NTNU
Innovation and Creativity

Using machine learning to balance metric trees

Erling Hagen

Master of Science in Informatics
Submission date: June 2006
Supervisor: Magnus Lie Hetland, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Preface

This thesis is written for the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) as part of a Master of Science in informatics program, with associate professor Magnus Lie Hetland as the main adviser. The assignment this thesis tries to answer is:

Metric indexing is one of the fundamental methods in the important area of similarity search. Many tree structures have been proposed for indexing metric spaces, most of which are static. The General Balanced Tree (GBT) method of Arne Andersson (Journal of Algorithms 30, 1999) is designed for index structures on ordered sets, but may be extended to other tree structures as well. This research task for this project is to combine the GBT approach of Andersson with existing static index trees for metric spaces to create new dynamic ones, and to compare these empirically with existing dynamic structures such as the M-tree (Proc. 23rd VLDB, 1997). A prototype implementation will be a central part of the work.

Abstract

The emergence of complex data objects that must to be indexed and accessed in databases has created a need for access methods that are both dynamic and efficient. Lately, metric tree structures have become a popular way of handling this because of the advantages they have compared to traditional methods based on spatial indexing. The most common way to handle indexing is to build tree structures and then prune out branches of the trees during search, and for a dynamic indexing structure it is important that these trees stay balanced in order to keep the worst case search time as low as possible. Normally, this is done based on complex criteria and reshuffling operations. Another way to handle balancing is General Balanced Trees (GBT), proposed by Arne Andersson (Journal of Algorithms 30, 1999), which uses simple, global criteria for rebalancing binary search trees by using total and partial rebuilding. This thesis explores if it is possible to apply this to metric tree structures, and especially two static metric tree structures called the Vantage Point Tree and the Multiple Vantage Point Tree. It discusses how to best make these into dynamic tree structures and how to apply balancing by using GBT paradigms on them. The results of the performance of the new tree structures are analyzed, and the results are compared against already existing structures. The results shows that this works for balancing the trees, and that the structures perform reasonably well compared to already existing structures.

Acknowledgments

First, I would like to thank my main adviser, associate professor Magnus Lie Hetland, who has been the best adviser any student could ever hope for, and given invaluable advice, guidance, feedback and support throughout the process of writing this thesis. I would also like to thank the Faculty of Information Technology, Mathematics and Electronics at NTNU, and especially Femke Driessen, who helped me get this second chance and gave me lots of support, and also the Department of Computer and Information Science at NTNU, including their student tutor Bård Kjos, for giving a lot of advice and support as well. Also, I would like to than Johan Høye who helped me with the proof reading of this thesis, and finally I would like to thank my friends and family for all the support they have given me.

Contents

Chapter 1

Introducion

1.1 Motivation

In database applications similarity search has become more and more common. Search in multimedia databases after sounds, images and video that are similar to a query criteria, searching for similar text strings and searches for DNA sequences are examples where similarity search is very useful. The nature of similarity search is a bit different from exact search in that it is more difficult to prune branches from the trees. This calls for special algorithms and structures designed for this.

The emergence of multimedia (MM) databases also require different qualities from normal database systems. Because calculating the similarity between different multimedia objects usually is much more expensive than with traditional database objects, it is very important to reduce the number of unnecessary distance computations to a minimum. And because fast retrieval of MM database objects is becoming increasingly important in several fields, for example in medicine and science, and finding ways to speed up the search by reducing the number of distance computations is therefore an area well worth investigating.

While traditional databases uses the B^+ [BM72] tree for searching, none of the versions of the B^+ trees are very good when dealing with multidimensional data. Versions of the R-tree [Gut84] has been used instead to handle the multidimensional data, and has gained a wide acceptance as the standard way of doing this. Sadly, the R-tree and it's variants do not offer very good results when the number of dimensions are high, due to what is called the "curse of dimensionality", which says that as the dimensionality of the of the data objects increases, the cost of retrieving data increases dramatically. Metric trees have been proposed as another way to deal with this. Unlike R-trees, which partitions the data in multidimensional rectangles based on the position of the objects in the indexing space, the metric trees uses the distance between the data objects to index them instead. Several of the

structures proposed for indexing the data in metric trees have greatly outperformed the R-tree and its variations, and are therefore very interesting alternatives for use in databases.

Traditional metric trees have been usually been static, which means that the whole data set has to be given in advance, and therefore require a complete rebuild once changes are made to the data set. This is a problem, as many applications require dynamic insertion and deletions on a frequent basis without having to wait for a long time while the indexing trees are rebuilt from scratch. To counter this, several dynamic structures have been proposed as well, like the M-tree and its family, but they are generally outperformed by the static structures when it gets to build- and query time.

To improve query time, it is important that the trees stay balanced so that the worst case number of node accesses needed are kept to a minimum. Most dynamic metric trees uses complex rules to stay dynamically balanced, but even then, they still perform much worse than a complete rebuild.

In [And99], Andersson proposed a way to rebalance binary search trees without the need for complicated rules by using partial and complete rebuilding to balance after insertions and deletions have made the tree too unbalanced. This is done by using only very simple global criteria to determine when the partial rebuilds are to be done, and are shown to be able to outperform the existing complex structures making the trees stay dynamically balanced.

The question we try to answer in this thesis is if these simple criteria can for deciding when to be rebuild the trees can be used to dynamically balance what is originally static metric structures and see if they can compete with the existing dynamic structures. If they do, they could be a powerful alternative to the currently existing ways to handle dynamic metric trees. Well known structures are implemented and tested so that comparisons are easy to other methods and research done. While more work should be done into this problem, this thesis should give a good indication of whether this theory really is worth further investigation.

1.2 Objectives

The objectives of this thesis is to find out whether dynamical balancing of metric trees with the General Balanced Trees method given in [And99] is a promising idea that should be explored further. This is done by implementing and testing the method on two popular static metric tree structures: The VP-tree and the MVP-tree. The results are then compare them with the M-tree, another popular and also dynamic metric tree structure.

While none of these trees in themselves are considered state of the art anymore, as all of these trees are well-known, these results should give a good indication of what can be expected from this method.

1.3 Contributions

- Proposed two tree structures based on General Balanced Trees combined with the VP-tree and the MVP-tree.
- Discussed how to best change the VP-tree and MVP-tree to work with the General Balanced Trees paradigm.
- Made two typical static metric trees, the VP-tree and the MVP-tree, dynamic, and implemented them.
- Tested different parameters with the VP-tree and MVP-tree to see how they would best work in a dynamic environment.
- Developed and implemented a delete algorithm for the M-tree based on the delete algorithms for R-trees.
- Added automatic inclusion of nodes for range search in MVP-trees and M-trees to make them more suitable for range searches with a large range, and tested how much it improves range searches.

1.4 Outline

Chapter 2 contains background information on the field of searching in metric spaces, as well as describing what has already been done on in this field, and detailed descriptions about the theoretical foundations of the algorithms used in this project. Chapter 4 contains the details regarding the implementations for this project. This includes more detailed descriptions of the algorithms, specific details done for this project that differs from the general versions, as well as information about the test sets used. Chapter 5 shows the results from the experiments, and is followed by discussions about each of the results. The general discussions over the results, as well as the conclusion for the project, is found in Chapter 6. This Chapter also contains notes about what could be done in future works in the same field. Some additional results for Chapter 5 are found in the appendix.

Chapter 2

Background Information

2.1 Traditional methods and problems

The traditional method for handling data in database systems has been the family of B^+-trees. The B^+ tree [BM72] is an extension of the B-trees [Bay71], where all the data objects are stored in the leaf nodes, while the internal nodes of the tree are just routing objects to the leaf nodes. The B^+ tree is primarily meant for alphanumeric data (i.e. one-dimensional), where the data can be ordered according to some criterion. They are dynamic trees that handle insertions and deletions in a balanced way, and grow in a bottom-up fashion by splitting upwards in the tree when a leaf node is full. It is a very powerful structure when dealing with the right sort of data, and has been the standard for handling this sort of data in database systems for a long time.

The problem with B^+ trees is that they do not perform well on multi-dimensional data where traditional sorting of the objects is difficult. This became clear when developers began to see the need to handle more complex data, like multimedia, geographical and medical data. Several versions of B-trees were proposed to counter this problem, but none of them managed to handle the requirements of the new application areas. To solve this, several different methods were created to deal with this problem.

Out of the new methods proposed, the R-tree and its family were by far the most successful. It was proposed as a way to deal with geometrical data in [Gut84]. The R-tree is a spatial access method where the data is structured in multi-dimensional rectangles, also called Minimum Bounding Rectangles (MBR). Just like the B^+-tree, the R-tree is dynamic and stores all the data objects at leaf level, and grows bottom-up by splitting upwards in the tree when the leaf nodes become full. Since the release of the R-tree, there have been proposed a lot of modified versions of it, some having several notable improvements. The R-tree has become more or less the standard for databases that uses multi-dimensional data and is implemented in many

well-known database applications.

2.1.1 The curse of dimensionality

The curse of dimensionality occurs when data retrieved form index structures have a large number of dimensions. The curse says that as the dimensionality of the vectors increases, the cost of retrieving data increases dramatically. The term was first used by Richard Bellman in [Bel61], and applied to how the volume rapidly increases when adding extra dimensions to a space. In similarity search, this means that as the number of dimensions increases, the number of nodes having to be visited during a search also increases dramatically. This in turn leads to more I/O operations and more distance computations that have to be computed. In other words, as these are the two things we try to avoid the most when searching, this can be a real problem.

Sadly, the R-tree does not scale well with respect to the number of dimensions. This is because of the fact that dimensionality increases the overlapping between intermediate nodes, therefore making pruning much less effective. Additionally, the increased number of dimensions leads to an increase in the storage space needed, which again leads to fewer object being stored in each node and smaller fan-out. If a single feature vector requires more storage space than the disk page can hold, it will be reduced to a linked list. While both these problems are true for all access methods, the R-trees are not good at handling this at all, and for a large number of dimensions the search performs no better than a sequential scan. According to [BKK96], overlap (at least two subnodes have to be accessed for every node) in a R*-tree [NBS90], which has become the norm for testing the capabilities of R-trees, reaches 100% with only 10 dimensions for uniformly distributed points, and with only 6 dimenions on real data. [LJF94] and [NS] shows similar bad results for the R- and R*-tree. As seen in section 2.1.2, this is very far from acceptable. Several new versions of the R-tree has been proposed, with maybe the X-tree [BKK96] being the most successfull according to [YMT96].

2.1.2 Disk I/O

If the database is large enough, the data needs to kept on disk instead of in the main memory, something that brings up several problems. Because reading from disk is a lot slower than reading from RAM, the number of nodes that have to be accessed during the search is suddenly becoming very important, because every node access means at least one I/O operation. This has become more and more important over the years, and most access methods now take this into account. The way to do this is usually by having fixed size and small nodes, and by having a large fan-out so that a lot of objects can be read at the same time.

One problem with search trees is that the data is not accessed in a sequential order, so the read-write head on the disk has to move to find the new data object on the disk. As reading this way is much slower than simply reading the same data from the disk sequentially, the question is where the threshold goes for when just reading all the data from a sequential scan is faster than accessing just some of the data in a fragmented order. [BGRS99] takes up the question for when a k-Nearest Neighbor (KNN, see Section 2.5) search is actually useful. While most of the study is about when KNN-search will actually result in good matches or not, which is not really the focus here, they also take up how many node accesses should be made before a sequential scan is a better alternative. According to them, when the number of dimensions is 10 or above, a linear scan easily beats complicated indexing structures. They do a closer study of this in [USB98]. It should be noted that this does not take into account that the distance function can be very heavy to compute, and in some applications this could be so significant that the matter of I/O becomes secondary. In those cases, linear scan could still perform much worse than complicated indexing structures, because a linear scan will still have to compute the distance to every object, even if the time used for disk access to find them is smaller.

2.2 Metric space

Formally, a metric space is a pair, $M = (D, d)$ where D is a domain of feature values that act as the indexing keys, and d is the total distance function. The distance function$d : D \times D \mapsto \Re$ in a metric space must have these properties:

Symmetry: $\forall x, y \in D, d(x, y) = d(y, x)$

Positiveness: $\forall x, y \in D, d(x, y) > 0 | x \neq y$

Reflexivity: $\forall x \in D, d(x, x) = 0$

Triangle inequality: $\forall x, y, z \in D, d(x, y) \leq d(x, z) + d(z, y)$

There are several variations of metric spaces, and [PZB06] list a few. One is pseudo-metric, where the positiveness-requirement does not hold. But as long as the triangle inequality holds, pseudo-metric spaces can be considered metric spaces when objects that have a distance of zero between them are considered to be one single object. Quasi-metric spaces are spaces where the symmetry does not hold, but there are ways to transform asymmetric spaces into symmetric spaces. And finally, the super-metric, or ultra-metric, is when the triangle inequality is a bit more strict, defined as $\forall x, y, z \in D, d(x, z) \leq max\{d(x, y), d(y, z)\}$. This means that at least two of d(x,y), d(x,z) and d(y,z) are the same, and these are used quite a bit in biology.

