strictly above or below it, depending on the query. For each node, we first calculate in which cell of the cutting the query point q lies. From that node, we gather the size of the set of lines that lie above or below the simplex and recurse on the child cutting. Let $Q(n)$ be the query time on a cutting tree. Then $Q(n)$ satisfies the following recurrence:

$$Q(n) = \begin{cases} O(1) & \text{if } n = 1 \\ O(r^2) + Q(\frac{n\log(r)}{r}) & \text{if } n > 1 \end{cases}$$

The recurrence solves to $O(\log(n))$, if r is a sufficiently large value. The space complexity of this structure is $O(n^{2+\varepsilon})$ and it can be constructed in $O(n^{2+\varepsilon})$, where $\varepsilon > 0$ is a practically small constant value [12].

To adapt the structure to triangular queries we can use the same technique used in range trees. A triangular range query can be seen as the combination of three distinct half-space queries. Each of these half-spaces can be transformed into a point in the dual space resulting in three query points: q_1, q_2 and q_3. The query triangle is intersected by a line l_i if that line sits below one of the query points and above another query point, it depends on the shape of the triangle, which query point is above or below. For the tree structure of a triangular range query, this means that in the first layer or level of the tree, we handle for example finding all the lines that are above q_1. In each node of this tree, we then store a subtree that is used for finding all the lines that are below q_2 and q_3 is handled in subtrees of the second level. The upside of this technique is that it comes at no additional asymptotic memory cost since the subtrees are created on the set of lines that are not eliminated by its parent node. The downside of the additional sub-layers is that the query time goes up to $O(\log^3(n))$.

While Clarkson used randomization to find a fitting cutting others have found deterministic algorithms to find a cutting of size $O(r^2)$. Matoušek [16] was the first to find such a deterministic algorithm. Afterwhich multiple algorithms were introduced that improved the running time of finding the cuttings deterministically, notably Matoušek [17] showed that it was possible in $O(n\log(r))$ if $r \leq n^{1/d-1}$. The best deterministic solution comes from Chazelle [10], who proved it was possible to find a cutting deterministically in $O(nr^{d-1})$ time.

3 I/O-efficiency

I/O analysis of algorithms and data structures is focused on minimizing the number of read and write operations between internal and external memory. For this analysis the I/O model was created by Aggarwal and Vitter [3]. Other analysis into external memory efficiently had been done before, but from that point on most of the research in this area is done using their model. The I/O model consists of two modules, the internal memory on one side and the external memory on the other. The model thus abstracts away the concept of internal caching. This however is not a problem since the cost of read and write operations surpasses that of internal memory operations significantly. All operations must be done in a CPU that uses data stored in internal memory. Transporting a single block of data between internal and external memory is considered an I/O operation and the complexity of an algorithm is the number of I/O operations needed to perform the algorithm. This means that operations performed on data in internal memory have no I/O cost.

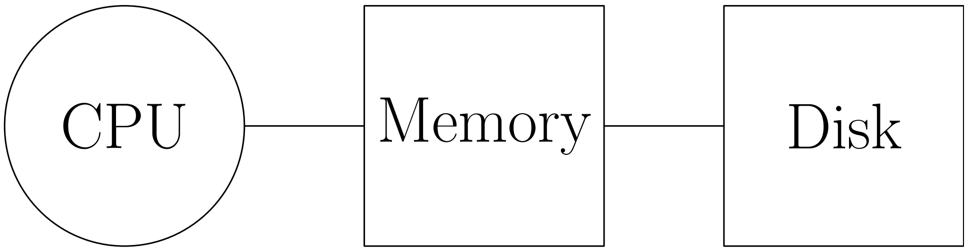


Figure 9: The I/O model

To summarize the model: Memory is divided into two boxes, first, we consider internal memory which is of size M and all calculations must be done on data stored in internal memory. The second box is the unbounded external memory. Both are stored in blocks of size B. Given a problem with n items, the goal is to optimize the number of blocks that need to be transported between internal and external memory to come to a solution.

3.1 Sorting algorithms

One of the first I/O problems that was researched was that of sorting a list in external memory. Aggarwal and Vitter [3] came up with two solutions for sorting on external memory.

First, we consider (external) merge sort. The idea behind merge sort is to first collect data into buckets and sort them. The buckets are then gradually merged to create a fully sorted list. In the first stage of the process $\Theta(n/M)$ buckets are created. A bucket consists of M/B blocks of data, which are sorted in internal memory and then written back to external memory. This results in $\Theta(n/M)$ sorted buckets in external memory and takes $O(n/B)$ I/O's, since each block is needed only once in internal memory. The second stage of the sorting process handles merging the created buckets. This is done by merging $M/B-\ 1$ buckets at a time. Each of the $M/B-1$ buckets submits their smallest block to internal memory, where the remaining empty block is filled by tacking the smallest data points in the current internal memory and writing back that block into external memory when it is filled. Each merging level costs $O(n/B)$ I/O operations since blocks are only replaced if all the items inside have been merged. The total sorting time then depends on the number of levels needed to sort the list, which is $O(\log_{M/B}(\frac{n}{B}))$. Resulting in a total I/O efficiency of $O(\frac{n}{B}\log_{M/B}(\frac{n}{B}))$.

The other external sorting algorithm uses distribution and thus works the other way around. First, the list is distributed into $M/B-1$ buckets, such that the first bucket contains the smallest items the second bucket contains the smallest items after that and so on. To do this each bucket is assigned a block of internal memory. The remaining block is then used to supply the data from external memory, which is distributed over the $M/B-1$ output buffers. This process is then done recursively on the created buckets, so in each level of the distribution the buckets become smaller in size. When the buckets fit in internal memory the entire bucket is loaded and sorted at once, since the buckets are already in order this creates a fully sorted list. In each level of the distribution each block of memory is used once in internal memory and thus costs $O(n/B)$ I/O operations. The number of levels needed to distribute the buckets down to size M is again $O(\log_{M/B}(\frac{n}{B}))$. So this algorithm also has a total I/O efficiency of $O(\frac{n}{B}\log_{M/B}(\frac{n}{B}))$.

Both of these techniques are optimal in terms of I/O efficiency. The important takeaway here is the two techniques used to make the algorithms I/O efficient. First, the distribution of data into chunks of size M/B, since the CPU can then operate on those chunks of data with no further I/O cost. The other technique is the use of buffer blocks. These techniques are frequently used for I/O efficiency problems.

3.2 B-trees

Binary search trees are one of the basic building blocks of computer science since they provide an efficient way to store and query data points. However, they are not optimised for use in external memory. B-trees in essence are the external memory variant of binary search trees and the understanding of this data structure is essential for understanding the I/O efficiency of more complicated tree structures. B-trees were first introduced by Bayer and McCreight [5], but we will also use the descriptions from Comer [13], since he also discusses some variants and optimizations of the B-tree.

To explain B-trees we start with the structure on which it is based, the binary search tree. Nodes in a binary search tree store a key item to divide the data into items with a smaller value than the key and items with a bigger value than the key. These two sets are stored in separate subtrees. These child trees then further divide the subsample into smaller and smaller samples until individual items are stored in the leaves of the tree. The dividing keys in the nodes are also part of the set of items. This means that the path of a query item q_i depends on a series of comparisons to decide, which path to take. An example of a Binary search tree is shown in Figure 2.

B-trees generalize this idea. Instead of constraining a node to one dividing key, each node can store multiple keys. This allows us to split the data sample into multiple subsamples. The first subsample is the range of items strictly below the smallest dividing key, the second sample contains the items between the first and second dividing keys and so on. In general, each node of a B-tree of order d has at most $2d$ keys and $2d + 1$ pointers. Each internal node also has a minimum of d children. This means that the B-tree keeps a useful characteristic from the binary search tree: it is a balanced structure. To explain why this is the case, we will review a B-tree's update operations.

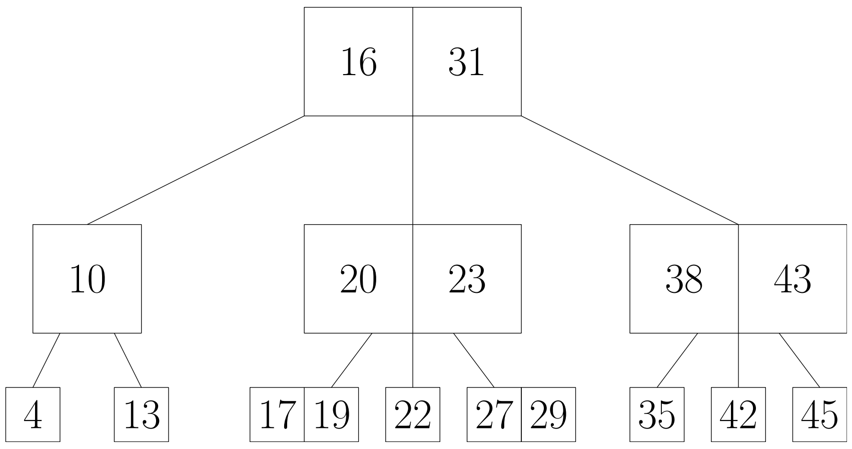


Figure 10: Example of a B-tree

The first operation to consider is an insertion, which has two parts. In the first stage, we find a suitable leaf for the insertion of the item. Then we re-balance the structure back up from the leaf to the root. For re-balancing, if a node has room to spare, we can simply add the item to the leaf. However, if a node overflows on an insertion, then a split occurs. In this split, the smallest d items are kept in one leaf and the biggest d items are stored in another leaf. The middle item is then stored upwards in the parent node as a dividing key. If the parent node is also full, then the same split occurs recursively until the root node is reached. In the case of delete operations, there are two possibilities. First, we will consider the case where the item we want to delete is stored in an internal node. This means that the deleted dividing key needs to be replaced. For this, we take the leftmost value of the right subtree. If this leaves a leaf with less than d keys we need to re-balance the structure. For this, we can borrow keys from neighbouring subtrees, but this distribution of keys can only occur if there is a total of more than $2d$ keys and items, else we need to concatenate leaves into a node. This is the opposite of a split from the insert operation, this process can also propagate up the tree. When deleting a leaf node, re-balancing is only needed if the parent node has below d children.

Now we can consider the cost of the operations. First, for search operations, we consider the height of the tree $h \leq \log_d(\frac{n+1}{2})$ [13]. This follows from the fact that each node, except the root node, has between d and $2d$ direct descendants. The time cost of insertions and deletions depends not only on the height of the tree, but also on whether the operation propagates up in the tree. This results in a time cost of $O(\log_d(n))$ in the worst case. To adapt the B-tree to external memory we set d to be $B/2$, thus fitting an internal or leaf node into one memory block. The structure has a linear memory cost of $O(n/B)$ and a height of $O(\log_B(n))$. The I/O cost of updating the tree is $O(\log_B(n))$ since that is the maximum number of times that an update can propagate up the tree.

We can now see that the strength of the B-tree lies in the branching factor d, which decreases the cost of operations for larger degrees, but the structure also has some disadvantages. The first disadvantage is poor memory utilization. The B-tree as described ensures that all nodes are at least half-full, which means that half the reserved space is empty. Another problem with B-trees is that they are not well suited for sequential processing. This is a product of the fact that node keys are also items. Lastly, B-trees are meant to be used in an online setting. This means that updates on the tree are done sequentially and thus the data structure is not optimized for the offline setting. In an offline setting, all the data is known beforehand and updates are done more efficiently in bulk or a lazy manner.

The first disadvantage is easily overcome: Instead of ensuring that nodes are half-full, ensure that nodes are at least two-thirds full. The only change required to achieve this is to split nodes into three instead of two when a split occurs.

Storing items as keys for the trees has disadvantages, so in the B$^+$ tree [13], all items are stored in leaves. This allows for better sequential processing, because most neighboring items will be located in a similar part of the tree structure and at the same level. Apart from better sequential processing, there are other advantages to abstracting items from keys. Since keys do not have to be items, we get more freedom in choosing those keys. As long as a key fits the range of items it divides, the key can be anything. This also means that the deletion update, without underflow, only needs to delete the item. A further variant, the prefix B$^+$-tree, is used to store strings in a tree structure. In this case, keys become the shortest prefix that divides the sub-trees, thus saving storage.

For the last disadvantage, the B-tree not being suited for an offline setting, Arge introduced buffer trees [4]. The main idea of buffer trees is to add a buffer of size $\Theta(M/B)$ to each node in the tree, in addition, each leaf now stores B items. Updates to the tree themselves become items that are added to the buffer of the root node, but do not immediately propagate down the tree. Instead, update operations are stored in the root until the buffer overflows, at which point the node begins a buffer-emptying process in which the blocks of the buffer are sorted and distributed over its child nodes. The main improvement of this structure is that the height of the tree gets reduced to $O(\log_{M/B}(\frac{n}{B}))$, the I/O cost of doing n insertions becomes $O(n \log_{M/B}(\frac{n}{B}))$, instead of the original I/O cost for this query in B-trees $O(n \log_B(n))$. Another improvement is that lazy structure allows us to handle a lot of operations simultaneously in an offline setting. One important note about the process in buffer trees is that some items might still sit in buffers after all the operations have been completed, so a final emptying action needs to be done that costs $O(n/B)$ I/O's.