2.3 Distance Function

Unlike for the traditional spatial structures, where the cost of computing a distance is usually very low, computing the distances in metric structures could be much more computationally heavy because of the potentially added complexity given by the requirements for the distance function in Section 2.2.

While any distance function will that satisfies the requirements can be used, there are several types of distance functions that are commonly used. Below is a brief description of them. See [PZB06] for a detailed descriptions of each.

Minkowski: A family called the L_p metrics, and is defined

$$L_p[(x_1, ..., x_n)(y_1, ..., y_n)] = \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p}$$

L_2 is the Euclidean distance, and used for many of the tests in this thesis. Has linear time complexity.

Quadratic Form Distance: From [HSE+95]. Can be adapted to more specific situations than the L_p metrics, like color histograms. Defined by the function $d_M(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y})^T \cdot M \cdot (\vec{x} - \vec{y})}$. Quite a bit slower than the L_p metrics because of the matrix multiplications that have to be done.

Edit Distance: Also called the Levenshtein distance, and used to check the difference between strings, which can be useful for things like spell checking and speech recognition. The general algorithm uses different weights for how many insertions, deletions and substitutions that have to be made to create two similar strings. However, for the function to be metric, the weight for insertion and deletion have to be the same, or the requirement of symmetry (Section 2.2) will be violated.

Tree Edit Distance: Used to find the similarity between different tree structures. The algorithm has $O(n^4)$ time complexity, and hence is quite slow. In the same way as the Edit Distance computes the minimum cost of transforming one string into another, the Tree Edit Distance tries to find the minimal cost of transforming one graph into the other, and counting the changes that have to be made.

Jaccard's Coefficient: Measurement of asymmetric information on binary and non-binary data. For binary data, the formula is

$$S_{ij} = \frac{p}{p + q + r}$$

where p is the number of variables that are positive for both objects, q the number of objects positive in i and not in j, and r the opposite.

More interestingly, for non-binary data the similarity can be computed using set relations by the formula

$$S_{AB} = \frac{|A \cap B|}{A \cup B|}$$

.

Hausdorff Distance: The Hausdorff distance between two given objects $A = \{a_1, ..., a_n\}$ and $B = \{b_1, ..., b_n\}$ is defined as

$$d(A, B) = max\{max_{a \in A} min_{b \in B} |b - a|, max_{b \in B} min_{a \in A} |a - b|\}$$

Named after Felix Hausdorff (1868-1942). Is more general than Jaccard's Coefficient, because objects don't need to be in or out. More details can be found in [Rot91].

Distance functions can be either discrete or continuous, and tree structures have been proposed for both. In fact, several tree structures will only work as intended with discrete distance functions, and while most structures proposed for continuous distance functions will work with discrete ones, they probably will not do that as effectively because far more objects will have the same distances. Several structures using both can be found in Section 2.6.

2.4 Indexing Metric Spaces

Indexing metric spaces can be done in one of two ways. The first one is to use distance transformations to map the metric space into an Euclidean space. If this is done, traditional spatial access methods like the R-tree family (section 2.1) can be used. But, as discussed before, traditional spatial access methods stop being effective when the number of dimensions is high. A detailed discussion about how to transform into Euclidean space is beyond this thesis, but it is important to note that several interesting domains it is not possible or cost effective to use a distance transformation.

The other way, and the one that is the focus of this thesis, is using distance-based indexing structures. Because distance-based structures are not bound by the transformation from metric to Euclidean space, many of them can index any metric space, and this method is therefore much more flexible than relying on using spacial methods.

The traditional way to to create a metric indexing structure is by using a tree-structure. There are two main ways to store objects in a tree structure as well, which basically is if the objects should be stored in both the leaves and the internal nodes, or in just in the leaves. Common for both is that

the internal nodes are used as routing objects to the nodes further down in the trees. Storing all the nodes in the leaves generally leads to trees that needs more disc space, because of the increase in nodes, but pruning usually works better on leaf objects, which could reduce the number of distance computations needed.

There two most common methods for constructing a metric tree are ball decomposition and generalized hyperplane partitioning. These were both named in [Uhl91], although ball decomposition was really only a slightly modified version of a method proposed in [BK73]. Generalized Hyperplane (GH) partitioning is more or less identical to bisector trees proposed in [KM83]. Ball decomposition uses the median of the distances to a vantage point in the node to partition the values. Once a vantage point is chosen, the distance to it from every node is computed. The median of the distances is then calculated, and the data set is split in two, where one part contains the objects where the distances are below or equal to the median, and the other where the distances are greater than or equal to the median. This is then done recursively on each subpart, creating a tree structure. Many variants of this exists, and versions with more than one vantage point have been proposed, i.e. in [Yia93].

The generalized hyperplane method picks two or more pivot objects, and then computes the distances from these pivot objects to the rest of the objects in the set. The remaining objects are then partitioned into groups, with the objects closest to the same pivot object in the same group. The algorithm is then used recursively on each of the new sets belonging to the pivot points. The original methods proposed by [KM83] and [Uhl91] used only two pivot objects, but there have been proposed structures using the same principles where more than two pivot objects have been used, i.e. in the M-tree (see Section 2.7.3.

Metric trees have proven to be a very effective way to index data for similarity search([CMN99]), and have shown to scale much better than the spatial methods when the number of dimensions rises. This is not to say that the curse of dimensionality does not apply for metric trees, and none of the metric indexing structures have been immune to this yet.

2.5 Similarity queries in metric spaces

As described in the previous sections, the types of similarity searches in metric spaces, which are the focus of this thesis, are entirely based around the distances between the objects in the metric space. The way a search is done in every metric tree-structure is by computing the distance to the root of the tree, and from there, according to some criteria, proceeds downward in the tree.

The two standard types of similarity search are range search and k-

Nearest Neighbor (KNN) search. A range search specifies a query object and a given range, and then tries to find all the elements in the indexing structure that are within the range of the query object. In other words, it is checking whether $distance(o, q) \leq r$, where range o is the object, q the query object and r the range. If the object is within the range, the object is included in the result of the search. When all data objects in the node are checked, the algorithm then checks if any of the children can include objects that are within the range of the search, and if so, it calls itself recursively on these children. An example of a basic range search algorithm is shown in Algorithm 2.5.1.

Algorithm 2.5.1: $rangeSearch(n : TreeNode, q : QueryObject, r : SearchRange)$ example of a range search algorithm

```
   Data: A node in the tree, the query object and a range for the search
   Result: A list containing the elements within the range of the search
 1 begin
 2     forall do_i ∈ n do //checking the data objects in the node
 3         d_i = distance(do_i, q)
 4         if d_i ≤ r then //data object is within range
 5             results = results + {do_i}
 6         end
 7     end
 8     forall c_j ∈ n do //checking the children of the node
 9         d_j = distance(c_j, q)
10         if d_j ≤ r + c_j.r then //distance is within range of search +
           range of the child
11             rangeSearch(c_j, q, r)
12         end
13     end
14 end
```

K-nearest neighbor search (KNN search) also takes a query object as a parameter, but instead of finding every object within a range, it finds the k objects that are closest to the query object. First it just includes every object until the result list contains k objects. But then, instead of checking the distance between the data objects and the query node against a range, it checks against the object in the result list that is the furthest away from query node. If the new object checked is closer to the query object than the object already in the list that is the furthest from the query object, the new object is included and the other object is removed. If there is a tie, it does not matter which one of them is in the list. Because the object furthest away is changed out with a closer one once one is found, the search radius will probably be constantly reduced as the search progresses. Other than that,

it functions basically the same as the range search, and calls itself down the tree if any of the children could hold potential objects. An example of a basic KNN search algorithm is given in Algorithm 2.5.2. A special version of the KNN search is the nearest neighbor search, which basically KNN search which just sets k to 1, but it is sometimes treated as a stand alone search method. Sometimes, you also give a range with the KNN search to limit the search, because objects far from the query object are not necessarily interesting even if few other objects are closer. An example of a method to improve KNN-search is given in [KS00], where a way to distinguish more significant NNS from less significant ones is proposed.

One modification that can speed up KNN-searching is to implement a priority for which branches to visit first. Because the range decreases when new objects are included in the result, the goal should be to find these objects as fast as possible so that range is decreased faster. This will in turn lead to more pruning, and therefore less cost in distance computations and I/O. A way to do this is to always go after the subtree with the lowest bound first, and from there work upwards until the remaining trees can be pruned out. Sadly, this will not work in many tree structures because they have no knowledge of the lower bounds.

Algorithm 2.5.2: $knnSearch(n : TreeNode, q : QueryObject, k : NuberOfResults)$ example of a k-nearest neighbor algorithm

Data: A node in the tree, the query object and the k number of elements to be included
Result: A list containing the k closest elements

```
1  begin
2      forall do_i ∈ n do //checking the data objects in the node
3          d_i = distance(do_i, q)
4          if ∃r_i|d_i < distance(r_i, q) then //data object is within range
5              results = results − {r_i}
6              results = results + {do_i}
7          end
8      end
9      forall c_j ∈ n do //checking the children of the node
10         d_j = distance(c_j, q)
11         if ∃r_i|d_i < distance(r_i, q) + c_j.r then //a child could be a
           better match
12             knnSearch(c_j, q, r)
13         end
14     end
15 end
```

In this thesis we will only consider algorithms that returns the correct

result from a search. So a range search will always return the correct number of objects that are within range, and a KNN-search will always return the k nearest objects. However, there have also been some research into approximation search algorithms that returns more or less the correct answers. This is mostly done in order to speed up search retrieval and to break the curse of dimensionality. Examples of this are [LCGMW02] and [CP00a].

2.5.1 Policies for avoiding distance computations

Because distance computations can be very expensive, computing any more of these than absolutely necessary should be avoided. Thankfully, when searching in metric spaces, there are several ways to avoid computing unnecessary distance computations by using the properties of the metric space. This is primarily because of the property of distance functions in metric spaces called the triangle inequality (Section 2.2). [PZB06] gives especially four ways of doing this. As you can see from some of them, several of these methods could also be able to reduce the number of disk I/O as well, because, if implemented correctly, they prune out part of the trees that are not interesting.

Object-Pivot Distance Constraint

The first constraint uses precomputed distances between the pivot objects and the children to prune out the children without making any new distance computations. As the distances have to be computed at build-time anyways, this is more or less a free check and can reduce the number of distance computations by quite a bit. The formula behind this is given in Lemma 2.5.1.

Lemma 2.5.1 *Given a metric space $M = (D, d)$, and arbitrary objects $q, p, o \in D$, it is always guaranteed:*

$$|d(q,p) - d(p,o)| \leq d(q,o) \leq d(q,p) \leq d(q,p) + d(p,o)$$

First compute the distance between the query object q and the pivot point p, which is probably already to check if the node should be entered. As noted, the distance between p and the objects node are already computed and stored. Because $|d(q,p) - d(p,o)| \leq d(q,o)$, we know that if $|d(q,o) + d(p,o)| > range$, q can not possibly be close enough to o, because $d(q,o) > range$. Therefore we do not need to compute the distance $d(q,o)$, and one distance computation is saved.

On the other hand, if $d(q,p) + d(o,p) \leq range$ the object can be directly included, because it is certain that $d(q,o) \leq range$, and again, one distance computation is saved.

If none of these requirements are fulfilled, $d(q, o)$ has to be computed, because there is no way of knowing whether the distance is within the range.

While this basic version can only be used on leaves, it can also be extended to work on internal nodes if the maximum range to the leaves below is known, by checking against $range + range(o)$ instead of just $range$.

Range-Pivot Distance Constraint

Because storing the distance to each child is in some occurrences unacceptable because of space requirements, an alternative exists where only the lowest and highest distance between the node and it's children are stored. While a bit weaker than the one in Section 2.5.1, this could save quite a bit of distance computations as well. The formula behind this is showed in Lemma 2.5.2. r_l is the lower bound of r, that is the lowest range from p to any object, and r_h is the highest.

Lemma 2.5.2 *Given a metric space $M = (D, d)$, and objects $o, p \in D$ such that $r_l \leq d(p, o) \leq r_h$, and given some $q \in D$ and an associated distance $d(q, p)$, the distance $d(q, o)$ can be restricted by the range:*

$$max\{d(q, p) - r_h, r_l - d(q, p), 0\} \leq d(q, o) \leq d(q, p) + r_h$$

This works much in the same way as the previous constraint, but now we know the node can be excluded if $max\{d(q, p) - r_h, r_l - d(q, p), 0\} \leq d(q, o) > range$, because we know that if this is true, $d(q, o)$ can not possibly be in range of the search. The 0 is there because no distance can be less than the 0 because of the positiveness requirement of a metric distance function (see Section 2.2).

In the same way, $d(q, p) + r_h > range$ means that the object can be included at once, because there is no way that $d(q, o)$ can be longer than the range of the search.

And as before, if none of these are true, the distance has to be computed to make sure if $d(q, o)$ is within range or not. This also only works on leaves, but can be extended if the range of the furthest data object in the subtree from the internal node is known.

Pivot-Pivot Distance Constraint

The next constraint is a bit weaker than both of the others, but can still be useful. This again only uses the ranges of the nodes, so not all the precomputed distances between nodes and subnodes and objects have to be stored. See Lemma 2.5.3 for the formula this is built on. Here r is the range of the node containing o, while r' is the range of the parent node to p, called p'.

Lemma 2.5.3 *Given a metric space $M = (D, d)$, and objects $o, p, q \in D$ such that $r_l \leq d(p, o) \leq r_h$ and $r'_l \leq d(q, p) \leq r'_h$, the distance $d(q, o)$ can be bounded by the range:*

$$max\{r'_l - r_h, r_l - r'_h, 0\} \leq d(q, o) \leq r_h + r'_h$$

As can be seen, if the maximum of $r'_l - r_h$ and $r_l - r'_h$ is larger than the range of the search, the node can be excluded without doing a distance computation. If the lower bound of r' is larger than the higher bound of r, then when now that $r'_l - r_h$ will be a positive number, and will also be lesser or equal to the $d(q, o)$. This is because of Lemma 2.5.1, which we apply to p in relation to p'. We therefore have an upper and a lower bound of $d(q, p)$ because we have $d(q, p')$, and we also know that $d(q, p) \in [r'_l, r'_h]$. For the same reason, we also know that $d(q, o) \in [r_l, r_h]$, and can from there say that if $r'_l - r_h > range$, o can be pruned without the need for a distance computation. And if r_l instead is larger than r'_h, the same applies for that.

On the other side, if the the sum of both nodes are greater than the range, $r_h + r'_h$, the node can be automatically included. This is because the sum of the upper bounds of the two nodes is the absolute maximum distance for q can be from o, and again this is because Lemma 2.5.1 is applied.

As we can see, not only one, but two distance computations are saved this way, because we do not need to compute the distance to either p or o if any of these criteria are true. If not, however, you will at least need to compute the distance to p, where you could try applying any of the previous lemmas.

Double-Pivot Distance Constraint

The previous three constraints are for speeding up the search for nodes that only use one pivot, like in ball partitioning. This method is, on the other hand, used for generalized hyperplane partitioning. Unlike the other two, here only the lower bound can be defined, and therefore only if the object should be completely pruned out can be determined. The definition behind this is given in Lemma 2.5.4.

Lemma 2.5.4 *Assume a metric space $M = (D, d)$, and objects $o, p_1, p_2 \in D$ such that $d(o, p_1) \leq d(o, p_2)$. Given a query object $q \in D$ and the distances $d(q, p_1)$ and $d(q, p_2)$, the distance $d(q, o)$ is lower-bounded as follows:*

$$max\{\frac{d(q, p_1) - d(q, p_2)}{2}, 0\} \leq d(q, o)$$

As can be seen, the $max\{\frac{d(q,p_1)-d(q,p_2)}{2}, 0\}$ is the lower bound of $d(q, o)$. This is because $d(p_1, q) - d(o, p1_)) \leq d(q, o)$ and $d(o, p_2) - d(p_2, q) \leq d(q, o)$. When added together, this yields $d(p_1, q) - d(p_2, q) + d(o, p_2) - d(o, p_1) \leq$

$2d(g,o)$, which divided on 2 is $\frac{d(q,p_1)-d(q,p_2)}{2} \leq d(q,o)$. Again, a distance can not be less than 0, so therefore the maximum of the answer and 0 used.

One nice thing about this constraint is that it does not need stored information about the distances between the pivots and the pivot objects, as it only needs to know which one it is closest to. But as the object will be a child of the closest pivot in a generalized hyperplane partitioned tree anyways, that is not a problem.

Pivot Filtering

Pivot filtering is a method where combining several pivot objects is used to improve the pruning. The formal definition is given in Lemma 2.5.5. See [Doh04] for a study of how to efficiently use pivot filtering.

Lemma 2.5.5 *Assume a metric space $M = (D,d)$, and a set of pivots $P = p_1,...,p_n$. We define a mapping function $\Psi : (D,d) \rightarrow (\Re^n, L_\infty)$ as follows:*

$$\Psi(o) = (d(o,p_1), d(o,p_2), ..., d(o,p_n))$$

Then, we can bound the distance $d(q,o)$ from below:

$$L_\infty(\Psi(q), \Psi(o)) \leq d(q,o)$$

Ψ in Lemma 2.5.5 is a mapping function that represents the precomputed distances between pivots and objects. Pivot filtering needs the distance between every pivot and the objects stored. But once this is done, the filtering can be applied. The nice thing her is that if only one $|d(q,p_i) - d(p_i,o)| > range$, the object can be excluded. As can be seen, this is simply based on using Lemma 2.5.1 multiple times.

2.6 Previous work

Here we show a brief history in the work of similarity searching in metric spaces. For a more detailed survey of this, [CNBYM01] and [PZB06] are recommended.

While [BK73] was about similarity search in key words in files, and not directly an attempt at indexing metric spaces, several of the methods given there have been adapted later for use in metric spaces. Primarily, three methods where proposed, all for discrete values. The first one was a hierarchical tree structure where objects are put in chunks with other objects at the same distance from some pivot object, which is arbitrarily chosen. It is then called recursively on each of these chunks. This structure was later named BK-trees. The second method partitions the data into a set number of sets of objects. For each set a pivot, or center as it is called here, object is

arbitrary chosen, and the maximum distance from this object to any other object in the set is stored. This is then done recursively on each set. But no information about how to partition the objects into sets is given. The last method relies on finding the maximal cliques at each level in the tree, where each clique is a set that can not have a diameter larger than some max diameter, which is dependent on the level in the tree. Because objects can be in several different clique, the objects which appears in the most number of cliques are chosen as pivot objects.

An attempt at minimizing the number of distance computations was given in [SW90]. This method is meant for situations where the cost of computing the distance between objects is very high, and uses a structure called an Appropriate Distance Map (ADM), which is a $n \times n$ matrix where they compute and store every distance in the data set. While this could be good for small datasets with extreme distance computations cost, it is not doable on large data sets.

In [Uhl91], two indexing structures where introduced. They were called ball decomposition and generalized hyperplane partitioning, and are both described in Section 2.4. [Yia93] gave the ball decomposition tree the name Vantage Point Tree (VP-tree, see Section 2.7.1 for a detailed description), and gave an analysis as well as suggesting ways to improve it, including ways to find better vantage points. [Chi94] contains some more analyzes of the VP-tree on image indexing, gives a few modifications for improvement, and proposes a way to do dynamically adjusting KNN-searches.

A new version based on the KB-trees approach is given in [BYCMW94], where Fixed-Queries Trees (FQ-trees) are proposed. The big difference here is that all pivot objects on each level in the tree are the same, while all the actual data objects are stored in the buckets in the leaves. This means that if many nodes on the same level are visited, only one comparison is needed. The disadvantage is that the trees are a bit higher. The idea of using a single pivot per level has since been used in several other structures. A slightly modified version of this is given in [BY97], where they use a fixed height for the tree instead, that is, all leaf buckets are at the same level in the tree. These trees are called Fixed Height FQTs, or FHFQT. While this makes the path to some objects longer than it needs to be, they claim this can actually improve the query time of the tree because of the one-object-per-level rule. Another version of this principle was proposed in [CMN99], and is called Fixed Queries Arrays (FQAs). This isn not exactly a tree, instead it is more of an array implementation of the FHFQT. This structure reduces the precision of the distances stored, and instead uses more pivot elements.

Back to the time line, Geometric Near-Neighbor Access Trees (GNATs) where introduced in [Bri95]. The indexing here is quite much like the GH-tree, and uses a k number of split points at the top level, where the value of k is decided by the cardinality of the dataset, and then partitions the rest of the data objects to whichever split point it is closes to. The distance to the

nearest and furthest of the objects associated with each split point is stored. This is then done recursively on each set belonging to a split point to build a tree. For searching it uses the triangle inequality to prune out searches by checking against center objects and pruning out those where the bounds exceed the range.

The M-tree was introduced in [PZR96], and later given more specific implementation details in [CPRZ97]. The M-tree is described in detail in Section 2.7.3. The most interesting thing about the M-tree was its dynamic properties, and it has spawned a large number of variations, including the Slim Tree [TTSF00] (described in more detail in Section 2.7.4).

In [BO97], the MVP-tree is introduced. This structure is described in detail in Section 2.7.2. But briefly it makes some improvements over the VP-tree by using previously computed distances to filter out nodes further down in the tree, and by combining two levels in the tree in one node, and by that reducing the number of distance computations needed to the vantage points.

[FsCCM00] explores ways to make the VP-tree dynamic, as well as discussing KNN-search in the VP-tree. They also take some inspiration from the node filtering of the leaves in the MVP-tree. It is tested against the R* and the M-tree, and they get significant improvements for KNN-search for both synthetic clustered data, uniform data and real data.

The M^2-tree is a version of the M-tree, and was presented in [CP00b]. It makes the M-tree able to process complex similarity search over objects represented by multiple features. This makes the M-tree objects able to store several feature vectors, which can help with indexing and search in many types of databases, for example in an image databases where nodes now can store both color distribution and keywords.

An interesting method called a List of Clusters (LC) was proposed in [CN00]. It is very simple, and works the way that it chooses one object from the set of objects, and finds the m closest objects to it and creates a cluster of them. It then does this recursively on the remaining objects until $n/(m+1)$ clusters have been created. During search, it just considers all clusters p and checks if $distance(q, p_i) - r > cr(p_i)$, where $cr(p_i)$ is the covering radius of p_i, q is the query object and r is the range of the search. If that is true, it enters, if not it goes on. More interestingly, if $distance(q, p_i) + r < cr(p_i)$ no other clusters needs to be examined, because all objects that can be within the range of the search has to be in this cluster. The authors show that this method is much better than a lot of existing methods, especially for higher dimensions.

Similarity search through hashing was first proposed in [CGZ01], where the authors propose an access structure called Similarity Hashing (SH). It is designed to reduce I/O costs by having objects in buckets and using precomputed distances to pivots for reducing distance computations. It uses a ρ-split function to group objects from X into $m + 1$ disjoint subsets, and

it is made sure that the distance between objects in two different subsets is at least 2ρ. The last of the subsets is called the exclusion bucket, and the same process is used on the exclusion bucket to form a new level, but not necessarily by using the same ρ-split function, though the same value for ρ has to be used. The result is that it can be guaranteed that only one of the m partitions of X can contain objects with a query range $r_x \leq \rho$. Then, when searching, only one bucket at each level has to be accessed as long as $r \leq \rho$, as well as the exclusion bucket. This means only $h+1$ buckets has to be accessed, where h is the number of levels. The authors then shows that this methods needs a lot less distance computations than the MVP-tree. This idea has later been improved into the D-Index [VDZ03], which also has an extension called the eD-Index [DGZ03], and they have both shown very impressive results.

The M^+ tree, proposed in [XZY03] is another variation of the M-tree, that takes some inspiration from MVP-trees as well. The authors show results where the M^+ has better query performance than the M-tree, but because it uses spatial data to help the filtering of the nodes, it can not be called true metric access structure. It uses the same principle to partition the space as the M-tree does, but then it partitions the newly created sets again, just like the MVP-tree does, according to a key dimension. But unlike the MVP-tree, only the distance to one object needs to be computed because there is no distance computations needed for the key dimension. The authors later proposed the BM^+ tree in [ZWZY05], which is a binary version of the M^+ tree. It uses a rotatable binary hyperplane tree to partition the subspaces instead of using the key dimension. This is shown to be an improvement over the M^+ tree.

[Sko04] proposed the Pivoting M-Tree, which is another variation of the M-tree from [CPRZ97], and which tries to reduce the volume of empty space created by the hyper-spheres used by the M-tree. This is done by creating a list of pivots p_l, computing the distances between them and the routing objects and storing them in an array, and then use pivot filtering to improve the search. As this is created at build time, it can decrease the number of distance computations that have to be done at query time because of improved filtering, but it can significantly increase the disk I/O costs because the pivot table must be read at query time.

2.7 Different tree structures

2.7.1 Vantage Point Tree

The Vantage Point Tree (VP-tree) was introduced in [Uhl91], but under the name ball composition. This is the original ball composition method, and it has spawned several similar variants using the same principles. It is a static metric tree structure that constantly divides the dataset S in two by

finding the median element, called the pivot element, and then dividing the set around it. It then recursively creates the tree by dividing the two parts again creating new nodes until all sets are empty.