3.3 Distribution sweep

Distribution sweep is an external memory variant of the plane sweep paradigm and was introduced by Goodrich [15]. The plane sweep technique examines the set of items N in sequence along a given dimension and keeps track of the data using a dynamic data structure, for example, a search tree. In external memory such dynamic structures do not provide an optimal I/O solution. Distribution sweep takes a different offline approach to the sweeping problem using distribution sort. The idea of distribution sweeping is to divide the data into $O(M/B)$ strips, each containing an equal number of data points. Handling the sweep of each strip can be done simultaneously. The remaining interactions across strips can be handled recursively afterwards. This results in a I/O complexity of $O(M/B + Q/B \log_{M/B}(n/B) + Q/B)$. The technique can be used for typical plane sweep problems like segment intersection reporting and the all-nearest neighbours problem.

Chiang [11] describes a practical experiment on the distribution sweep proposed by Goodrich. They compare distribution sort with three other sweeping algorithms using different dynamic data structures: a dynamic B-tree, a 2,3,4-tree and a 2,3,4-core-tree. The problem to be solved was: Given a set of line segments S report the number of intersecting segments K. For S they used three random generators to generate uniformly distributed sets of line segments with differing attributes. On a set of short line segments with a small number of intersections, the B-tree and 2,3,4-tree perform better than the distribution sweep. In this case, the dynamic tree structure made is relatively small and thus performs better than the distribution sweep. This is an interesting interaction that can only be found by doing experimental research. On the other sets of line segments distribution sort was the best, because those sets contained more intersections. Another observation was that B-trees always had the biggest I/O cost by a big margin. This shows that we get valuable insight from practical tests on these data structures and algorithms.

4 I/O-efficiency analysis cutting trees

Now that we have an idea of how the cutting tree works, as well as how we can analyse the data structure in an external memory setting, we can start the I/O analysis. First, we will go over each part of the building process. Starting with the dual transform in Section 4.1. Then we look at the sampling process in Section 4.2 and how to do the line-cell intersection in Section 4.3. In Section 4.4, we look at the building process as a whole. Lastly, we look at the query cost in Section 4.5.

4.1 Dual conversion

The first step of building the cutting tree is applying the dual transform to the set of input points N. This creates a set of lines L, as explained in Section 2.2.2. To transform a block of points in N to lines in L, we read a block of data from external memory into internal memory. We apply the transformation to all the points in the block, this has no I/O cost. Afterwards, we write back the resulting lines into external memory. The I/O cost of converting a set of points N into its dual counterpart is $O(n/B)$.

Lemma 4.1. *Let N be a set of n points in the plane, stored in external memory. Then, converting N into its dual space equivalent set of lines L costs $O(n/B)$ I/O operations.*

4.2 Sampling process

Given a set of lines L, we want to construct a cutting, a set of disjointed simplices that cover the plane, with the characteristic that the lines are as evenly divided over the cells. For this, we use an algorithm that Clarkson introduced [12], which was later improved by Chazelle [10]. To construct the cutting we need to create a sample set $R \subset L$.

In Section 2.2.2, we saw that the probability that a random sample R has the characteristic that each cell is intersected by at most $O(\frac{n \log(r)}{r})$ lines of L, is at least 1/2 if r is a large enough value. We can thus assume that we can find such a sample in a constant number of attempts. This idea can be used in the external memory setting, but we must decide how to select a sample.

The sampling method should pass through the set of lines L once. This way we can use a single input buffer block to feed the lines to the algorithm. Reservoir sampling, introduced by Vitter [20], fits our needs. The algorithm creates a simple random sample of a fixed size without the knowledge of the set size, where each item has the same probability of being selected for the sample. We use reservoir sampling to pick a sample of size r. The method starts by filling a reservoir or array A of size a with the first a lines $l_1, ..., l_a$ from the ordered set of lines L. Then the algorithm passes through the remaining lines of L. The lines are read into internal memory through the single input buffer. For each new line l_t, a random number j between $[0, t]$ is generated. If the generated integer $j < a$, then $A[j] := l_t$. The probability that a new item enters the reservoir is the same as the probability for each item to be retained, this is why the created sample is random. For the algorithm to pass through each line once we need to store the reservoir in the remaining blocks. This means that the maximum size a_{max} of set A is $a_{max} < M - 1$. This also means that $r < M - 1$, since we pick a sample of size r. The I/O cost of reservoir sampling is $O(n/B)$ since each line is passed through internal memory once.

Lemma 4.2. *Let L be a set of n lines, we can then use reservoir sampling to pick a simple random sample $R \subset L$ in $O(n/B)$ I/O operations.*

4.3 Line-cell intersection handling

After the sampling process, we can construct a cutting. A cutting is a set of disjointed simplices that cover the plane. The cutting is constructed by triangulating the sample R, the resulting cutting has $O(r^2)$ simplices. After the construction of the cutting we still need to handle the line-cell intersections and validate the cutting. Each simplex of the cutting should intersect at most $O(\frac{n \log(r)}{r})$ lines of L.

Let a cutting C have c simplices, where $c = O(r^2)$, such that $c < M/B - \alpha$, where $\alpha > 1$ is a small constant. If this is the case, then we can use the space in the internal memory in the following way: one input buffer to load in a block of lines one at a time, c blocks of output buffers, one for each simplex and the remaining blocks to store the simplex information. This simplex information contains the location of the simplex and an integer to keep track of the lines above the simplex.

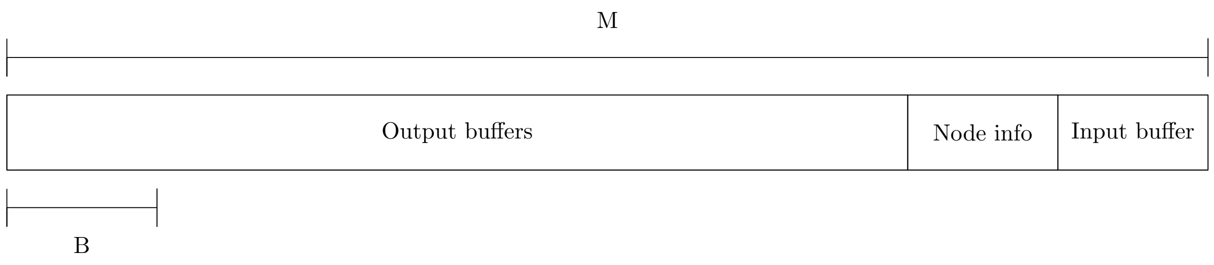


Figure 11: Internal memory layout for line-cell intersection handling

The conflict lists and counters of lines above the simplex can then be constructed in the following way. Load in a block of line data from L into the input buffer. In internal memory, naively check for each line and simplex pair whether the line is above, below or intersecting the simplex. For lines above the simplex, simply add to the above counter. For lines intersecting the simplex, we store them in the output buffer block assigned to that cell. If the output buffer is full write it back into external memory.

This distribution of internal memory space makes it so we only need to read each block of data into internal memory once. So the I/O cost of reading L is $O(n/B)$. The writeback cost is dependent on how many lines intersect each simplex. We assume that each simplex has at most $O(\frac{n\log(r)}{r})$ intersections, so the I/O cost is $O(c(\frac{n\log(r)}{r}/B)$. A cutting has $O(r^2)$ cells, so $cn\frac{\log(r)}{r} > n$. This means that we can leave out the cost of reading the lines.

Lemma 4.3. *Let C be a $\frac{1}{r}$-cutting with c simplices and L be a set of n lines, then we can produce the conflict list of the cutting at an I/O cost of $O(c(\frac{n\log(r)}{r})/B)$.*

During the line-cell intersection process, we simultaneously validate that the cutting has no cell with more than $O(\frac{n\log(r)}{r})$ intersecting lines. Once a cell has more intersecting lines than the allowed limit we know that the cutting is invalid and we restart the process by selecting a new sample. From Section 4.2, we expect to find a valid sample in constant tries. This means that the I/O construction time of a cutting is $O(r^2(\frac{n\log(r)}{r})/B)$.

Lemma 4.4. *Let L be a set of n lines, then the I/O cost of constructing a valid cutting is $O(r^2(\frac{n\log(r)}{r})/B)$, where r is the size of the random sample R used in the construction process.*

4.4 External memory cutting tree

In this section, we discuss how we can combine the previous steps to construct a cutting tree. Starting with the conversion to the dual space, as shown in Section 4.1.

The tree structure is achieved by recursively creating cuttings and pairing subtrees to the created cells. In an inner node, we store the accumulated lines above the current subtree and a cutting, where the cells refer to subtrees. In leaf nodes, we store the final number of lines above the subtree and a list of lines that are still in contention. The input for a recursive call to create a subtree is the subset of lines $L_c \subset L$ that conflicts with the parent cell, if $L_c < M$ we create a leaf node instead.

From the previous sections we gather that the I/O cost of building the entire tree structure is $O(n^{2+\varepsilon}/M)$, the proof is shown below and uses induction.

Lemma 4.5. *The following recurrence, that signifies the cost of constructing the cutting tree, solves to $O(n^{2+\varepsilon}/M)$:*

$$T(n) = \begin{cases} O(n/B) & \text{if } n \leq M \\ O(rn\log(r)/M) + T(\frac{n\log(r)}{r}) & \text{if } n > M \end{cases}$$

Proof. We will induct on n. First, we prepare the recurrence for induction by eliminating the big O resulting in the following definition: $T(n) = ar^2T(\frac{n\log(r)}{r}) + \frac{bnr\log(r)}{M}$, where $a, b > 0$ and r is a sufficiently large value lower than n. Important to note is that $n\log(r)/r < n$. The base case solves to $T(n) = O(n/B)$ for $n < M$. For the induction hypothesis, we assume: $T(n') \leq cn_0^{2+\varepsilon}/M$, for all $0 < n' < n$, where $c \geq 2b$.

$$
\begin{aligned}
T(n) &= ar^2T(\frac{n\log(r)}{r}) + \frac{bnr\log(r)}{M} \\
&\leq ar^2\frac{(c(\frac{n\log(r)}{r})^{2+\varepsilon})}{M} + \frac{bnr\log(r)}{M} \\
&\leq \frac{ac(n\log(r))^{2+\varepsilon}}{r^{\varepsilon}}/M + \frac{bnr\log(r)}{M} \\
&\leq (\frac{a\log^{2+\varepsilon}(r)}{r^{\varepsilon}})\frac{cn^{2+\varepsilon}}{M} + \frac{bnr\log(r)}{M}
\end{aligned}
$$

For $r\log(r) \leq n^{1+\varepsilon}$ and $b \leq \frac{c}{2}$, we get $bnr \leq bn^{2+\varepsilon} \leq \frac{1}{2}cn^{2+\varepsilon}$.

For a large enough value of a', the following holds: $a'\log^{2+\varepsilon}(r) \leq a'r^{\varepsilon/2}$. We use this to solve the following:

$$
\begin{aligned}
&\frac{ar^{\varepsilon/2}}{r^{\varepsilon}} \leq \frac{1}{2} \\
&\frac{a}{r^{\varepsilon/2}} \leq \frac{1}{2} \\
&2a \leq r^{\varepsilon/2} \\
&r \geq (2a)^{1-\varepsilon/2}
\end{aligned}
$$

Now we can substitute in $(\frac{1}{2})cn^{2+\varepsilon}$ and solve the induction:

$$
\begin{aligned}
T(n) &\leq \frac{(\frac{1}{2})cn^{2+\varepsilon}}{M} + \frac{(\frac{1}{2})cn^{2+\varepsilon}}{M} \\
&\leq \frac{cn^{2+\varepsilon}}{M}
\end{aligned}
$$

By the principle of induction, the claim holds for all $n > 0$.

■

Compared to the original data structure, the I/O efficient version of a cutting tree described in this section is faster by a factor of M. This means that storing a list of lines in the leaf nodes makes the data structure less costly. In essence, this eliminates the deepest levels of the tree structure. What makes this even more significant is that the deepest layer of the tree is the most costly to create. Each subsequent layer has $O(r^2)$ more nodes, but each subsequent layer has $\frac{n\log(r)}{r}$ intersecting lines. This means that the cost of constructing a layer can be calculated by the following formula $f(i) = n(r\log(r))^i$, where i is the depth of the layer. This also shows that for each value of r, each subsequent layer is significantly more costly than the previous layer. This means that eliminating the deepest layers eliminates the most costly layers of the data structure.

4.5 Query cost

For a half-space range counting query, the input is a half-space and the output is the number of points inside the half-space. Assuming the query line is already in internal memory, we can freely transform it into a query point q, that is its dual space equivalent.

Next, we start by reading in the root node and performing a point-location query on the cutting stored in the root. This is done naively in our case: we read the cutting into internal memory and check for each cell if q is inside that cell. Once we find the correct cell. The I/O cost of reading the entire cutting from external memory is $O(r^2/B)$. This process is recursed until a leaf node is reached.

For the leaf node, we still need to check for each of the remaining lines if it is above or below q. We can then report how many points sit inside the query half-space. The I/O cost of this process is $O(M/B)$, since each node has at most M remaining intersecting lines.