There are several ways to find the pivot element. The one proposed by [Uhl91] is simply to select an arbitrary element from S as the pivot element. Another method, presented in [Yia93], is to take a list of samples from S and check which one gives the largest variance when comparing the distances from it to nodes from another list of samples from S. It then sorts the list using the distance to the pivot element as the "sorting value", and stores the median distance in the node. It then divides the list in two, with the half of the elements closest to the pivot in one list and the half furthest away in the other list. So both the leaves and the internal nodes in the tree hold data elements.

Algorithm 2.7.1: $buildTree(S : dataObects)$ for constructing the VP-tree

Data: A set of data objects S
Result: The root node to the VP-tree containing the objects in S

```
1  begin
2      if |S| = 0 then //create an empty tree
3          return Null
4      end
5      else //create a new node
6          node = Node
7          node.S_vt = selectVantagePoint(S)
8          S = S − {node.S_vt}
9          node.median = distance(node.S_vt)
10         S_1 = ∀S_i ∈ S | distance(node.S_vt, S_i) < node.median
11         S_2 = ∀S_j ∈ S | distance(node.S_vt, S_j) ≥ node.median
12         node.leftChild = buldTree(S_1)
13         node.rightChild = buildTree(S_2)
14
15 end
```

Because the list of samples is always split in two, the tree is guaranteed to be balanced. It also makes it easy to prune out unwanted branches of the tree. When doing a range search the distance from the query object to the pivot element needs to be computed. If the pivot element is inside the range it is added to the result list. It then checks if the distance plus the range is larger or smaller than the median stored in the node. If it is smaller, it recursively searches down the path of the smaller distances, and if it is bigger, it searches the other one. Sadly, if the search range is large, it can therefore happen quite often that both subtrees have to be search and

Algorithm 2.7.2: $selectVantagePoint(S : dataObects)$ as a way of selecting a vantage point.

```
Data: A set of data objects S
Result: The root node to the VP-tree containing the objects in S
1  begin
2      P = randomSample(S)
3      bestSpread = 0
4      forall P_i ∈ P do
5          D = randomSample(S)
6          mu = median{∀D_i distance(P_i, D_i) | D_i ∈ D}
7          spread = variance{∀D_i distance(P_i, D_i) − mu | D_i ∈ D}
8          if spread > best_spread then
9              best_spread = spread
10             best_p = p
11         end
12     end
13 end
```

no pruning occurs.

Unfortunately, it does not take any advantage of pruning because of already calculated distances, which some of the of the search algorithms for other tree structures do. This means that no checks for free are given, and therefore the algorithm has to compute the distance to every object it is examining. However, if the distance from the pivot object to the query object added to the median is less than the range, you can safely include every object in the left subtree, that is, the subtree that includes distances below the median. This is because of the Range-Pivot distance constraint, discussed in 2.5.1, and can in theory reduce the number of distance computations a bit if there are a lot of objects that are close together. However, this only works for KNN-searches, as here the distances have to be computed anyways to see if they fit into the list of k elements.

But possibly the largest problem with the VP-tree is disk IO. Because the tree only stores a single data object in each node, the algorithm has to make one disk access for each data object it examines, which makes the disk I/O the same as, or even worse if Lemma 2.5.4 is used, than the number of distance computations for search. To the VP-tree's defense, this was not an issue when it was designed, and [Uhl91] does not mention it at all, but it could still be a big bottleneck if the tree is used on large databases where the entries have to be stored on disk.

Algorithm 2.7.3: $rangeSearch(Q : dataObject, r : range, node : treeNode)$ which returns every object in the tree within the range of the target data object Q

```
Data: Target object Q and search radius r
Result: The list of data objects within the range
1  begin
2      d = distance(Q, node.S_v)
3      if d ≤ r then //include the vantage point
4          result+ = node.S_v
5      end
6      if d + r ≥ node.Median then //search the right subtree
7          rangeSearch(node.rightChild)
8      end
9      if d + node.Median ≤ r then //every object from the left subtree
       can be included
10         includeAll(node.leftChild)
11     end
12     else if d − r ≤ node.Median then //search the left subtree
13         rangeSearch(node.leftChild)
14     end
15 end
```

2.7.2 Multiple Vantage Point Tree

The Multiple Vantage Point Tree (MVP-tree) [BO97] is similar to the VP-tree in many ways, as they both use the median to find pivot elements and partition the data around them based on the relative distances between them. The main difference is that the MVP-tree uses precomputed distances between the data objects the pivot objects to help prune out the data elements without computing the distances between them and the query object.

A node in a MVP-tree consists of two vantage points, and therefore each level in the MVP-tree can be seen on as two levels in a VP-tree. This makes it possible to have a much larger fan-out in the MVP-tree, which results in more data objects in the leaf nodes and less in the internal nodes. The fan-out of the node can be set to any number, and is called m^2, where m is the number of partitions created by each vantage point. In this thesis, only the binary MVP-tree will be considered, which has a fan-out of 2^2.

There can be k data objects stored in any leaf node, and k is not bound to the fan-out in the internal nodes. In fact, [BO97] shows that it can be a good idea to use a larger number for k than for m^2; because the filtering based on the distances to parent objects can only be used in the leaf nodes, it is a good idea to have as many of the data objects there.

The first vantage point, S_{v1}, divides, just like a vantage point in a VP-

tree, the space into two groups. This once again is done by sorting the elements according to the distance to the vantage point, and then splitting it into two groups, with the half closest to the vantage point in one and the one furthest away in the other. It then finds another vantage point, S_{v2}, from the list of objects that are furthest away from S_{v1}. It then computes the distances between all the remaining elements to S_{v2}, and then sorts the two lists according to that, and partitions the space into four groups based around the medians.

The medians are stored in internal nodes. The median M_1 is for the partitions based on the first vantage point, and the medians $M_2[0]$ and $M_2[1]$ are based on the second. In the leaf nodes the distances between each of the data points and the vantage points of that leaf is stored, along with the first p distances between each of the data points and the vantage points along the path from the root node. The full pseudo code for the build procedure is shown in Algorithm 2.7.4.

For range search in the MVP-tree, [BO97] proposes a recursive depth-first search algorithm. For each node along the path, it first computes the distances from the query object to the two vantage points, and adds them to the result. It then checks the distances with the medians to see which subtrees that can be pruned out from the search.

If it's a leaf node, it uses the distance between the data objects and the vantage points to filter out objects by using the properties of the triangle inequality. The distance was stored in the node on creation time, so no distance computations have to be made. If this check is passed, it then uses the history of the distances stored in the node to do the same against the p first vantage points from the root to see if it can do any more filtering. As these distances were computed at creation time as well, no distance computations have to be made here either.

If the data object in question can not be filtered out using this method, the actual distance between the data object and the query object is computed. If it is inside the range of the search, the object is added to the result. The full pseudo code for the range search is shown in Algorithm 2.7.5.

The disk I/O problem is heavily reduced with MVP-tree compared to the VP-tree. This is partly because of the fact that the MVP-tree stores two or more vantage points instead of one, which reduces the number of disk accesses needed to get all the relevant vantage points. But the biggest improvement is that the MVP-tree preferably stores a large number of data objects in the same leaf node. Therefore, all objects in a leaf node can be retrieved with only one access, and the number of disk accesses needed should be reduced considerably if using a large value for k, the number of data objects in the leaves. This method has some similarities with the LAESA algorithm (see [MOV94]).

Algorithm 2.7.4: $buildTree(S : dataObjects, level)$ for building a MVP-tree

Data: A set of data objects S
Result: A root node to a MVP-tree containing the data objects

begin
  if $|S| = 0$ **then** //create an empty tree
    return *Null*
  end
  else if $|S| < k + 2$ **then** //create a leaf node
    $node = Node$
    $node.Sv1 = selectArbitrary(S)$
    $S = S - \{S_{v1}\}$
    forall $S_i \in S$ **do** $node.D_1[i] = distance(S_{v1}, S_i)$
    $node.S_{v2} = S_i \in S$ where $max\{D[i]\}$
    $S = S - \{S_{v2}\}$
    forall $S_j \in S$ **do** $node.D_2[j] = distance(node.S_{v2}, S_j)$
    return *node*
  end
  else //create an internal node
    $node = Node$
    $node.S_{v1} = selectArbitrary(S)$
    $S = S - \{node.S_{v1}\}$
    foreach $S_i \in S$ **do**
      if $level \leq p$ **then** $S_i.PATH[level] = distance(S_{v1}, S_i)$
    end
    $S = sort(distance(node.S_{vi}, S_i \in S))$
    $node.M_1 = median(S)$
    $SS_1 = \forall S_i \in S$ where $distance(node.S_{v1}, Si) < node.M_1$
    $SS_2 = \forall S_i \in S$ where $distance(node.S_{v1}, Si) \geq node.M_1$
    $node.S_{v2} = selectArbitrary(SS_2)$
    $SS_2 = SS_2 - \{node.S_{v2}\}$
    foreach $S_i \in S$ **do**
      if $level < p$ **then** $S_i.PATH[level + 1] = distance(S_{v1}, S_i)$
    end
    $SS_1 = sort(distance(node.S_{vi}, S_i \in S))$
    $node.M_2[0] = median(SS_1)$
    $CL[0] = \forall S_i \in SS_1 \mid distance(node.S_{v2}, Si) < node.M_2[1]$
    $CL[1] = \forall S_i \in SS_1 \mid distance(node.S_{v2}, Si) \geq node.M_2[1]$
    $SS_2 = sort(distance(node.S_{vi}, S_i \in S))$
    $node.M_2[1] = median(SS_2)$
    $CL[2] = \forall S_i \in SS_2 \mid distance(node.S_{v2}, Si) < node.M_2[2]$
    $CL[3] = \forall S_i \in SS_1 \mid distance(node.S_{v2}, Si) \geq node.M_2[2]$
    for *noe* **do**
      $node.children[i] = buildTree(CL[i], level+ = 2)$
    end
    return *node*

end

Algorithm 2.7.5: $rangeSearch(Q : targetObject, node : treeNode, r : range, level : levelInTree)$ for doing range search in a MVP-tree

```
Data: A target object Q and the search range r
Result: A list containing the similar objects
1  begin
2      ds_v1 = distance(Q, node.S_v1)
3      if ds_v1 ≤ range then result+ = node.S_v1
4      ds_v2 = distance(Q, node.S_v2)
5      if ds_v2 ≤ range then result+ = node.S_v2
6      if node is a leaf node then
7          foreach S_i ∈ node.dataObjects do
8              d_1 = D_1[node.p]
9              d_2 = D_2[node.p]
10             if (ds_v1 − r ≤ d_1 ≤ ds_v1 + r) and
               (ds_v2 − r ≤ d_2 ≤ ds_v2 + r) then
11                 forall i = 0..node.p do
12                     if ∀(PATH[i] − r ≤ S_i ≤ PATH[i] + r) then
13                         if distance(S_i, Q) ≤ range then result+ = S_i
14                     end
15                 end
16             end
17         end
18     else
19         if l ≤ p then PATH[l] = ds_v1
20         if l < p then PATH[l + 1] = ds_v2
21         if ds_v1 − r ≤ node.M_1 then
22             if ds_v2 − r ≤ node.M_2[0] then
               rangeSearch(Q, node.children[0], level + 2)
23             if ds_v2 − r ≥ node.M_2[1] then
               rangeSearch(Q, node.children[1], level + 2)
24         end
25         if ds_v1 + r ≥ node.M_1 then
26             if ds_v2 − r ≤ node.M_2[0] then
               rangeSearch(Q, node.children[2], level + 2)
27             if ds_v2 − r ≥ node.M_2[1] then
               rangeSearch(Q, node.children[3], level + 2)
28         end
29     end
30 end
```

2.7.3 M-tree

In [PZR96] they give three main goals for the M-tree, which, at the time of writing, no other metric index structure had accomplished. These were:

Paging: The tree consists of fixed-sized nodes for optimizing when trees are on external memory devices.

Balancing: To speed up the search speed all of the nodes should be at the same length from the root.

Dynamism: The tree should be able to deal with insertions and deletions without degrading search and without rebalancing globally.

The main difference between the M-tree and the other trees discussed so far is that the M-tree is a dynamic structure. While not as revolutionary as the VP-tree, the M-tree has spawned several successors as well, and has become a popular structure for testing for newer structures against. Like all the other structures in this thesis, it structures the objects depending on the distance between them, not because of position. The tree uses the generalized hyperplane principle described in [Uhl91]. The objects are then stored in fixed-sized nodes, so that the tree is paged. All the data objects are stored at the leaf level of the tree, while the internal nodes store routing objects, which are used to guide insertion, deletion and search. Each node can potentially hold any number of objects, which is one of the strengths of the M-tree, because a bigger fan-out improves both the number of distance computations and the number of I/O accesses.