The total I/O cost is the sum of costs for each inner node and the cost of a leaf node. The depth of the cutting tree is $O(\log(n)/M))$, this follows from the fact that we stop creating nodes when the number of remaining lines is smaller than M. This means that the total I/O query cost is $O(r^2\log(n)/BM)+M/B)$.

Lemma 4.6. *Let there be a cutting tree on a set of points N and let q be a query half-space. We can find the number of points inside q, using the cutting tree at an I/O cost of $O(r^2 \log(n)/BM) + M/B)$.*

The most prominent factor here is the cost of reading in the cutting of each inner node. A more efficient external memory point-location algorithm, rather than naively checking each cell, could improve the query cost significantly.

Theorem 4.7. *Let N be a set of n points on the plane. We can construct an external memory cutting tree at an I/O cost of $O(n^{2+\varepsilon}/M)$. The I/O cost of the half-space range counting query on the created data structure is $O(r^2 \log(n)/BM) + M/B)$, where $r < M/B - \alpha$, where $\alpha > 1$ is a small constant.*

5 Cutting tree implementation

In the previous section, we discussed the theoretical external memory cutting tree. In this section, we will look at the practical implementation of the cutting tree. This implementation is again focused on half-space range counting queries. So for the building phase, the input is a set of points and for a query, the input is a line or half-space.

5.1 Libraries and tools

For the implementation of the cutting tree, we have chosen to use C++. A big factor in the choice of language was the available literature and libraries built for C++, since other research into geometric algorithms was also done using C++. For the geometric computations, we will use CGAL [19] (Computational Geometric Algorithms Library). The arrangement functionality of CGAL [21] allows us to create the cuttings and handle the point-location queries. We will also use CGAL for triangulation [23] and some basic geometric functions.

To store the cutting trees in external memory we use the STXXL library [14]. This library takes normal STL structures, like lists, stacks, and vectors, and allows us to store them in external memory, while we can still use them like normal STL structures. We use the vector functionality to store the nodes of the cutting tree. STXXL also provides some useful tools to keep track of the performance of the program.

5.2 Nodes and storage

A model of the basic storage structure is shown in Figure 12. We store each tree node in an `STXXL::vector`. To store data in a file we need to format the data to only contain raw data, like integers, floats and characters. This means that we can not store the CGAL classes and variables, since those contain pointers and other structures that are incompatible with the `STXXL::vector`. Instead, we use the built-in CGAL arrangement IO functionality to store each arrangement in a unique text file. Each inner node then contains an integer value `cuttingIndex` to point at the location of the arrangement file. Furthermore, in the leaves of the tree, we store the remaining conflicting lines by taking the m and $-b$ values of the line ($y = mx + b$) and storing them in an array of doubles `lineM` and `lineB`. We can use these doubles to reconstruct the lines during the query process.

Since the nodes are stored in a vector, we use the index of the sub-nodes in the vector to point from nodes to their subtrees. So, a node keeps an array of indices (`int[`r^2`]childIndex`) that point to sub-trees. For each face of the arrangement we then store an index of the child array. This way the cells of the arrangement are connected to the corresponding sub-trees.

The remaining data is stored in the node. The number of lines above the current sub-tree (`int above`) and its id (`int id`). For the `STXXL::vector` the structure it stores must fit into the blocks and pages of a memory file. Since this is not necessarily the case for the tree nodes we use a container struct to store the nodes. The container is a character array of size 2^i, where i is the smallest number that would allow the node to fit, however, all containers should be the same size. This is done by copying the memory of the node directly into the character array, which in essence is just an array of bytes.

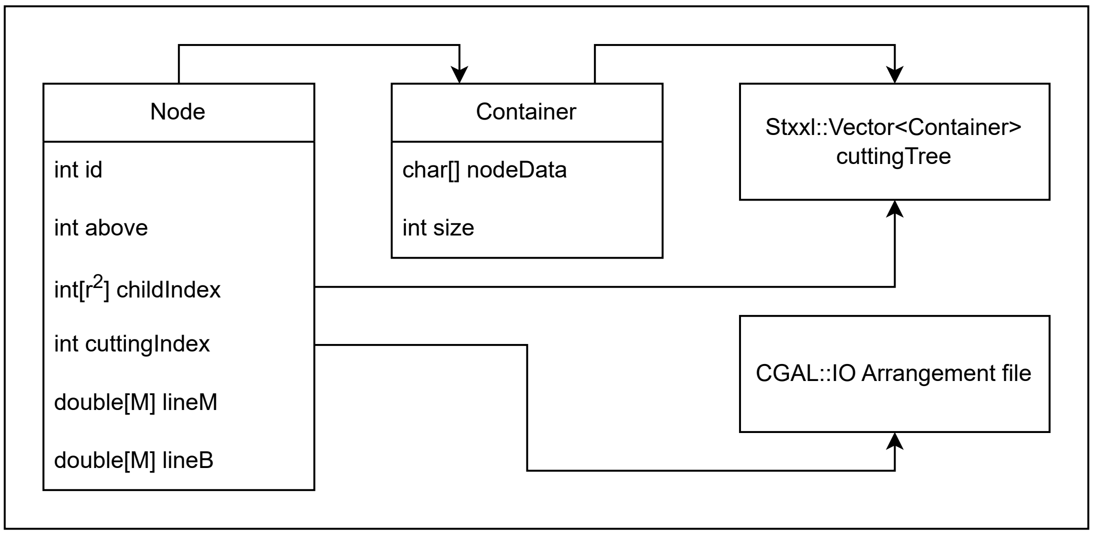


Figure 12: Basic Cutting Tree memory structure

One easy way to save some memory storage is to create a different node catered to leaves and inner nodes since they do not store the same data. In inner nodes, we need the arrangement index and array of sub-tree indices, while the leaf nodes need to store the lines that remain conflicting. Creating distinct structures for both nodes allows us to store the nodes in the container more efficiently. In the experiments, we test whether this is indeed the case.

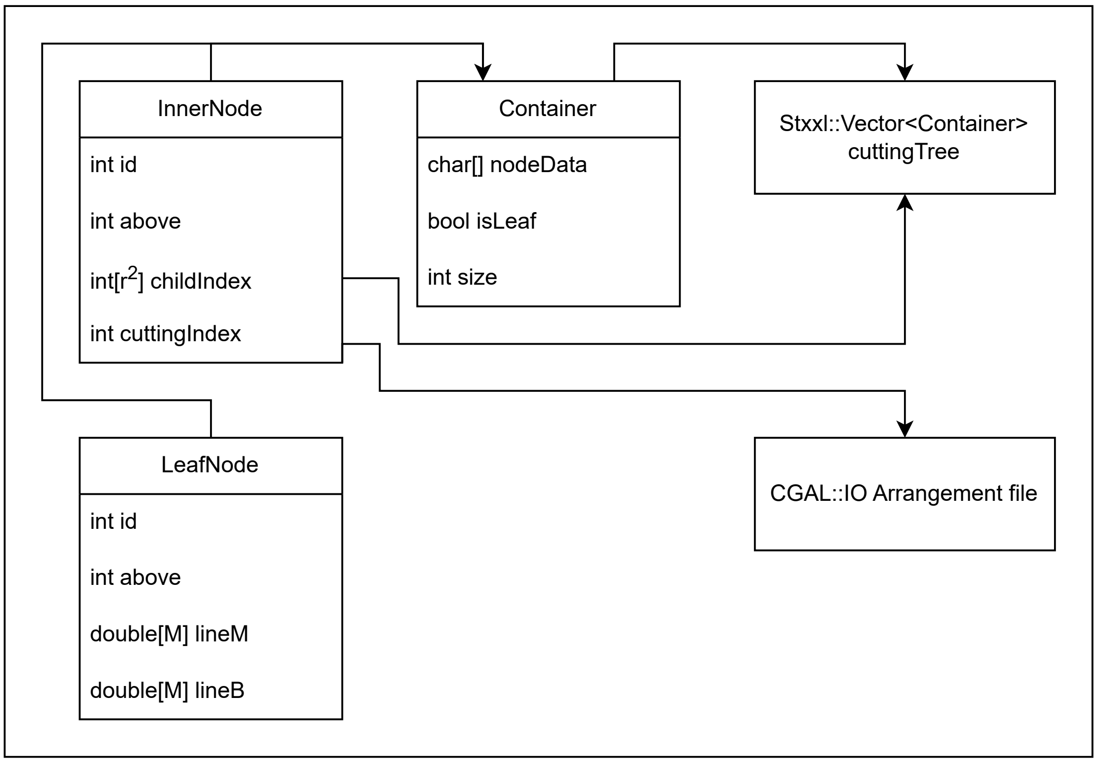


Figure 13: Cutting Tree structure, with separated leaf and inner nodes

5.3 Building the tree

To create the tree structure we recursively call a function that creates nodes of the tree. These nodes are then stored in the aforementioned vector file. The function returns the index of the created node. The node itself is stored in the vector file.

To create a node we first check whether the number of conflicting lines L is above limit M, if it is below M we create a leaf node. To create an inner node we first build a cutting by sampling L and triangulating the CGAL arrangement. For each cell in the cutting, we then check each line-cell pair for intersections. When an arrangement is found that is unbalanced, so there is a cell with too many intersections, we restart this process.

After the line-cell intersection, we have a cutting, a list of the number of lines above each cell of the cutting and for each cell in the cutting the set of lines intersecting it. For each cell, we then create a new node using recursive calls and store its index in the parent node. Finally, if all sub-trees are complete we store the original node in the vector file.

5.4 Handling query's

The input for a query is a half-space or line. We then start by connecting a `STXXL::vector` to the vector file and recovering the root node. The recursive query process is then set in motion. For a given node we need to know if it is a leaf or inner node as we handle them differently. For inner nodes, we recover the corresponding arrangement and do a point-location query to find the cell containing the query point. We then use the subtree index list to recover the next node in the sub-tree and recurse. When we arrive at a leaf node we check for each of the remaining lines, whether that line is above or below the query point. We then add this to the stored value of lines above the subtree and report it.

6 Results

In this section, we will discuss the results of the experiments. The focus of the experiments was to find the bottlenecks in the implementation and to confirm its viability. The experiments were done on a virtual Windows 10 environment. The internal memory was limited to 2 GB and 1 CPU was made available.

For the experiments, we do a single run, using a point set of size 5000, generated by the CGAL random point generator. For the experiments, it is unclear how much of the 2 GB of internal memory is used by the Windows operating system. This means that we based the value of M on the set size n and chose $M = 500$. We decided to test a strict resampling limit and a forgiving resampling limit. The strict limit forces a resample when a cell has more than $5 \cdot \frac{n \log_{10}(r)}{r}$ intersections. The forgiving limit is $5 \cdot \frac{n \ln(r)}{r}$. For the free variable r, we have chosen to test values between 10 and 25 at intervals of 5. We apply the experiments to the basic model (Figure 12) and the separated model (Figure 13), in which we store the leaves and inner nodes in different classes.

We gather data on different aspects of the cutting tree structure during the experiments. First, we will look at the cost of building the entire data structure. Then we will discuss the data gathered in the build process of creating individual inner nodes.

To test the query times, we generate 1000 lines using the CGAL random generator. We then apply these queries to the generated trees from before and keep track of their performance.

Lastly, we also did some tests to verify the performance of the cutting creation process. In these tests, we create nine cuttings for the same number of lines and value of r. We can then examine the performance of creating a cutting for varying input sizes and values of r.

6.1 Build data

In this section, we will discuss the results of the total build costs of the cutting tree structures. The main interest of this section is in the time and memory cost of the data structure and where these costs come from.

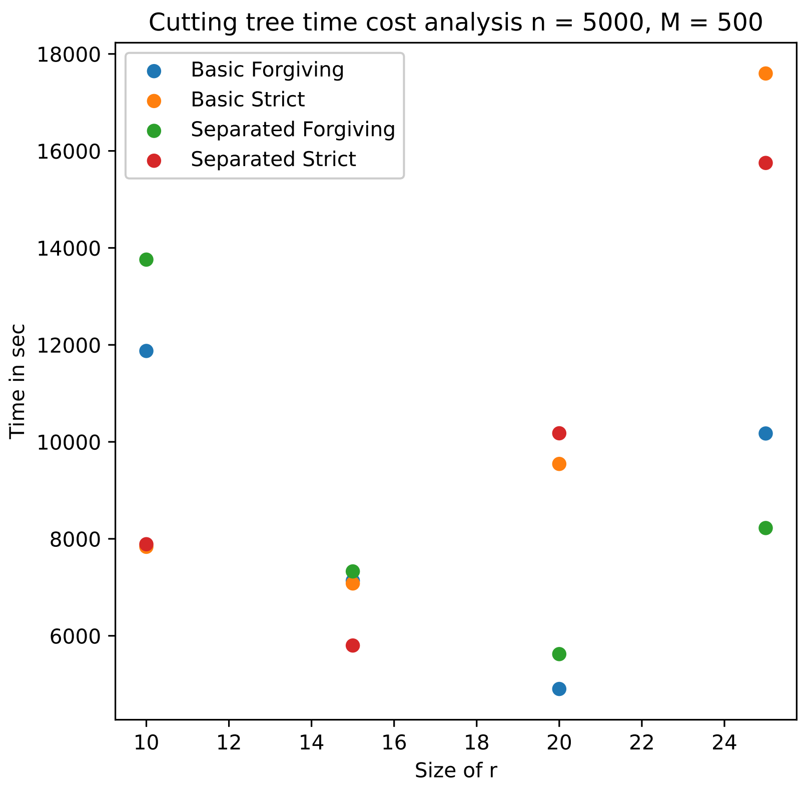


Figure 14: Total time complexity of the building process

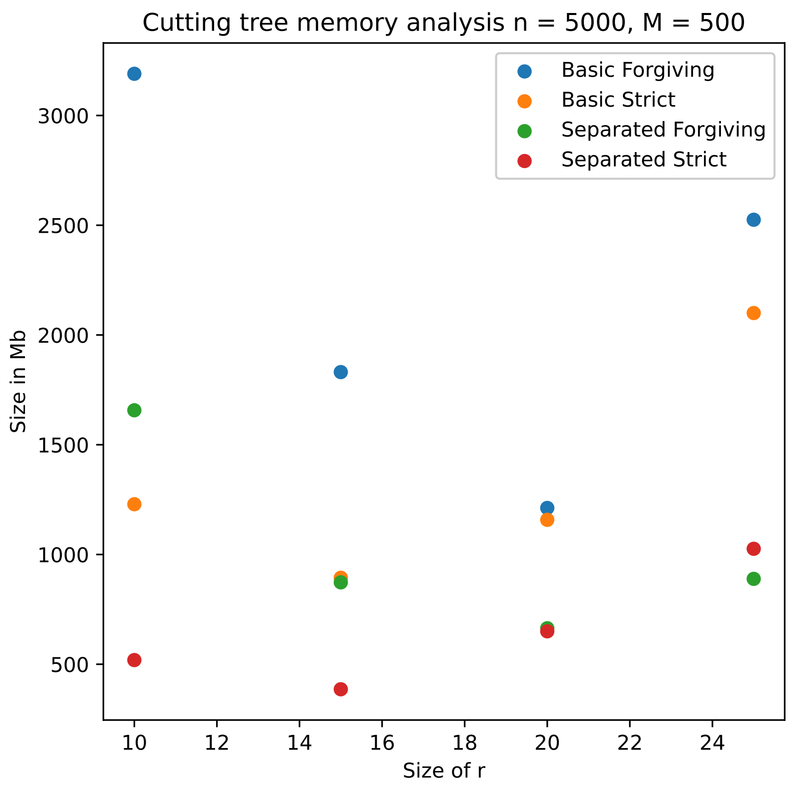


Figure 15: Total memory complexity of the building process

In Figures 14 and 15 we observe that making the sample size r bigger has diminishing returns, with the best performance being around $r = 20$. Another observation is that for the building time, the strict method starts out with a lower time cost, but becomes more expensive once the sample size grows. For the memory complexity, we observe that the method that separates inner and leaf nodes has a lower memory complexity. We also observe that for the most part, the stricter method costs less memory. To explain these observations we will look at the number of nodes in each of the trees and how the memory cost is distributed between node data and arrangement data.

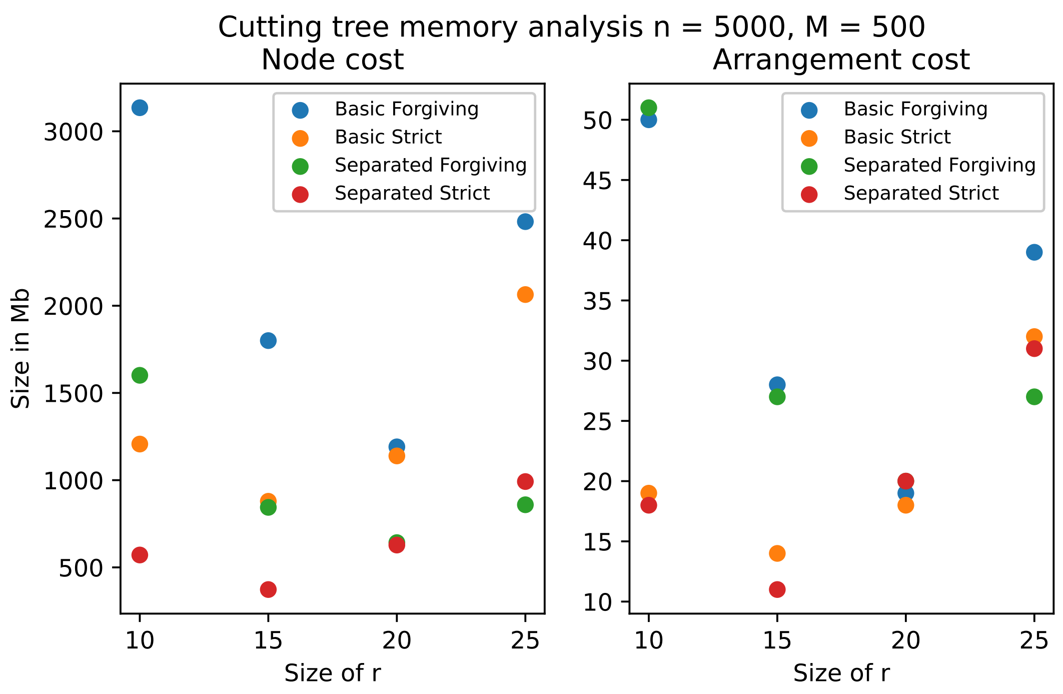


Figure 16: Distribution of memory space between arrangement and node data

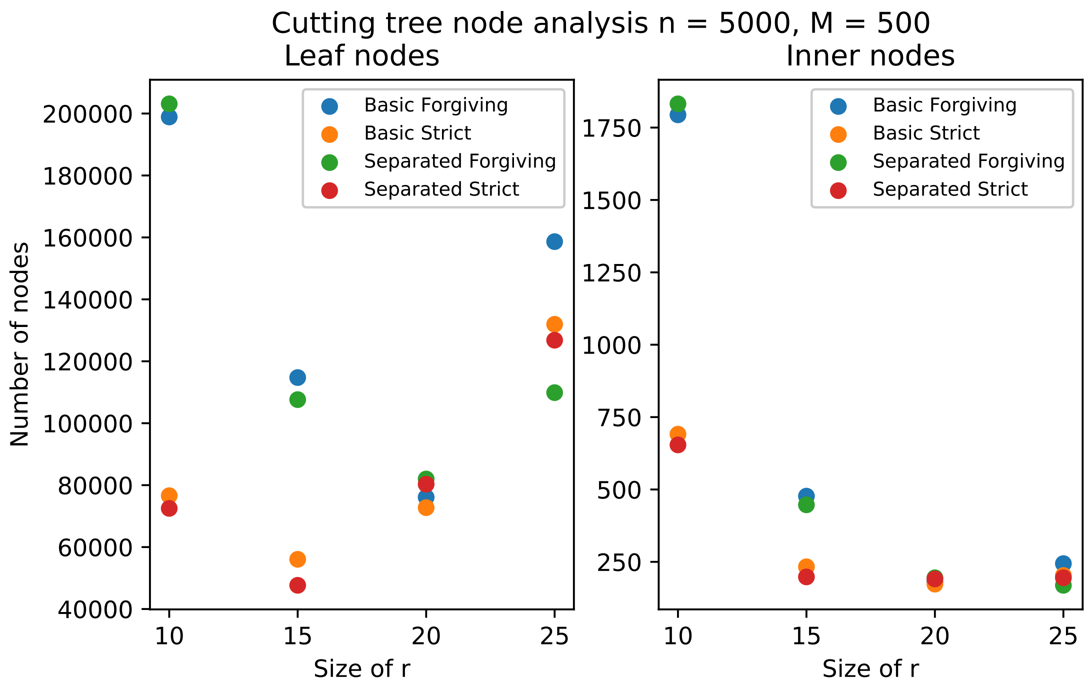


Figure 17: The distribution of nodes in the tree structures

From Figures 16 and 17 we can gain insight into the observations on the general time and memory complexity. For the memory cost it is evident that the total cost of storing all the nodes is the bottleneck. This is explained by the fact that for each tree we create a 100 times more leaf nodes, than inner nodes. This means that the number of leaf nodes is directly correlated to the memory cost. This also explains why the stricter model seems to have a lower memory cost, it divides the lines more evenly and thus needs fewer nodes. For a better understanding of the time complexity, we will need the data gathered on the build process of individual nodes in Section 6.2.

6.2 Inner node data

The focus of this section is on the data gathered during the build process of the inner nodes. We compare the time it takes to build the cutting to the time it takes to handle the line-cell intersection. We are also interested in the number of iterations needed to find a suitable cutting sample and the level of each inner node.

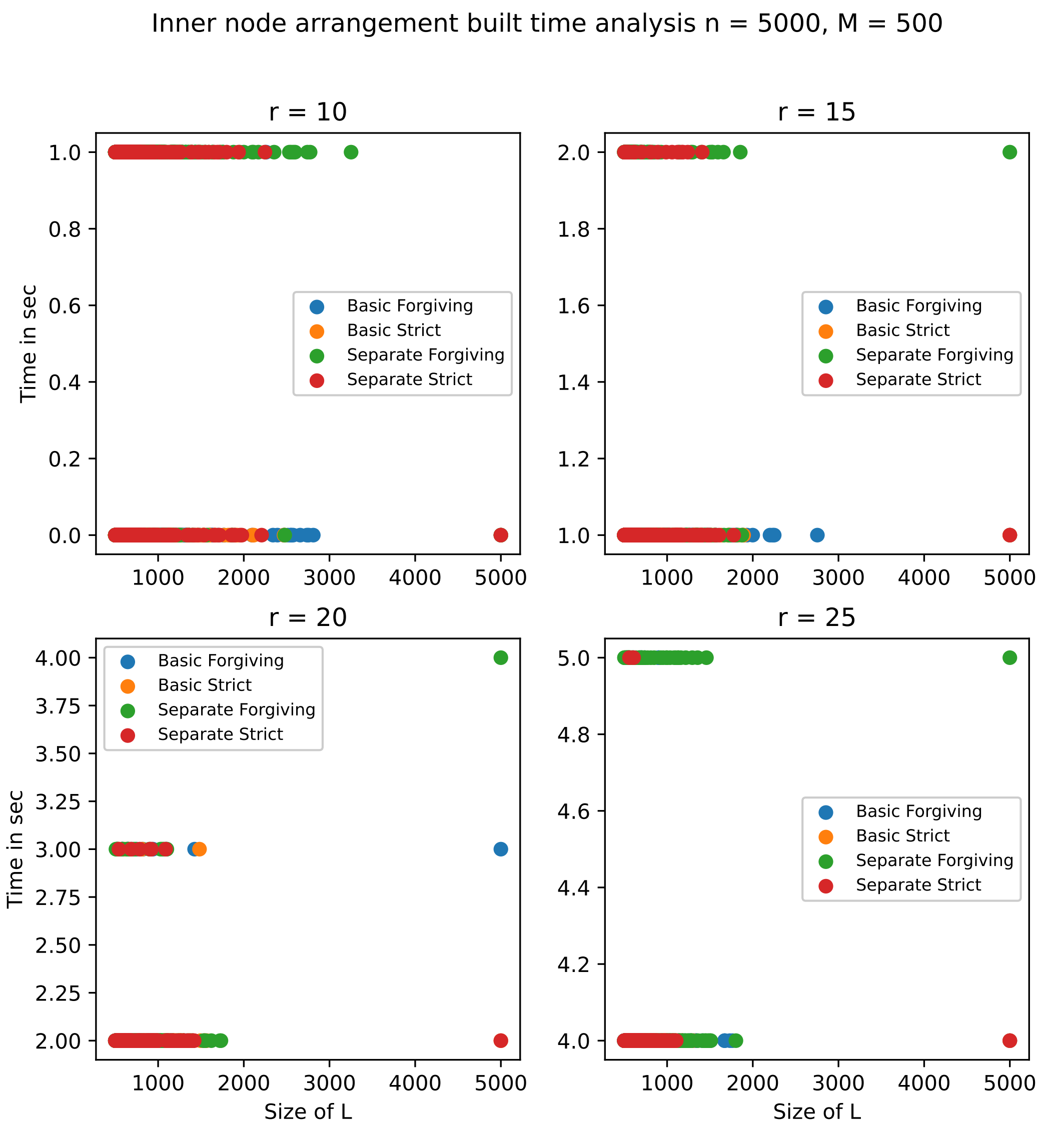


Figure 18: The time cost of building a cutting sample, which is only dependent on r as expected