The M-tree uses the triangle inequality to prune out branches of the tree from search without doing unnecessary distance calculations. It does this by using the distance between the parent and the query object and between the parent and the node. It then checks this against the range of the query search plus the range of the node. As can be seen, this is the same as the Range-Pivot Distance Constraint given in section 2.5.1, and the definition in Lemma 2.5.2. See [CPRZ97] for a proof of this lemma.

This way, only the distance to the node only has to be computed if there is a possibility that any objects of interest are within the subtree of the node. According to [CPRZ97] this can save up to 40% distance computations.

Data objects in the original M-tree [CPZ97] are inserted one one by using the insert-procedure (Algorithm 2.7.6). This is done by search downwards from the top until you find the ideal leaf node, and try to insert the data object there. The routing objects is found by examining all the routing objects in the internal nodes and then choose the one that is closest to the object you want to insert. The insert procedure is then recursively called on these objects. When a leaf node is finally reached, it tries to insert the object. If the leaf node is full, this can not be done, and the split procedure is called, which handles the rest of the insert process.

Algorithm 2.7.6: $insert(N : node, O_n : mTreeEntry)$ for inserting a data object into the M-tree

```
   Data: Root node and data object to insert
   Result: The object is inserted in the tree
 1 begin
 2   E = N.getEntries(); if Nisnotaleaf then
 3     forall O_r ∈ N do
 4       if distance(O_r, O_n) ≤ O_r.range then N_in+ = O_r
 5     end
 6     if N_in ≠ ∅ then
 7       O*_r ∈ N_in | min{distance(O*_r, O_n)}
 8     else
 9       O*_r ∈ N | min{distance(O*_r) − O_r.range}
10     end
11     insert(O*_r.node), O_n)
12   end
13   else
14     if Nisnotfull then N.store(O_n)
15     else split(N, O_n)
16   end
17 end
```

The split procedure (Algorithm 2.7.7) creates a new node so that more objects can be stored. It then promotes two objects from the set of the objects in the old node and the new object to be inserted, and these are the routing objects for the new nodes. The way the promote algorithm is implemented is completely independent from the rest of the tree, and [CPZ97] proposes several different algorithms for doing this, and analyzes the results. But the general point is to chose the two objects which the data can be partitioned best around. It then partitions the objects, which means assigning each of them to a node. [CPZ97] gives two examples for doing this, a generalized hyperplane method and a balanced method, and reports that the generalized hyperplane method gives the best results. Then it replaces the original routing object in the parent node for the first node with the new one, and tries to insert the second one as well. If this goes well, the procedure returns, but if the parent is full as well, the split function is again called on the the parent node with the second routing object as the new object to be inserted. If it is the root that is full, this means a new root is created and the two routing objects created by the split inserted into the new root.

Searching in the M-tree is heavily dependent on Lemma 2.5.2 to prune out nodes and therefore avoid unnecessary distance computations, and unlike

Algorithm 2.7.7: $split(N : node, O_n : mTreeEntry)$ for splitting full nodes to make room for new entries

Data: Entries in full node and entry to be inserted
Result: A tree where all the entries are inserted

```
1  begin
2      E = ∀entries ∈ N ∪ O_n
3      if notroot then
4          O_p = N.parent
5          N_p = O_p.node
6      end
7      N' = newnode
8      promote(E, O_p1, O_p2)
9      partition(E, O_p1, O_p2, N_1, N_2)
10     N.store(N_1.entries)
11     N'.store(N_1.entries)
12     if Nisroot then
13         N_p = newnode
14         root = N_p
15         N_p.store(O_p1)
16         N_p.store(O_p2)
17     end
18     else
19         N_p.replace(O_p, O_p1)
20         if N_pisfull then split(N_p, O_p2)
21         else N_p.store(O_p2)
22     end
23 end
```

the MVP-tree, it can also do so before reaching the leaves. The algorithm first checks whether it is a leaf node. If it is not, it goes through all the routing objects in the node, and uses Lemma 2.5.2 to see if the object is worth considering. Because the distance from the parent node to the query object has already been computed, and the range from each node to its parent is stored in the node at insertion time, this check is for free when concerning distance computations. If the object is worth considering, the distance is computed and a check if the distance from the object to the parent is less than the range of the node added to the range of the query. If the distance is within this range, the procedure is called recursively on the node associated with the routing object, if not, the object is discarded from the search.

If the node is a leaf node, more or less the same happens, but the check is now only against the radius of the search, because a data object has no

range associated with it. If the free check is passed, it has to compute the actual distance, and if it's less than the range as well, the data object is included in the search result. The pseudo code for the range search is shown in Algorithm 2.7.8.

Algorithm 2.7.8: $rangeSearch(N : node, Q : queryObject, r : searchRadius)$ for finding the objects in the tree within the search radius of the query object.

```
Data: node, query object and search radius
Result: A list containing the objects from the tree within the search
        radius
1  begin
2      O_p = N.parent
3      if Nnotleaf then
4          forall O_r ∈ N do
5              if |distance(O_p, Q) − distance(O_r, O_p)| ≤ O_r.radius + r
               then
6                  d = distance(O_r, Q)
7                  if d ≤ r + O_r.range then
8                      rangeSearch(O_r.node, Q, r)
9                  end
10             end
11         end
12     end
13     else
14         forall O_j ∈ N do
15             if |distance(O_p, Q) − distance(O_r, O_p)| ≤ r then
16                 d = distance(O_j, Q)
17                 if d ≤ r then
18                     result+ = O_j;
19                 end
20             end
21         end
22     end
23 end
```

Surprisingly, there exists no delete algorithm for the M-tree. [CPRZ97] only says that the delete method is not described because of space limitations, but to my knowledge, no deletion algorithm has been published so far. Also, the authors implementation of the M-tree apparently does not have a deletion procedure because of a bug in GIST [HNP95]. The way this is solved in this thesis is shown in section 4.2.3.

Because inserting objects one by one can not guarantee an optimal tree, the authors proposed a bulk loading algorithm in [CP98] to create a better tree when a lot of the data objects are known in advance. But the biggest advantage is that it drastically reduces the I/O costs when building the tree, as well as reducing the number of distance computations down to the level of the less advanced split policies.

2.7.4 Slim Trees

[TTSF00] has several interesting proposals, including an updated version of the M-tree called the Slim Tree. Firstly they propose two new methods for determining how good a tree is, called the fat-factor and the bloat-factor. These are meant to show how much overlap a tree has. If the overlap between nodes is large, the tree will have to search more nodes than it should, and query time will go up. They then propose the Slim Tree as way to reduce the bloat factor to fix this. The fact factor is about how many nodes that must be visited when a search is done, while the bloat factor also takes into consideration how efficient the nodes are at stored so that the volume of wasted disk space can be reduced.

They also present a new splitting algorithm based on minimum spanning trees. It creates a minimum spanning tree and then removes the longest edge, so the result is two new trees. It then takes the objects from the two trees and put them into their corresponding nodes, one for each tree. The objects with the minimum distance to all the other objects in each tree are chosen as the pivots for the nodes. But this will not guarantee that the node is split in a balanced way, so another way is to chose the one from the longest edges that results in the most balanced split.

The Slim Tree is s in [TTSF00] showed to be quite a bit better than the M-tree, and is a very interesting tree structure. But the scope of this thesis only allowed the GB-trees to be compared with the normal M-tree. However, the MST algorithm from [TTSF00] is implemented. The details about the MST algorithm can be found in Section 4.2.3.

2.8 General Balanced Trees

In [And99], Arne Andersson shows that a binary tree does not need any other properties than being of logarithmic height in order to be kept maintenanced as an efficient balanced search tree. The name general balanced tree is given because no shape restrictions are made except that it needs to be of logarithmic height. This is done by using partial rebuilding of the tree if simple requirements are met. The normal way to keep a tree balanced in an efficient manner is to use some more or less complex balance criterion. In the binary search tree world, as Andersson discusses, this includes structures like AVL-trees ([AVL62]), Symmetric binary B-trees ([Bay72]), Weight-balanced

trees ([NR72]) and several others. Because all of these structures are far from trivial, the question Andersson asks is if we really need all this when the only thing we really want is trees of logarithmic height.

To counter this, Andersson details a class of trees called general balanced trees (GB-trees), which only uses a very weak global criteria which is the relation between the size and the maximum height of the tree. This also has the benefit that no extra information is needed in the nodes in the tree. Partial rebuilding is then used on the tree to keep it balanced. This has been used in balancing weight-balanced trees before, but Andersson shows that the maintenance cost for general balanced trees are lower than those of weight-balanced trees.

The main idea behind maintaining a general balanced tree is to let the tree grow as it wants to as long has $h(T) \leq \lceil c * log_2(|T|) \rceil$. When this is not true, the tree can be partially rebuilt to give a lower height at a low amortized cost. This is done because of the observation given in Lemma 2.8.1, given and proven in [And99].

Lemma 2.8.1 *Let T be a binary tree, $h(T) > \lceil c * log_2(|T|) \rceil$. Let v be the lowest node (any of) T's longest path(s) such that $h(v) > \lceil c * log_2(|v|) \rceil$. Then*

$$\delta > (2^{1-1/c} - 1)|v| - 1$$

Just by using lemma 2.8.1, Andersson claims that it is possible to maintain a balanced tree efficiently during insertions, and states Theorem 2.8.1, which he then proves.

Theorem 2.8.1 *If no deletions are made, a binary search tree T with maximum height $\lceil c * log_2(|T|) \rceil > 1$, can be maintained at an amortized cost of $O(log_2|T|)$ per insertion. No balance information is needed in the nodes, only one global integer is needed.*

As we can see from Theorem 2.8.1, Andersson claims that as long as the tree follows his simple criteria, it can be maintained with only a amortized cost of $O(log_2|T|)$ when it comes to insertions, and still be balanced. By cost he means the number of internal nodes involved in the partial rebuilding.

He also gives a theorem for handling deletion in a general balanced tree (Theorem 2.8.2), which is also proven in [And99].

Theorem 2.8.2 *Given constants $c > 1$, and $b > 0$, a balanced tree T with maximum height $\lceil c * log_2(|T| + b) \rceil$ may be maintained without any balance information stored in the nodes, using two global integers, at an amortized cost of $O(log_2(T)$ per update.*

The result of these theorems is that insertion and deletion are performed as normal, and whether or not the balancing is to be done is defined by very simple rules. For insertions, the point where the partial rebuild is going to happen is found by traversing the insertion path until the lowest node v is found where $h(v) > \lceil c * log_2(|v|) \rceil$. When this node is found, the tree with v as a root is rebuild and the new root is inserted where v was in the tree.

For deletions, nothing is done until $d(T) \geq (2^{b/c} - 1)|T|$, but when this threshold is reached, the whole tree is rebuild for perfect balance. Just as with c, nothing is said about how big b should be.

The complete analysis of the constant factor in rebuilding the tree is too large to be shown here, see [And99] for the complete analysis. It also gives a comparison with weight-balanced trees to show that general balanced trees perform better. But the result is that the partial rebuilding using these simple rules should indeed be a good idea.

It is worth noting that Andersson gives no implementation details at all, just the theoretical basis, so how this compares on a more low-level scale is never explored. This should not be a real issue though, as it works with all existing algorithms for building a binary tree with logarithmic height.

Chapter 3

Methods and and Solutions

3.1 General Balanced Metric Trees

In theory, applying the GB-trees idea (Section 2.8) on a normal metric tree should make it able to compete with the dynamic versions that which have their own balancing criteria and ways to fulfill these. Just like [And99] says that most binary search tree structures uses advanced techniques for staying balanced, and using the the ideas given there could simplify the balancing prospect of metric trees just like they do for binary search trees. but there are a few problems that have to be overcome first. These include that similarity search in metric spaces is more complex than normal binary search and therefore require more advanced structures, and also because metric trees that do not use complicated rebalancing rules are usually static by nature.

Applying the GB-trees idea on metric trees is both similar and different from applying them to normal binary trees. Most metric tree structures, though some are based on having more than two siblings per node, can be binary if the parameters are set that way, and some are only binary. So in this respect, the ideas from [And99] can be implemented more or less directly, and the calculations given there will work the same way.

However, most metric structures uses more than two siblings per node at some level to allow pruning because of the principles given in section 2.5.1 to be much more effective. Giving an analysis of how the the ideas in [And99] will work on trees in more dimensions is beyond the scope of this thesis, so we will only consider trees that are binary when it gets to internal nodes, although one of the structures can have more than two data objects stored in them at leaf level.

There are generally two types of metric trees: static and dynamic. Most of the tree structures fall under static. The dynamic tree structures already have complex systems for staying balanced, so applying the GB-tree ideas to them is a bit useless. But one important fact is that dynamic trees generally

perform worse than static trees given the same input data. Even with the bulk loading algorithm, the M-tree does not perform any better than the old MVP-tree according to [CP98]. So the question is, can static metric structures be made dynamic, and with the help of the GB-tree principles become a better search tree than the existing dynamic structures using more complex balancing criteria?