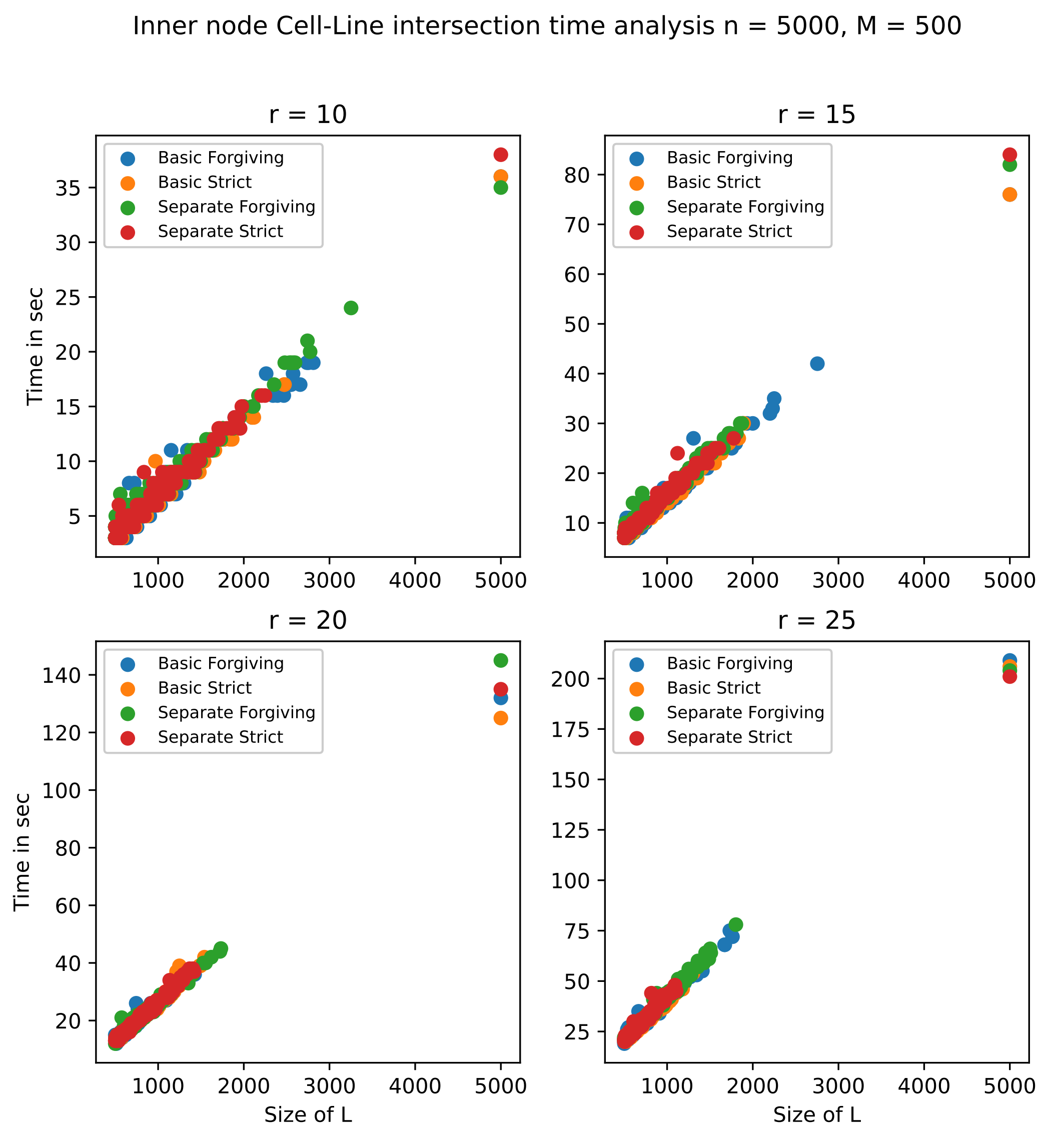


Figure 19: The time cost of handling all line-cell intersections of a cutting

From Figures 18 and 19, we gather that the cost of computing the line-cell intersections greatly overshadows the cost of constructing the arrangement. This is also something that was theorized in the analysis. For the line-cell intersection, we have a linear connection between sample size r and line set size L, this is because we need to check each line-cell pair. The arrangement construction cost is only dependent on the number of created cells r^2.

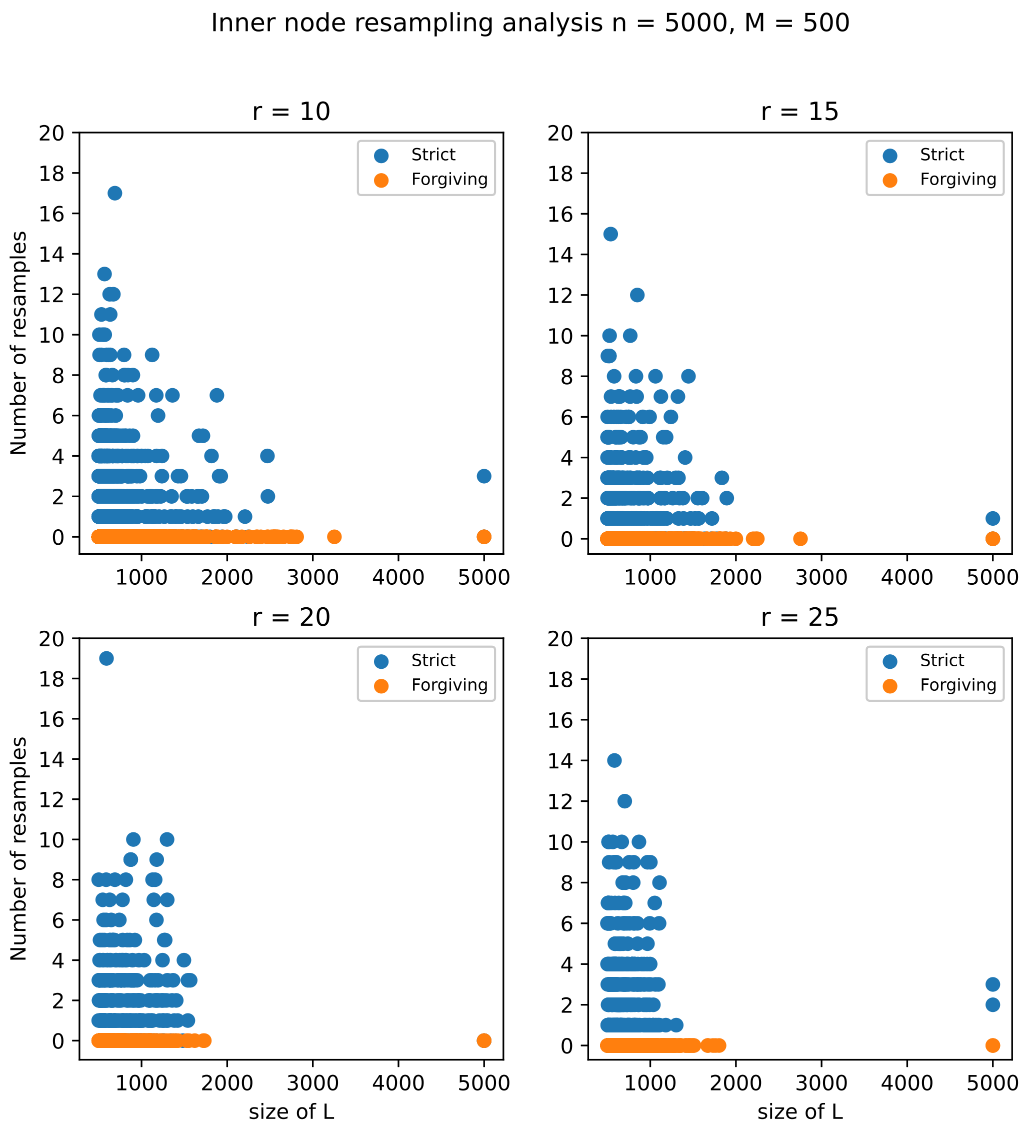


Figure 20: The number of resamples needed to find a valid sample in the strict model

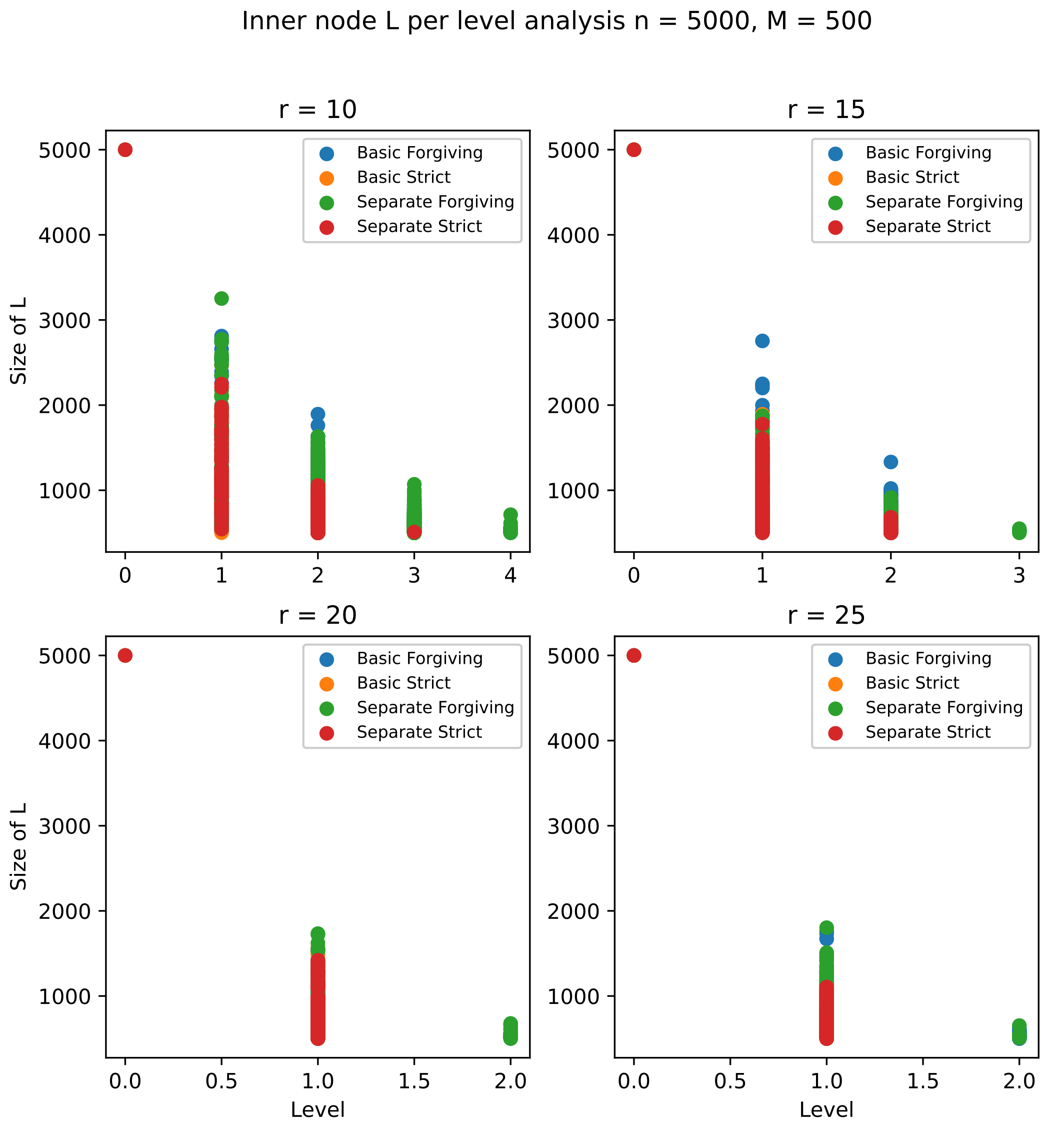


Figure 21: The level of each inner node of the tree

Figures 20 and 21 give us a better insight into the structure of the created trees and the cutting process. What becomes clear from both figures is that an increase in sample size r reduces the number of needed levels, because the lines are more evenly spread out. We can also observe that the stricter model has a lower depth, this again is caused by a better spread of the lines over the cells in the created cuttings. From Figure 20 we can gather that the amount of iterations needed to find a valid sample in the strict method is random, which is logical due to the random nature of picking a sample. The forgiving method accepts each cutting and thus has no resamples.

6.3 Query data

In this section, we focus on the total query time compared to the naive approach of checking for each point individually whether it lies in the query half-plane or not. We will also look into what part of the query takes the most time. Finally, we will investigate the distribution of query times in a histogram.

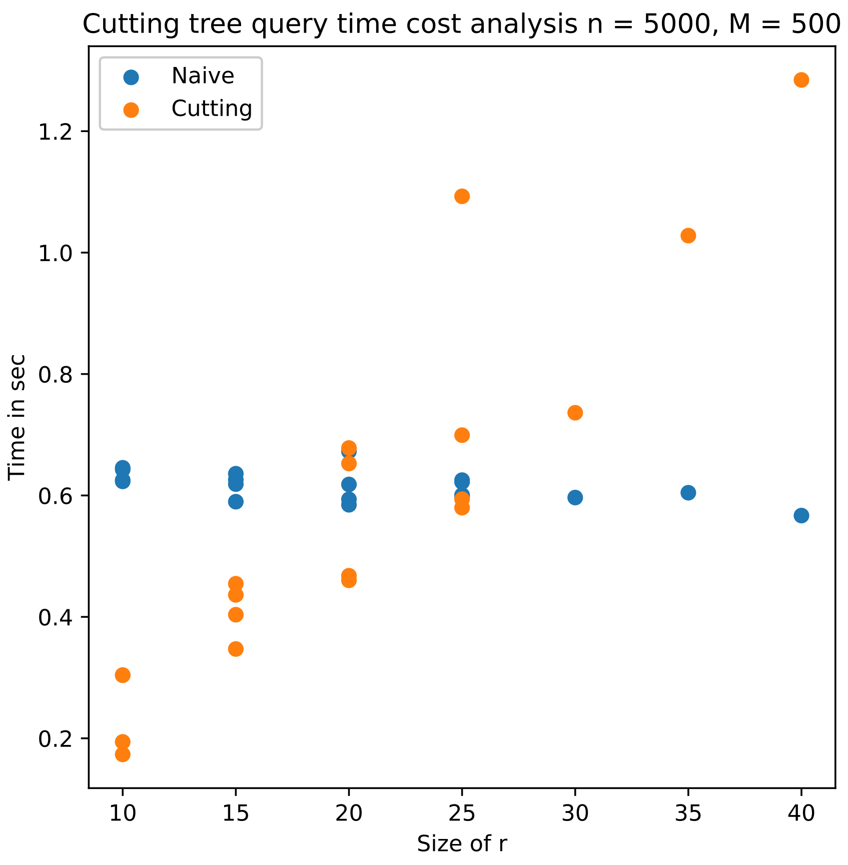


Figure 22: The average total query time over a 1000 queries

In Figure 22 we observe that the query of the cutting tree is faster for some values of r than that of the naive approach of checking each point. We expect that this difference will only grow with a bigger point set N. We do however see that for the bigger sample sizes, the query time grows linearly with the sample size r. This is caused by having to read a bigger arrangement from memory if the sample is bigger. This means that to optimise the query time we want r to be as small as possible.

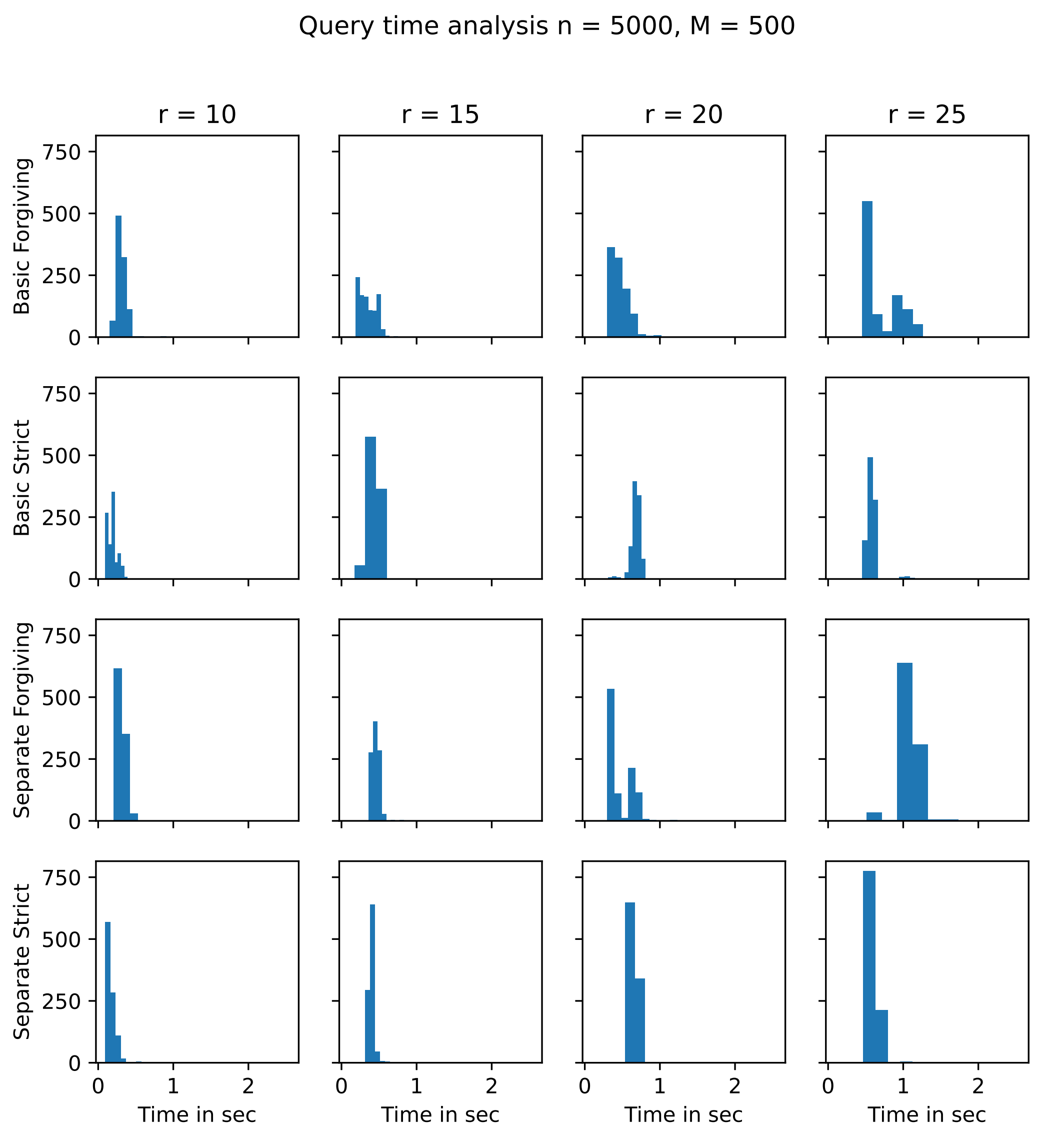


Figure 23: Query times of each query on each tree

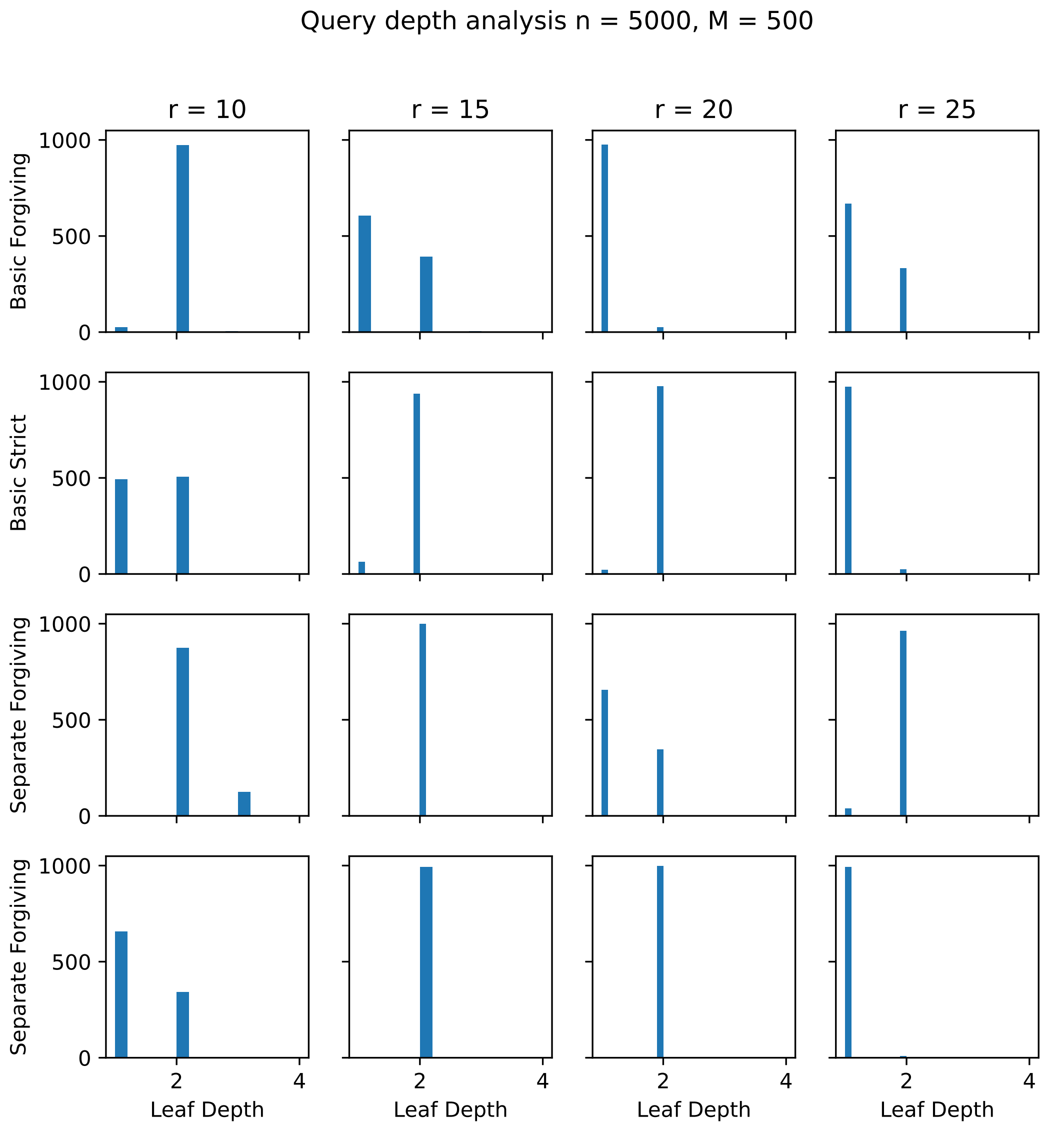


Figure 24: Depth of the leaf of each query on each tree

In Figures 23 and 24 we see a difference between the strict and forgiving methods. The stricter method is faster in general. This is mostly because the strict trees distribute the lines better over the cells, which lowers the depth of the tree. Since the primary cost of the queries is the reading of the cuttings, it is evident fewer levels mean fewer cuttings to read and thus faster queries. However, for some values of r the forgiving method is better. This means that for some values of r the forgiving method is good enough at spreading the data. The stricter method in these cases creates a data structure of similar depth while having to resample in the building process.

6.4 Cutting construction data

In this section, we investigate the distribution of conflicting lines in cuttings. This will give us insight into how well the lines are spread out over the cells of the cutting. The histograms show how many cells of the cutting have each number of intersections. The x-axis shows the number of intersections observed and the y-axis shows how many times each interaction level has been observed. We do nine runs with the same variable setup.

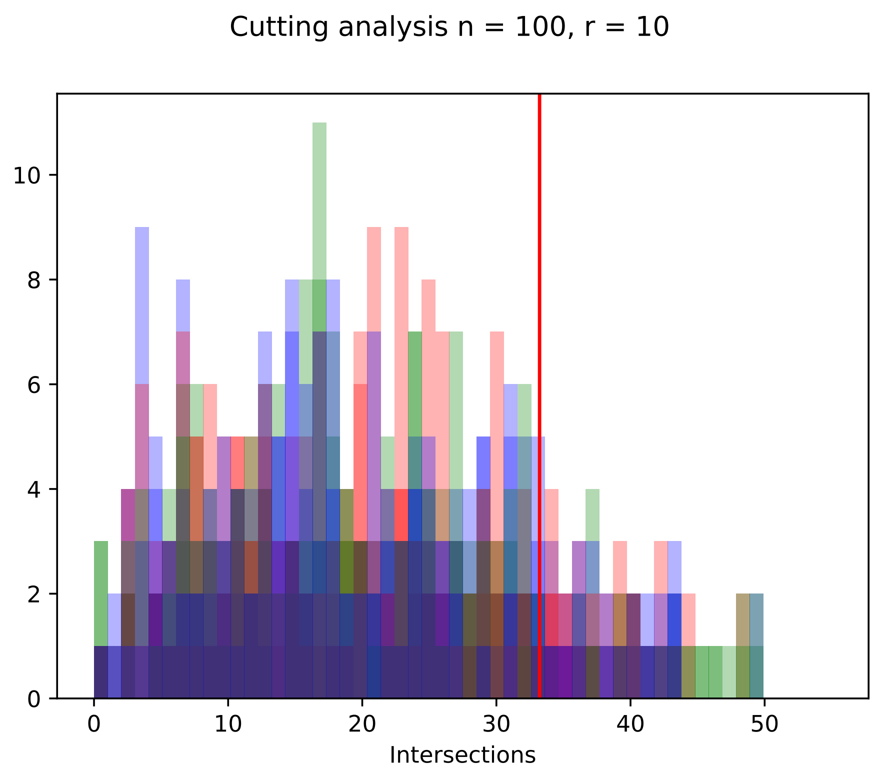


Figure 25: Histograms of the number of lines that intersect each cell of the cutting, where the red line signifies the limit $\frac{n \log_2(r)}{r}$