Two ways to adapt static metric structures to dynamic structures using the rebalancing principles from [And99] are described in section 3.2 and 3.3. Other tree structures faces more or less the same problems when being adapted. Some of the problems occure because metric trees uses distances while binary search trees normally uses positions. Both of these implementations uses the static building as a bulk loading algorithm, and then allows for insertions and deletions of new and old objects.

To give an example, one problem with deletion is that it is probably much more problematic finding the ideal node to replace the one being deleted if it is an internal node. This is because the tree has much less knowledge about how the subtree is partitioned below any point in the the tree, so the node being the closest one in the subtree can not be found without doing a much wider search which will include a many extra distance computations and disk IO. But a potentially much worse problem is that how the nodes are distributed could change if a different node is selected to replace it. The closer the better, but it is impossible to guarantee that it will not happen unless the new object is identical to the one deleted. In any way, the cost of deleting an object is suddenly much higher than it would have been for a binary search tree.

Another problem is finding a new pivot when inserting. Most metric trees use some sort of pivot value to structure the objects around. Ball composition methods use the median to the pivot object, while generalized hyperplane methods use the distances from the nodes to the pivot objects directly. But how is this done when there are no objects to calculate these values from? In the original algorithms, this was not a problem, because if this was a leaf node, those values were not needed because of the static nature. However, this has suddenly become an issue because now more nodes could be added underneath the leaf nodes from the static building. When more objects are inserted, this must be dealt with, and the cost of inserting objects is suddenly higher than it would be for a binary search tree.

3.2 General Balanced Vantage Point Tree

The first algorithm to be tested with the GB-tree paradigm is the Vantage Point Tree (VP-tree), described in Section 2.7.1. As this in its original structure is a binary tree with the same properties pretty much like normal binary search trees, applying the methods described in [And99] to the VP-

tree should be easier than most other structures.

The first problem in applying the GB-tree paradigm to the VP-tree is that the VP-tree is a static structure. There have been proposed ways to do the VP-tree dynamic (see [FsCCM00]), but not by using such simple criteria as the ones proposed by Andersson. Here follows a description of how the dynamic VP-tree used in this thesis works.

The insertion algorithm works more or less the same as in a normal binary search trees. To insert the object i, the algorithm traverses recursively downwards the tree until an empty leaf is found, and the node is inserted there. But of course, the node uses the distance between i and the pivots in relation to median as the criteria for which subtree to traverse. If the other child of the new parent node p to i is empty, the median in p is updated to be the distance between i and p, or $d(i, p)$. If not, the median in p is kept as it is. When a object is inserted, v, that is the total number of empty leaves, has to be updated. Because a new node will occupy one empty leaf but add two, v is incremented by one. Along the way, the path to the inserted node is recorded, to help with the rebuilding. The pseudo-code for the insertion algorithm is found in Algorithm 3.2.1.

Sadly, this way could potentially lead to very unbalanced insertions, as there is no guarantee that the chosen median will not be a very bad one, leading to every node after that one being inserted in the opposite subtree. But hopefully, the partial rebuilding will do that the tree is rebuilt and therefore balanced with much better choices for medians.

Deletions are a bit more tricky. The problem is, as mentioned in Section 3.1 internal nodes. Unlike in binary search trees, where a perfect replacement can be found and it is known where it is, this is not necessarily the case in a VP-tree. First of all, the algorithm does not know exactly where the best replacement could be, as there exists no information in the internal node about how the data is partitioned further down in the tree, except for the fact that the ones with the smaller distance than the median are to the left and the larger to the right, of course. This means that the algorithm could be required to perform quite a number of distance computations just to find the best node. Second, once this node is actually found, there is no guarantee that the internal one can just be removed and the leaf inserted instead, because this could change which objects should be in the left and right subtree. So in that case, all the distances between the new pivot and the objects further down the tree would have to be computed, and the necessary relocations must be made.

As this is not a very good solution, there are several ways this can be handled. One is to simply mark the node as deleted, but not physically delete it. Then, when the next rebalancing happens, the node will not be included when the tree new tree that is created. This is sort of similar to the way deletion is done in List Of Clusters (see [CN00]), and has several benefits, as no costly distance computations needs to be made to remove

Algorithm 3.2.1: $insert(node : treeNode, io : insertNode, path : treeNodeList)$ for inserting an object into a GBVP-tree

Data: A tree node and the node to be inserted
Result: The root node to the VP-tree with the node deleted

```
1  begin
2      path = path + {node}
3      distance = d(node.pivot, io)
4      if node.leftChild = ∅ and node.leftChild = ∅ then
5          node.leftChild = io
6          node.median = d
7          v = v + 1
8      end
9      else if distance < node.median then
10         if node.leftChild = ∅ then
11             node.leftChild = io
12             v = v + 1
13         end
14         else
15             node.leftChild = insert(node.leftChild, io, path)
16         end
17     end
18     else
19         if node.rightChild = ∅ then
20             node.rightChild = io
21             v = v + 1
22         end
23         else
24             node.rightChild = insert(node.rightChild, io, path)
25         end
26     end
27     return node
28 end
```

the object, and no rebalancing needs to be done. The tree will also hold its original balance, as the originally chosen vantage point is hopefully a good one. One thing that is not so good about this way though, is that that the height of the tree will not decrease with deletions. This causes several problems, one being that the search time in the tree will increase because no nodes are removed, but this could probably be remedied by the partial rebuilding because the height of the tree will increase faster, resulting in an increased number of partial or complete rebuildings. A related problem could be that the formulas in [And99] are based on the fact that deletions

will reduce the height of the tree, and therefore the number of complete rebuilds could become too large considering that only partial rebuilds may be necessary.

Another approach is to simply rebuild the subtree starting with the internal node due for deletion, and just exclude the node from the rebuilding. This will cause a new balanced subtree with a new root, and therefore of course be of perfect height. But the problem is clearly that it would require a large bunch of partial rebuilding. Because the VP-tree has no containers for leaf nodes, roughly half of the nodes inn the tree will be internal nodes, which means that probably around half of the deletions would require some partial rebuilding. On the other hand, as this will reduce the number of partial rebuildings in relation to the global criteria of GB-trees, because the tree will be kept in an overall better shape. But one important question that should be asked is whether or not a deletion should be given the same weight as it is done in [And99] if deletions are handled this way, because, given that the tree has already been partially rebuilt, the importance of a complete rebuild may be reduced.

If the node to be deleted is a child, it is just removed. And of course, v has to be updated if any changes are made to the tree. Pseudo-code for the delete algorithm using rebuilding is shown in Algorithm 3.2.2. Because the algorithm using marking only needs very slight changes to this one, it is not given here, but the only difference is that instead of rebuilding the tree and returning it, it just returns the original node after marking it.

If it is determined that a partial rebuilding is to be done, according to the criteria in [And99], the algorithm travels down the path given in the insert, and checks where the partial rebuilding is to be done. When the node is found, the algorithm simply uses the same algorithm for building the tree which is used when the tree is first created, as described in Section 2.7.1. Pseudocode is given in Algorithm 3.2.3. When the whole tree has to be rebuilt, all the objects in the trees are just collected and the rebuilding from Section 2.7.1 and the root of the tree is set to the root returned from the rebuilding.

3.3 General Balanced Multiple Vantage Point Tree

Applying the GB-tree paradigm to the MVP-tree (Section 2.7.2) is a bit more difficult than with the VP-tree for a number of reasons. While there are many similar characteristics between the VP-tree and the MVP-tree, the MVP-tree also have several features that makes it more different from a regular binary search tree than a VP-tree.

First of all, the MVP-tree uses buckets to store the leaf nodes in that can be of variable size, which makes it a bit more difficult to make the tree dynamic than with a VP-tree, because to be effective, the structure will need

Algorithm 3.2.2: $delete(node : treeNode, do : deleteNode)$ for deleting an object from a GBVP-tree using rebuilding

```
   Data: The node in the tree and the object to be deleted
   Result: The root node to the GBVP-tree with the node deleted
 1 begin
 2     if node.vp = do then
 3         if node.leftChild = ∅ and node.rightChild = ∅ then
 4             return ∅
 5         end
 6         else
 7             nodes = getAllNodes(node.leftChild) +
               getAllNodes(node.rightChild)
 8             return buildTree(nodes)
 9         end
10     end
11     else
12         distance = d(node.vp = do)
13         if distance < node.median then
14             node.leftChild = delete(node.leftChild, do)
15         end
16         else
17             node.rightChild = delete(node.rightChild, do)
18         end
19         return node
20     end
21 end
```

some sort of split policy. One curious problem that rises is to actually define what is an empty leaf node when a leaf can have n number of objects in a bucket. Counting every free space in the bucket as an empty node would be a bad idea, because then the number of empty nodes compared to the tree height would become way off. So the best way to do it is probably to just consider it empty if the whole bucket is empty, in other words if the leaf node contains no data objects other than the pivot object(s). This should not directly influence the criteria for balancing the tree, as the height of the tree is unaffected.

As the MVP-tree combines two and two levels in the tree into one node, and then shares the second vantage point between all four children, one node can not be considered one level. This is because the tree then can not be categorized as a binary tree when one node will have a fan-out of at least four child nodes. So instead, each node in the GBMVP-tree is considered to be two levels to make up for this.

Algorithm 3.2.3: *partialRebuild*(*path* : *treeNodeList*) for doing a partial rebuilding of the tree

```
   Data: The insertion path
   Result: The root of the new subtree
 1 begin
 2     i = 0
 3     while i < path.length do
 4         if path.length − i ≤ ⌈c ∗ log2(path[i].v)⌉ then //This node is
           not too high
 5             nodes = getAllNodes(path[i − 1])
 6             if path[i − 2].leftChild = path[i − 1] then
 7                 path[i − 2].leftChild = buildTree(nodes)
 8             end
 9             else
10                 path[i − 2].rightChild = buildTree(nodes)
11             end
12             return
13         end
14         i = i + 1
15     end
16 end
```

Because of the buckets, insertion needs some sort of split function to deal with what happens when a bucket is full. The one used in this project simply uses the buildTree-algorithm for the MVP-tree (Algorithm 2.7.4). It takes the two vantage points and the data objects in the node, adds the new object to be inserted, and simply rebuilds them. This will create three new nodes, where the root node of the new tree will replace the old leaf node. In this way, the height of this branch of the tree should grow with two each time a node is split. When a leaf node is reached and the bucket is not full, the data object to be inserted is just put in the bucket. When a node is inserted,the distances in the path to the node must also be recorded and stored in the node so that filtering can be used on the node. The path of the nodes must be recorded as well in case of a partial rebuild when the insertion is complete. The pseudo code for the insertion algorithm can be found in Algorithm 3.3.1, and the split algorithm can be found in Algorithm 3.3.2. It is to be noted that any split-algorithm can be used, as long as it returns the node to a new subtree.

As can be seen in Algorithm 3.3.2, one problem is that the path to the new nodes has to be updated. This means that the buildTree-algorithm for the MVP-tree (Algorithm 2.7.4) must be modified to be able to take an already created path downwards the tree. This is trivial though, as it is just

Algorithm 3.3.1: $insert(node : treeNode, in : insertNode, level : integer)$ inserting a node in a GBMVP-tree

```
   Data: A tree node, the node to be inserted and the level in the tree
   Result: The root of the new subtree with the inserted node
 1 begin
 2     if node.children = ∅ then //node is leaf
 3         if node.dataObjects.length ≥ k then //list of dataobjects if
           full
 4             node.children = node.children + split(node, in)
 5             node.dataObjects = ∅
 6         end
 7         else
 8             if level ≤ p then ds_v1 = distance(in, node.S_v1)
 9             node.PATH[level] = ds_v1
10             if level < p then ds_v2 = distance(in, node.S_v1)
11             node.PATH[level + 1] = ds_v2
12             node.dataObjects = node.dataObjects + {io}
13         end
14     end
15     else
16         ds_v1 = distance(in, node.S_v1)
17         ds_v2 = distance(in, node.S_v1)
18         if level ≤ p then node.PATH[level] = ds_v1
19         if level < p then node.PATH[level + 1] = ds_v2
20         if ds_v1 < node.M_1 then
21             if ds_v2 < node.M_2[0] then
               insert(node.children[0], in, level + 2)
22             else insert(node.children[1], in, level + 2)
23         end
24         else
25             if ds_v2 < node.M_2[0] then
               insert(node.children[2], in, level + 2)
26             else insert(node.children[3], in, level + 2)
27         end
28     end
29 end
```

Algorithm 3.3.2: $split(node : treeNode, in : insertNode)$ for doing a simple split of a node

Data: A tree node and the node to be inserted
Result: The root of the new subtree after the split

```
1 begin
2 |  nodes = getAllNodes(node) + {in}
3 |  return rebuildTree(nodes, path, level)
4 end
```

to send the path as a parameter and then include it with the objects when they are inserted into a node. When the tree is built from the ground, an empty list is simply used as a parameter instead.

Deletion of internal nodes is done much in the same way as with the GBVP-tree (Section 3.2), as the whole subtree is rebuilt when a pivot object in an internal node is the one to be deleted. This will again result in more rebuilding, but hopefully the fact that the subtree is now balanced will reduce this penalty, as the number of rebuilds done because of the global criteria will be reduced.

To better optimize the tree when deleting nodes from the buckets in the leaves, some inspiration from the R-tree [Gut84] was considered, which is also the basis for the deletions in M-trees (Section 2.7.3) that is used in this thesis. This led to quite a few problems though, again because of the split around the medians. The idea of the nodes deleting themselves upwards in the tree if the nodes have less than a given number of objects in them will also not work, because the objects will be inserted again at the exact same spots unless the pivot objects are renewed. And if the pivot values are renewed, the whole subtree has to be rebuilt because there is no way to be sure if there are objects in lower branches that are now partitioned wrong.

Because of this, the implementation used here simply deletes the objects if they are in the buckets in the leaves. The balancing criterion of the GB-tree should make sure that the tree can not stay unbalanced for long, so even if the query search right after the deletion could end up visiting more nodes than needed, this should be fixed once the first partial rebuild for this section is initiated.

Another point against implementing any extra balancing when nodes are deleted is that implementing too many complicated ways to balance the trees will actually work against one of the points presented in [And99], as one of the main points there was that no complicated balancing criteria is needed to keep a binary tree balanced.

One of the two delete algorithm for deleting objects from the GBMVP-tree is showed in 3.3.3. However, there is one modification that can be done, which was discussed in Section 3.2, and that is that the vantage points are

only marked as deleted and not actually removed from the tree at the point of deletion. Instead, the vantage points are not included when the next rebuild happens. As the modifications to Algorithm 3.3.3 that are needed are very small, no pseudo code is included.

3.4 Test Methods

3.4.1 About The Experimental Design

In this thesis, mainly two things are the criteria for how a tree is judged. These are the number of distance computations and the number of disk I/O that has to be done to build, search and maintain the trees. As described in Section 2.1.2 and Section 2.3, which one of these are the most important depends on the problem domain; If the distance computation is heavy enough to compute, the number of disk accesses will not matter that much because, in the end, almost all the run time is used on the distance computations. However, if it is not very hard to do the computation, the number of disk accesses will be much more important because reading and writing to secondary memory is very slow.

In the end, the actual time it takes to run an algorithm is what matters. However, there are several reasons for why only these two criteria are used, and why not the actual running time is recorded and analyzed.

First of all, all implementations (see Chapter 4) are done in a high level language. Therefore there is too much magic going on behind the scenes to be sure exactly why an algorithm perform they way it does without some serious knowledge and analysis of the programming language, and this is beyond the scope of this thesis. As a result, it would be difficult to say exactly why an algorithm does not perform as well as it theoretically should have done and why another, in comparison, performs better.

Another reason is that it is difficult to say if the implementation is perfect or not. This is partly linked to the previous reason, but it is more on the personal level. Because tinkering with algorithms will always be able to improve how they perform at runtime, it is hard to say exactly when it is good enough to be tested against other algorithms.

Finally, and most importantly, the problems these algorithms are important for are also the problems where the number of distance computations and disk access is so important that they overshadow most other aspects. While this does not need to be true for more than one of them, at least one of them will require so much time to calculate that the rest of the running time will be minimal compared to it.

This does not mean that the running time of algorithms are not important, but it is the reasons why it is not the focus of this thesis. Of course, actual implementation details should be considered if any of the structures are to be used in a real system. As i.e. [MS94] shows, theoretically good

Algorithm 3.3.3: $delete(node : treeNode, do : deleteNode)$ for deleting an object from a GBMVP-tree using rebuilding

Data: The node in the tree and the object to be deleted
Result: The root node to the GBMVP-tree with the node deleted

```
1  begin
2      if node.S_v1 = do then
3          nodes = {node.S_v2}
4          forall C_i ∈ node.Children do
5              nodes = nodes + getAllObjects(C_i)
6              return buildTree(nodes, node.S_v1.PATH)
7          end
8      end
9      else if node.S_v2 = do then
10         nodes = {node.S_v1}
11         forall C_i ∈ node.Children do
12             nodes = nodes + getAllObjects(C_i)
13             return buildTree(nodes, node.S_v1.PATH)
14         end
15     end
16     else
17         if nodeisleaf then
18             if ∃O_j ∈ node.dataObjects | O_j = do then
19                 node.dataObjects = node.dataObjects − {O_j}
20             end
21         end
22         else
23             if ds_v1 ≤ node.M_1 then
24                 if ds_v2 < node.M_2[0] then delete(node.children[0], do)
25                 else delete(node.children[1], do)
26             end
27             else
28                 if ds_v2 < node.M_2[0] then delete(node.children[2], do)
29                 else delete(node.children[3], do)
30             end
31         end
32     end
33     return node
34 end
```

does not necessarily mean good in practice. But [MS94] also discuss how the results changed when the implementations got better, which backs up why it is not considered here.

The counting of distance computations and disk I/O is done on a pure logical level. See Section 4.4 for a description of how this is implemented.

3.4.2 Test Sets

For the tests done in this thesis data objects placed in a spatial domain are used, and they are distributed after two different criteria. The first one is clustered data, which means that the all the data objects belong to one of potentially many clusters. The range between the center of the cluster and the data objects that belong to it are random, but they are required to be within a given distance determined by some criteria. So if D is the set of data objects, C is the set of data clusters and X is the set of random variables following these criteria and n is the number of dimensions, $\forall d_i \; d_{i0}, ..., d_{in} = c_j + x_k \mid d_i \in D, C_j \in C, x_k \in X$.

The other one are uniformly distributed data that are simply given a uniformly distributed random value for each dimension. There is some range requirement so that the random variables can not be outside. So if D is the set of data object and X is the unlimited set, all random uniformly distributed variables $\forall x_i \; minValue \leq x_i \leq maxValue \mid x_i \in X$, $\forall d_i \; d_{i0}, ..., d_{in} = x_k | d_i \in D, x_k \in X$.

3.4.3 Distance Functions

Below follows a description of the two distance functions used in the tests in this thesis. While they are briefly mentioned with the rest of the common distance functions in Section 2.3, they are described in more detail here, as well as details about how they are used in this thesis.

L_2 metric

Because all the tree structures tested in this thesis are created to handle continuous distance functions, most of the test in the in this thesis uses a continuous distance function called The L_2 metric, which is better known as the Euclidean distance. The L_2 metric distance function is given by $distance = \sqrt{v_1^2 + ... + v_n^2}$ where v is a feature vector consisting of numbers and n is the number of dimensions. It is used for several reasons, mostly because it is, as it can be computed in linear time, very fast to compute and therefore very efficient when testing out the different algorithms. It is also easily understood and because of that it is easy to see when the results are the way they are. It is important to note that, although all the data objects have spatial coordinates, all that is taken into consideration is the distance

function and no knowledge about the geometric properties of the objects are presumed.

Edit Distance

The Edit Distance, or Levenshtein Distance, is used to test the trees on discrete data. The Edit Distance, as described in Section 2.3, is an algorithm for computing the distance between two text strings. It does this by counting the minimum number of changes that has to be done in order to transform one of the strings into the other. There are three possible ways to mutate a string.

1. substitute a letter
2. insert a letter
3. delete a letter.

The general Edit Distance function lets the implementation decide the weights individually for the three operations. However, in order for this to be a metric distance function, insert and delete needs to have the same weights to make sure that the symmetry requirement is in place. As an example, with insert set to 2 and delete to 1, distance(”some”, ”something”) would be 5, while distance(”something”, ”some”) would be 10. The most common way to use the algorithm is to use 1 for each of the weights.

Sadly, the Edit Distance is far slower than the L_p metrics, and have a time complexity of $O(mn)$ where m is the length of the first word and n is the length of the second word.

3.4.4 Searching

To give a good verdict of how the trees perform, both range search and KNN-search will be done on all the trees discussed in this thesis. Although they are linked in many ways, they can perform quite differently. This is especially true for more advanced tree structures where more efficient pruning is applied.

Range Search

For range search, different search ranges are used. For searches in the L_2 metric, the range is scaled after how many dimensions are used. A range of 0.1, 0.2, 0.3, and 0.5 is used. As a search of range 0.5 should include at least half of the objects in the tree, the search is scaled so that it is this way. Therefore, for n dimensions, $range = \sqrt{range_1^2 + range_2^2 + ... + range_n^2}$. Without doing this, search in higher dimensions would give no results at

all, because the average distance will increase with the number of dimensions.

Done this way, the search efficiency should vary a lot between clustered and uniformly distributed data. As long as the search hits reasonably close to a cluster, a lot of different search objects will be found. Of course, if it does not, it could end up empty or close to, so the number of clusters will have a lot to say.

For uniformly distributed data, the search will return more or less the same amount of of hits each time, but the number will probably be less. This should also increase a lot when dimensions grow, because there are no centers, so the pivot objects will be much further away from their children than with clustered data. An advantage of this is that the pruning of the search could improve, while the number of actual hits will decrease.

K-Nearest Neighbor Search

No upper bound is given for the KNN-search used for the test in this thesis. It is quite normal to set a maximum range for how far away an object in the tree can be from the query object, which is because normally objects outside of some range is not really interesting even though they are closer than most other objects. While this can be be a good idea, the search becomes more or less exactly like a normal range search if a maximum range is given no k objects are inside of the given range. Therefore, in this thesis, the KNN-search is a pure KNN-search where the result will always return k objects, at least if there are that many objects in the tree. While the implementation of the KNN-search for the different type of trees will be different, with some tree structures optimizing more than others because they can guess where the closest samples will be and therefore do prune better, they will all follow this principle.

3.4.5 Repetitions

It is important to that the results from the searches are as general as possible. One chosen query object could give a very different results from another, and as there are a lot of random elements in the building of a tree, so could one tree from another. So it is important to make sure that the results are as representative as possible for the structures.

For each of the given number of objects to be included int he tree, several trees are created to make sure that the searches are not to affected by some very lucky or unlucky randomization. First, all trees are dependent on the randomized nature of the test set. The randomized nature of the picking of the vantage points is the biggest problem for the VP-tree and MVP-tree, while the most important factor for the M-tree is the order in which objects from the test sets are inserted and deleted. Both of these factors

are actually important for the GB-tree variants, as they are both vulnerable to the choosing of vantage points and the order of insertion and deletion, though the partial and complete rebuilding makes it much more dependent on the former than the later.

Searching is done much in the same way. A large number of searches has to be made on each of the tree structures in order for the search to be able to give a generalized result, as the picking of the search object is important. While a KNN-search can result in a very fast convergence toward the optimal k objects, and therefore prune out most of the search branches because of a low maximum range, the complete opposite could be true if the query object is not right inside a cluster.

As an example, if 10 trees are chosen to be built for each number of objects, and 100 searches are to be made on each of those for each range or k, $10 * 100 = 1000$ searches are made for each number of objects for each range and k.

When discussing the individual tree structures by themselves, a random tree and random test set and random query objects are created individually for them. However, when directly compare them against each other, the test sets used are the same to make the comparing as fair as possible. The same is done for the query objects, as the same query objects are used for each of the trees to remove any chance that one tree structure got ”better” query objects than another.

Chapter 4

Implementation

4.1 Language

All algorithms were implemented in Python. Python was chosen because it is very fast to prototype algorithms in the language. As the purpose of this thesis is only to test different methods for number of distance computations and disk access, the runtime speed of the program was not a major priority. Because code written in Python is relatively slow compared to several other languages, the algorithms should probably be implemented in a more suitable language, like C++, to optimize more for speed.

For optimization of the running time, Psyco [Rig04] was used, which is a kind of a just-in-time (JIT) compiler for Python, although the author claims it more of a just-in-time specializer.

4.2 Algorithms

This sections describes the actual implementation of the algorithms used in the tests. This includes what is actually implemented, what changes are made from the theoretical versions, optimizations done to speed up search and limitations of the implementations.

4.2.1 VP-tree

Both the version from [Uhl91] and [Yia93] are implemented to test if the improvement in the later really have that much to say when compared to other indexing structures. The first one just takes a random sample from the list of data objects and uses it as a vantage point. The second one uses a random function to decide which samples are to be included in the sample sets (the $getSamples(S)$ function), as seen in Algorithm 2.7.2. If the random number is higher than the given threshold, the sample is included in the list.