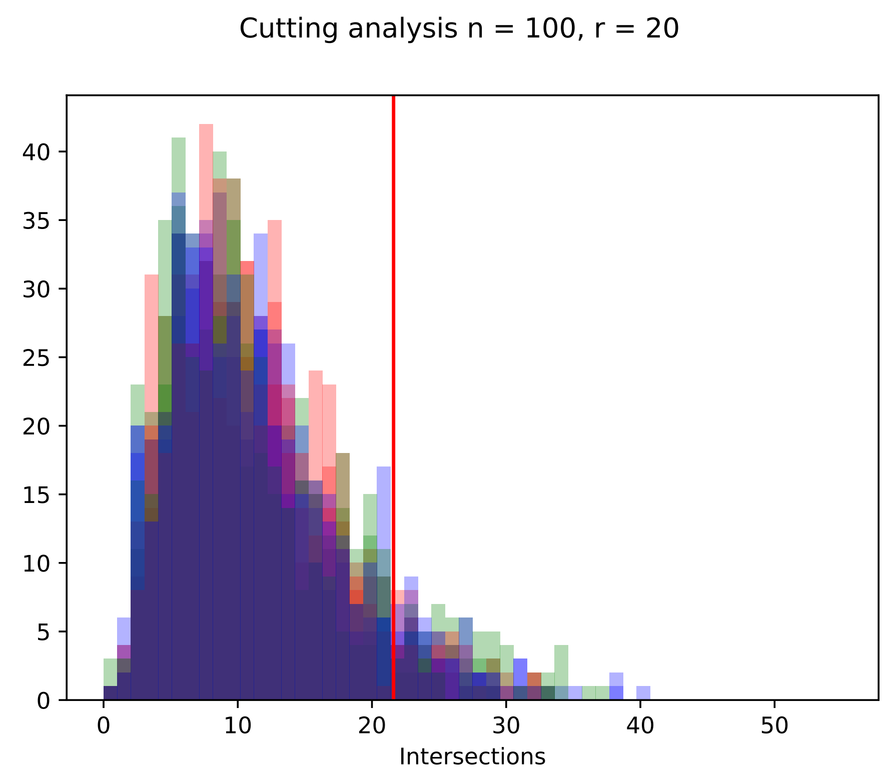


Figure 26: Histograms of the number of lines that intersect each cell of the cutting, where the red line signifies the limit $\frac{n \log_2(r)}{r}$

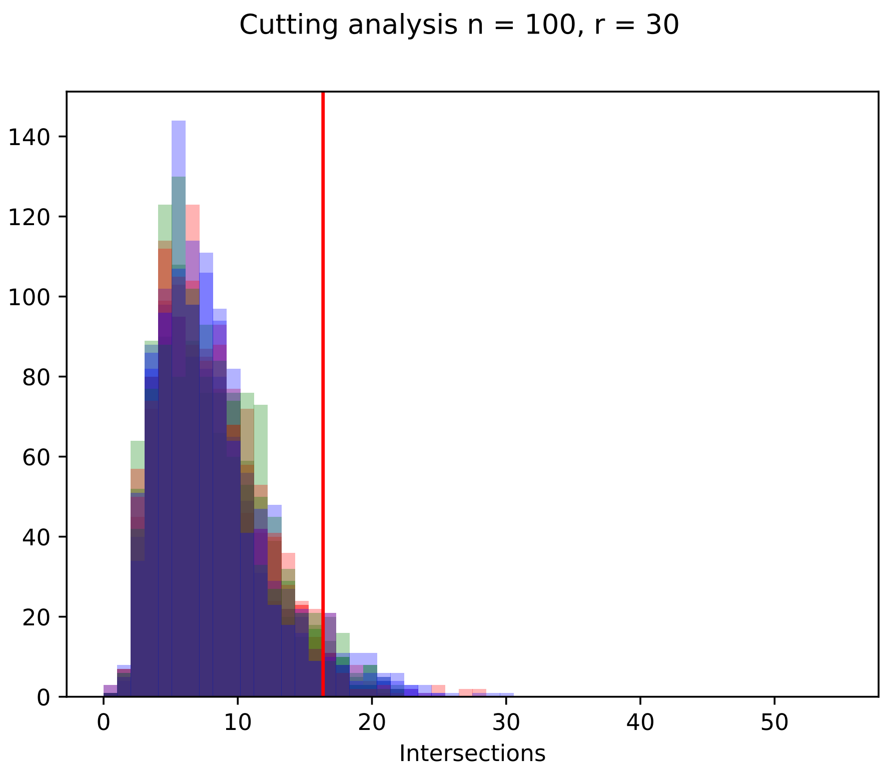


Figure 27: Histograms of the number of lines that intersect each cell of the cutting, where the red line signifies the limit $\frac{n \log_2(r)}{r}$

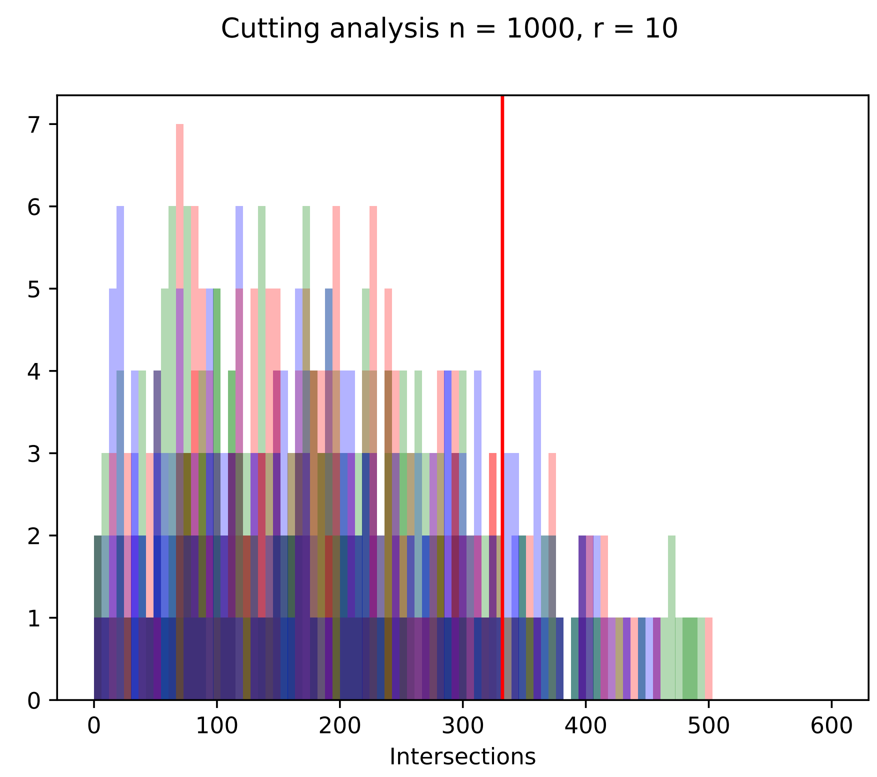


Figure 28: Histograms of the number of lines that intersect each cell of the cutting, where the red line signifies the limit $\frac{n \log_2(r)}{r}$

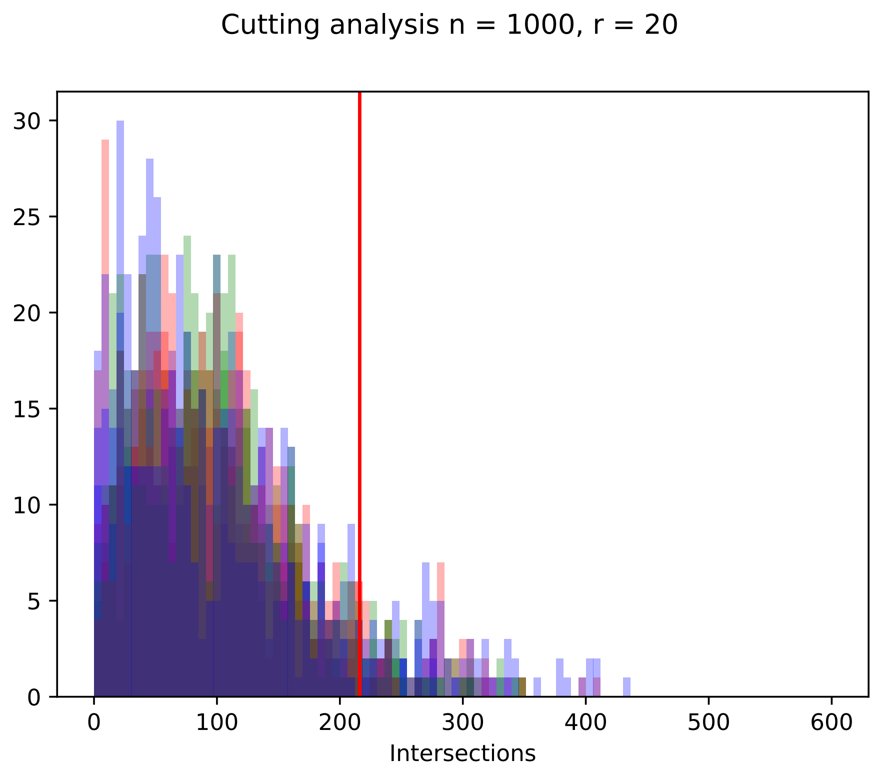


Figure 29: Histograms of the number of lines that intersect each cell of the cutting, where the red line signifies the limit $\frac{n \log_2(r)}{r}$

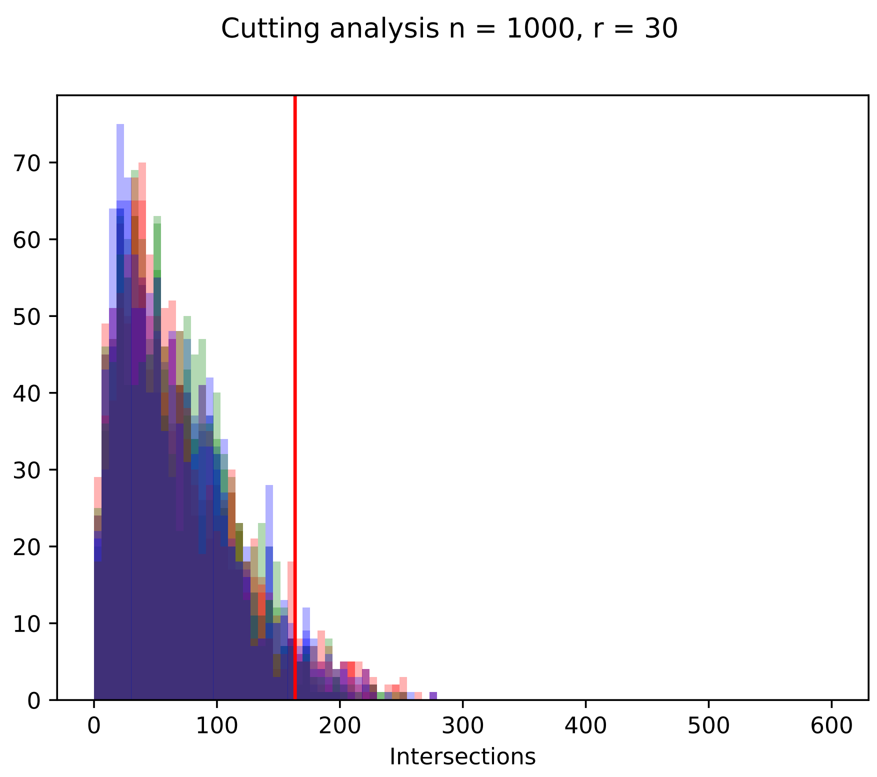


Figure 30: Histograms of the number of lines that intersect each cell of the cutting, where the red line signifies the limit $\frac{n \log_2(r)}{r}$

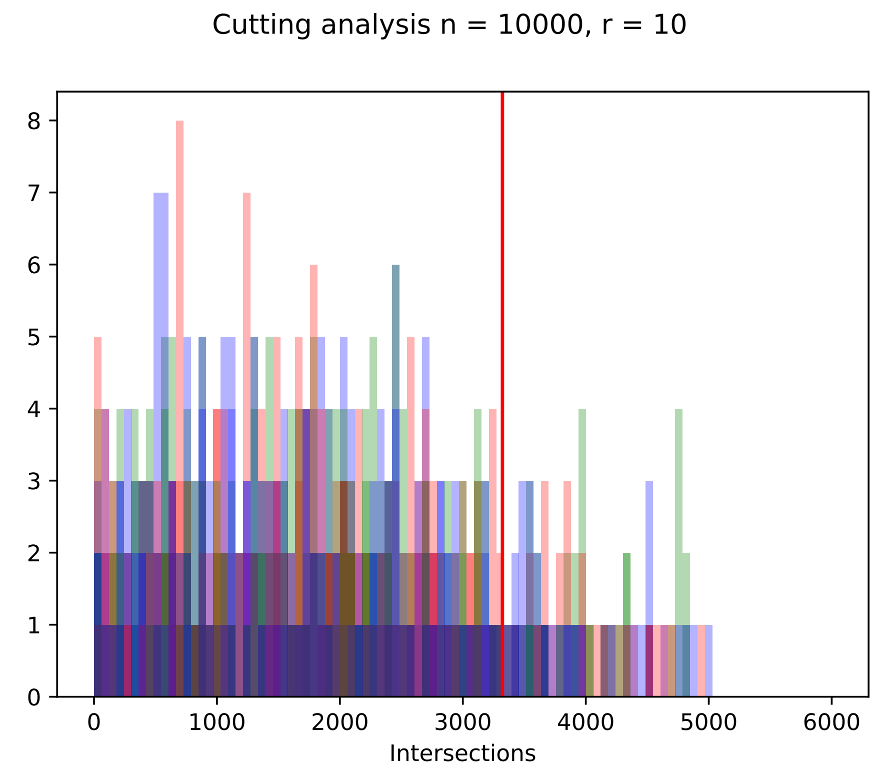


Figure 31: Histograms of the number of lines that intersect each cell of the cutting, where the red line signifies the limit $\frac{n \log_2(r)}{r}$

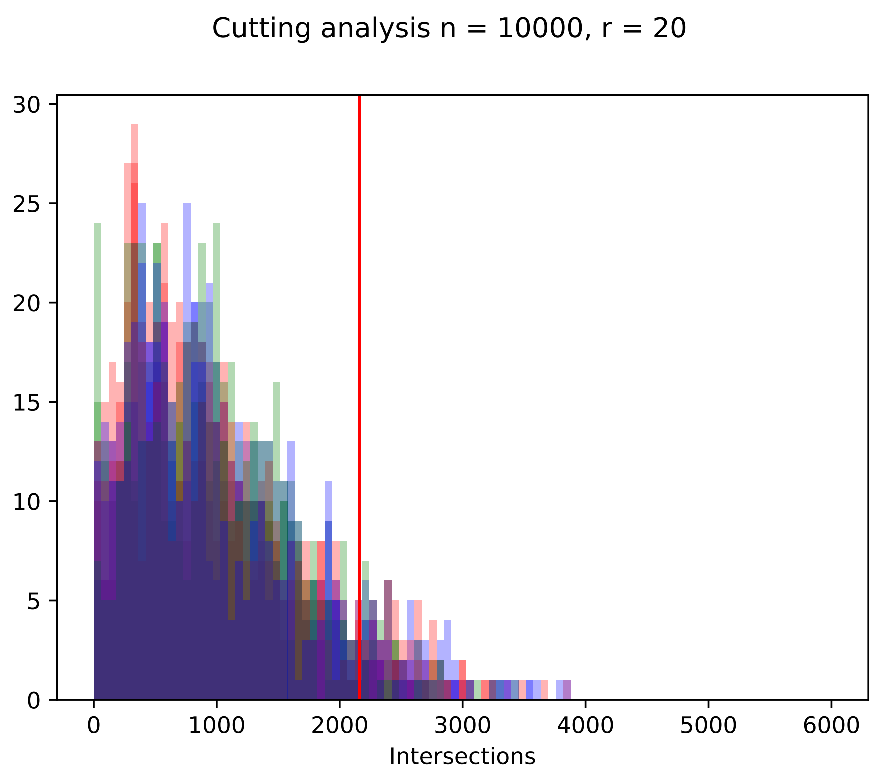


Figure 32: Histograms of the number of lines that intersect each cell of the cutting, where the red line signifies the limit $\frac{n \log_2(r)}{r}$

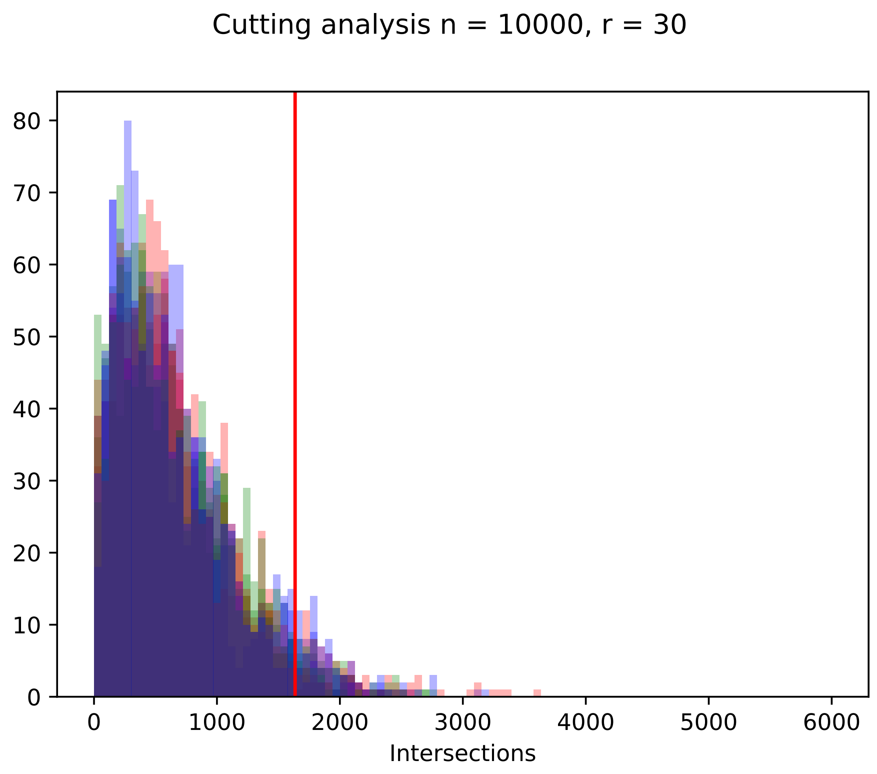


Figure 33: Histograms of the number of lines that intersect each cell of the cutting, where the red line signifies the limit $\frac{n \log_2(r)}{r}$

What we find in Figures 25 to 33, is that the larger a sample size r gets the better it distributes the lines over the cells. We observe this in the fact that the conflict sizes remain closer together for larger sample sizes. Also, the cell with the maximum number of intersecting lines decreases. We also observe that for each point set size n, the histograms are visually similar, so the exact set N has no impact on the cutting quality. Lastly, we notice that each constructed cutting exceeds $\frac{n \log_2(r)}{r}$ intersections. The theoretical analysis from Clarkson [12], uses the big-O notation, so we have some leeway. We do observe that the number of cells that exceed the limit is relatively small.

7 Conclusion

In this master thesis, we have summarized the research into several range searching algorithms and I/O efficiency problems. Using these techniques, we have analysed the cutting tree from the perspective of storing the data structure on external memory. This is the first time this data structure has been studied from this perspective. In the analysis, we have found that the external memory-cutting tree has an I/O building cost of $O(n^{2+\varepsilon}/M)$ and an I/O query cost of $O(\frac{r^2}{B}\log_r(n)/BM) + M/B)$, where $r < M/B - \alpha$, where $\alpha > 1$ is a small constant. However, I have limited the analysis to the half-space range counting problem, while ideally a cutting tree is used for simplex range-searching purposes. The analysis result shows that the construction and storage process of the cutting tree is more efficient by a factor of M. Our external memory cutting tree stores multiple lines in the leaf nodes, which means that we can eliminate the need for the lowest levels of the tree structure. This is significant since each lower level costs more to construct than each previous level combined.

Apart from the I/O efficiency analysis, I have also implemented a prototype of a cutting tree, something that has not been documented before to our knowledge. This has allowed us to gain new insight into the workings of the cutting tree in a practical setting. The biggest contributor to the large memory cost of the algorithm is the number of leaf nodes that are created. For the query time, the main bottleneck is the cost of loading the arrangements from an external file. This means that in this implementation the lowest possible value of r gives the best query time. For the construction process, we observe diminishing returns in the chosen value of r. It is clear that a bigger sample size divides the data better and thus has a lower construction time and memory cost. However, there is a tipping point where the structure can not get more efficient. Any r bigger than that tipping point creates a cutting tree structure with unnecessary nodes, thus increasing the construction time and memory cost with no benefit. For the resampling limit, we observe the same diminishing returns phenomenon for similar reasons. At some point, a more forgiving limit will create a cutting tree of a similar quality to the stricter method. Any r bigger than that tipping point will perform unnecessary resamples, which increases the build time and memory cost.

For further research, it would be interesting to analyse how we can improve the I/O cutting tree and apply it to simplex range searching. The first logical improvement would be using a more sophisticated point-location algorithm. Both the theoretical and practical analyses show that this is a major bottleneck in the current model.

Another improvement can be found in the creation and calculation process of a cutting. In the current model, we need to resample when we find a cell that has too many intersections. It would be interesting to investigate how much the data structure would improve if we were to use the sampling method proposed by Chazelle [10].

References

[1] Pankaj K Agarwal. “Range searching”. In: *Handbook of discrete and computational geometry*. Chapman and Hall/CRC, 2017, pp. 1057–1092.

[2] Pankaj K Agarwal. “Simplex range searching and its variants: A review”. In: *A Journey Through Discrete Mathematics: A Tribute to Jiří Matoušek* (2017), pp. 1–30.

[3] Alok Aggarwal and S Vitter Jeffrey. “The input/output complexity of sorting and related problems”. In: *Communications of the ACM* 31.9 (1988), pp. 1116–1127.

[4] Lars Arge. “The buffer tree: A technique for designing batched external data structures”. In: *Algorithmica* 37 (2003), pp. 1–24.

[5] Rudolf Bayer and Edward McCreight. “Organization and maintenance of large ordered indices”. In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 1970, pp. 107–141.

[6] Jon Louis Bentley. “Multidimensional binary search trees used for associative searching”. In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[7] Jon Louis Bentley. “Multidimensional divide-and-conquer”. In: *Communications of the ACM* 23.4 (1980), pp. 214–229.

[8] Jon Louis Bentley and Jerome H Friedman. “Data structures for range searching”. In: *ACM Computing Surveys (CSUR)* 11.4 (1979), pp. 397–409.

[9] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. ISBN: 3540779736.

[10] Bernard Chazelle. “Cutting hyperplanes for divide-and-conquer”. In: *Discrete & Computational Geometry* 9.2 (1993), pp. 145–158.

[11] Yi-Jen Chiang. “Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep versus plane sweep”. In: *Computational Geometry* 9.4 (1998), pp. 211–236.

[12] Kenneth L Clarkson. “New applications of random sampling in computational geometry”. In: *Discrete & Computational Geometry* 2 (1987), pp. 195–222.

[13] Douglas Comer. “Ubiquitous B-tree”. In: *ACM Computing Surveys (CSUR)* 11.2 (1979), pp. 121–137.

[14] Roman Dementiev, Lutz Kettner, and Peter Sanders. “Stxxl: Standard Template Library for XXL Data Sets”. In: *Algorithms – ESA 2005*. Ed. by Gerth Stølting Brodal and Stefano Leonardi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 640–651. ISBN: 978-3-540-31951-1.

[15] Michael T Goodrich, Jyh-Jong Tsay, Darren Erik Vengroff, and Jeffrey Scott Vitter. “External-memory computational geometry”. In: *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*. IEEE. 1993, pp. 714–723.

[16] Jiří Matoušek. “Construction of ε-nets”. In: *Discrete & Computational Geometry* 5 (1990), pp. 427–448.

[17] Jiří Matoušek. “Efficient partition trees”. In: *Proceedings of the seventh annual symposium on Computational geometry*. 1991, pp. 1–9.

[18] Jiří Matoušek. “Range searching with efficient hierarchical cuttings”. In: *Proceedings of the eighth annual symposium on Computational geometry*. 1992, pp. 276–285.

[19] The CGAL Project. *CGAL User and Reference Manual*. 5.6.1. CGAL Editorial Board, 2024. URL: `https://doc.cgal.org/5.6.1/Manual/packages.html`.

[20] Jeffrey S. Vitter. “Random sampling with a reservoir”. In: *ACM Trans. Math. Softw.* 11.1 (Mar. 1985), pp. 37–57. ISSN: 0098-3500. DOI: `10.1145/3147.3165`. URL: `https://doi.org/10.1145/3147.3165`.

[21] Ron Wein, Eric Berberich, Efi Fogel, Dan Halperin, Michael Hemmer, Oren Salzman, and Baruch Zukerman. “2D Arrangements”. In: *CGAL User and Reference Manual*. 5.6.1. CGAL Editorial Board, 2024. URL: `https://doc.cgal.org/5.6.1/Manual/packages.html#PkgArrangementOnSurface2`.

[22] Dan E Willard. “Polygon retrieval”. In: *SIAM Journal on Computing* 11.1 (1982), pp. 149–165.

[23] Mariette Yvinec. “2D Triangulations”. In: *CGAL User and Reference Manual*. 5.6.1. CGAL Editorial Board, 2024. URL: `https://doc.cgal.org/5.6.1/Manual/packages.html#PkgTriangulation2`.

A Table of variables

N	a set of input points
L	set of lines in the dual space
R	a random sample taken from a set of lines
C	a cutting of a set of lines using a random sample
n	number of points in N
r	number of lines in R
c	number of simplices in C
d	dimensionality or order of a problem
p	probability of success for a Bernoulli trial
L_i	layer/level of a tree-structure
q	a query point/line/shape
A	a 'reservoir' or array used to store the items in the sampling method
Q	the set of query points/lines
K	a set of items to report after a query
M	number of items that can be stored in internal memory
B	number of items that fit into a single memory block