The lower the threshold, the better the vantage point should be, and therefore, in theory, a more optimal tree should be built. But because of the higher building cost because of the extra distance computations needed, the question is if this is really worth it, and if so, where the threshold should be.

4.2.2 MVP-tree

The implementation done for the tests in this thesis follows the algorithm described in [BO97] and [BO99], implementing all the optimization considering reductions in distance computations. This means that, except for the necessary distance computations between the query object and each of the vantage points in each node visited, no other distance computations are made until it is made sure that the leaf nodes can not be filtered out by the already computed distances.

There is one difference between the range search algorithm used here and the one in [BO97], because the algorithm given there prunes out subtrees that could hold potential nodes within the search range. This is because it excludes all subtrees where the spherical cut made by the range around the query object isn't completely inside the space given by the vantage points and medians. This has several problems, the most obvious being that a lot of objects will be pruned out even if they are well within the range of the search object. This results in the odd behavior that increasing the search range will in several occurrences decrease the number of data objects found. Therefore it will be used a slightly modified version here where all subtrees containing potential subtrees will be searched. But it is to be noted that the same algorithm is repeated in [BO99], this could be an error in the interpretation of the algorithm

For some reason no version of the object-pivot constraint (see Section 2.5.1) is used in [BO97] for directly including nodes in the MVP-tree when doing range searches. This was used in the original VP-tree in [Uhl91], and could be able to save some distance computations there because the nodes below the median can directly included if the criterion is satisfied without actually computing the distances. Because this should work just as well in the MVP-tree, the implementation used in this thesis includes it. Therefore, if $distance(p, q) + median \leq range$, the subtree below the median is directly included. This is done when checking against both pivot nodes in this implementation of the MVP-tree, and should be able to save some distance computations, especially when the range is large and the nodes clustered together.

4.2.3 M-tree

In [CPRZ97] they propose several promotion algorithms for picking new pivot nodes when nodes have to be split because they are full. Out of the

ones discussed, the mMRAD algorithm gives the best results. This algorithm considers all possible pairs of objects to check which pivots that minimizes the maximum of the two radii. How the objects are partitioned amongst the two pivots are determined by the partitioning algorithm, and here the generalized hyperplane method gives the best results. While this way to promote gives some very good results, it is also computationally expensive, as the time complexity is $O(n^3)$. But because this algorithm was, according to [CPRZ97], this is the only one of the originally proposed algorithms that is implemented in this thesis, along with the random one used as reference.

As discussed in Section 2.7.4, [TTSF00] proposes another promoting algorithm based on the well-known Minimum Spanning Tree (MST). Also as noted, 2.7.4 shows that this gives almost as good results as the mMRAD algorithm, but is much faster to compute thanks to the time complexity of only $O(n^2 * log(n))$. Because of these good results, MST partitioning is also implemented. There are a lot of different algorithms proposed for creating MSTs, where the most commonly used are Prim's and Kruskal's algorithms. In this implementation, Prim's algorithm is used, but any algorithm that produces a MST can be used. See [MS94] for a detailed discussion about various types of MST algorithms and how effective they are, both in theory and implementation. However, unless an extreme fan-out is used, this should not have much to say.

It is to be noted that as we only look at the number of distance computations and disk I/O (see Section 3.4), the improved running time of the MST algorithm will not be shown in our results. But as the MST algorithm also needs $O(n^2)$ distance computations for building the graph, and according to [TTSF00], does not partition the data quite as well as the mMRAD algorithm, the MST algorithm will actually show up a bit worse in the results here. Therefore it is important to remember the limitations of the test in this thesis when considering a partitioning algorithm based on these results, as the MST algorithm could be better in practice, analogous to the good old quicksort vs. mergesort problem.

Just as with the MVP-tree, no reason was found why there is not any automatic inclusion of objects on a pivot level when doing range search in the M-tree. Because the upper bound of the objects beneath a routing object is known, it is possible to check whether all nodes beneath the routing objects have to be within search range by using Lemma 2.5.2 from the Pivot-Range Distance Constraint (Section 2.5.1). This has been implemented, and should be able to save some distance computations when the range grows because, as the range grows, the number of nodes that can be directly included should grow as well. As the algorithm in [CPRZ97] has no way of using the extended range to make the search more effective, this could be of help. See Section 5.4 for a comparison between one using this improvement and one that does not.

4.2.4 Deletion in the M-tree

As discussed in Section 2.7.3, the authors of the M-tree released no delete algorithm for it. Because a delete algorithm was needed to make the proper comparisons in this thesis, several options where considered for the implementation to make deletions possible.

The simplest version is to just delete the object from the leaf node and leave everything as it is. This could result in trees with a lot of empty leaf nodes, and this could result in slow tree traversal because the tree would consist of more nodes than necessary, and could also result in unbalanced trees.

A probably better method is the one used in the original R-tree [Gut84], and also in some of the other versions of it like the R*-tree ([NBS90]), where the entire node is deleted if there are less than $M/2$ objects left in it after a deletion, where M is the maximum number of objects that can be stored in a node. Then objects that where left in the deleted node are inserted into a list R. After that, the algorithm proceeds all the way to the root node. When the root node is reached, all objects in R are inserted into the tree with the normal insert algorithm. The pseudo code for the slightly changed version to suit the M-tree can be seen in procedure 4.2.1. This algorithm is used in this implementation of the M-tree. The algorithm for the condense operation is shown in procedure 4.2.2.

One problem with deletion is to actually find the desired node. The split algorithm could do so that the closest node to the target object is no longer the right path to traverse, because it changes the pivot objects in the node after insertion. Therefore several branches may need to be searched. Because of this, the algorithm traverses the subtree with the closest distance first, and then tries the next one. If no subtree is found which the object could be in according to Lemma 2.5.2, the algorithm backtracks. Sadly this leads to increased delete time, and a more direct access method would have been preferred.

To promote a new routing object in the condense algorithm, this thesis uses a minimum radius function that chooses the object that gives the least radius, or range, as the new pivot object. Just like the mMRAD algorithm, this needs $O(n^2)$ distance computations, but the partitioning should be very good considering the performance of the mMRAD algorithm in [CPRZ97]. To see how much this has to say, a random algorithm has also been implemented. While this should lead to worse query performance, it does not need any distance computations. And finally, the option of just ignoring to update the routing object is given. Because the routing object is more or less ideal in the first place, and the nodes are deleted once the number of nodes is less than $M/2$, where M is the fan-out, constantly updating the routing object could be a waste of distance computations. But if this is done, the range is still adjusted to make sure it is not larger than it needs

Algorithm 4.2.1: $delete(N : node, DO : deleteObject)$ for deleting objects from the the M-tree.

```
   Data: Node N and object to delete DO
   Result: The tree without the object
 1 begin
 2     if N is leaf then
 3         N = N − DO
 4         condenseTree(N)
 5         if N is root and N.numberOfChildren() == 1 then
 6             root = Oi.node | ∃O_i ∈ N
 7         end
 8     end
 9     else
10         O_p = N.parent
11         O = ∀O_j ∈ N | |distance(O_p, DO) − distance(O_j, O_p)| ≤
           O_j.radius
12         sort(O)
13         while notfound do
14             O_j = next(O)
15             delete(O_j.node, DO)
16         end
17     end
18 end
```

to be. As all the distances are stored already, this does not increase the number of distance computations.

4.3 General Balanced Trees

The algorithms for the GBT variants are implemented more or less exactly as they are described in Chapter 3.

One difference from the implementation and the theory is that [And99] claims that no extra information has to be stored in the nodes, just the global values for the height and the number of free nodes v. While this is indeed possible, it has one very big disadvantage. When searching for partial rebuilds, the height insertion path has to be checked against the v of the nodes. The problem is that there is no way of knowing exactly how many free nodes there are underneath the node in interest without doing an exhaustive search if there is no information stored about it anywhere, which in turn would cause a very large number of unnecessary disk I/O. Therefore, in this implementation, the value v is stored in every node to make sure that no extra searching has to be done. While this will increase the size of the

Algorithm 4.2.2: *condenseTree*(N : *node*) for removing underfilled nodes.

```
   Data: Node N
   Result: A condensed tree
 1 begin
 2     X = N
 3     R = ∅
 4     while X is not root do
 5         O_p = X.parent
 6         N_p = O_p.parent
 7         if X contains less than M/2 entries then
 8             N_p.children = N_p.children − X
 9             R = R + {X}
10         end
11         else
12             promote(O_pn)
13             N_p.replace(O_p, O_pn)
14         end
15         X = N_p
16     end
17     forall R_i ∈ R do
18         insert(R_i)
19     end
20 end
```

nodes, the reduce in disk I/O while searching will hopefully make up for it. The value for v is updated when an insertion, deletion or rebuilding has happened. The nodes in question are updated when the recursion is backing up toward the root after the desired operation has been performed.

Both the delete methods described in Sections 3.2 and 3.3 are implemented. While the algorithm that has automatic rebuild on deletion of internal nodes does not have a lot of maintenance functions, it could still be interesting to see the difference between a completely naive algorithm that is completely dependent on the rebuilding by the GB criteria and one that tries to reduce the problem to some extent on its own.

4.4 Counting of Distances and disk I/O

The number of distance computations are simply stored in a global variable, and each time the distance function is called it is incremented by one. This way the results will show the actual number of distance computations, not the theoretical number. Therefore it is very important that no more distance

computations are made than what is absolutely required. The algorithms have been tuned to reduce this number, but it could be that a few more are done than the minimum possible.

Because the tree structures are implemented in such a high level language as Python it is difficult to measure the number of disk accesses on a hardware level. Therefore a simpler scheme is implemented here. Just like with the distance computations a global counter is used. For each search the counter is reset and whenever a node is visited for the first time, this counter is incremented by one. When backtracking through an already visited node the counter is not incremented because it is presumed that the object is now in main memory as well as in secondary, and therefore no disk access has to be made.

As can be seen, a big fan-out is very useful because reading a node will include all the objects stored in it. In other words, the more objects stored in the node, the more objects will be read for "free" each time a node is read. When determining the fan-out in practice it is important to scale the number of nodes so that the node will stay paged. This means that the more complex the keys for the data objects are, the more space they will occupy, which again means that fewer objects can be stored in each node. The result is that as the number of dimensions grow, the smaller the fan-out will be.

This is not directly taken into account here, so the numbers serve more for testing than practical implementation. The maximum fan-out a node can have will be determined by the implementation and the properties of the data objects. For a discussion on about benchmarking of trees, see [PZB06].

4.5 Test Sets

The synthetic feature vectors used for the L_2 metric distance function all have values $0 \leq x \leq 1$ for each dimension. For the uniformly distributed test sets, all variables are simply given a value determined by the random function in Python for each dimension.

For the clustered data sets, a number of cluster centers are first randomly generated, where the number of cluster centers are given in advance. Then the actual data objects are created from these. This is done by randomly assigning each data object to one of the clusters, and then adding a small uniformly distributed variable between some range, like $[-1, 1]$. This method takes inspiration from [BO99].

The data sets used for the edit distance function is taken from the the International Ispell word list [Kue93]. Ispell is a spell-checker for Unix. The list used here is the is contained in the file 'english.3', and it contains 19708 different English words. The test sets are created by picking the preferred number of random words from this list, and making sure that no word is

picked more than once.

When doing insertion in the static trees, just one data set is generated and the tree is built from them. This is done a bit differently with the dynamic trees. First, an initial set of data objects are generated using the above method, and the trees are built using it, and stores a list containing all the objects in the trees. Then, there is a random chance for there being an insertion or a deletion. For insertion, a new data object is generated and inserted into all the trees, and this data object is then added to the list of the objects in the trees. When a deletion is done, a random sample is picked from the list of data objects in the trees and is then deleted from all the trees in the test. The object is also removed from the list. This is done a given number of times. It is important that every tree in the test will contain the same objects at any time.

4.6 Correctness

To make sure that the answers retrieved from the queries are indeed the the correct ones, every algorithm has been tested against linear scan algorithms for correctness, and are only used if they give the correct answer on every test. That means that every range search should return every value within range, and ever KNN-search should return the k nearest neighbors every time. As well as being thoroughly tested in advance, most tests done where the results are used in this thesis include a linear scan on every query to make sure that there are no errors. However, some of the more computationally heavy tests do not include this because of the extreme expense of doing a linear scan on the set of data objects for every query, so these algorithms have only been tested in advance on similar datasets.

Chapter 5

Results

5.1 Test Set

The test sets where created in the way described in Section 4.5. The results of the distance distributions for clustered data sets are shown in Figure 5.1 for 10 dimensions and 5.2 for 30 dimensions, while uniformly distributed are shown in Figure 5.3 and Figure 5.4. The slight skew is due to a rounding error.

The clustered data sets have to spikes. The one with the smallest mean distance is for the data objects within the cluster, while the one further away are for the rest of the objects. Notice how the distances converges toward the two means when the number of dimensions grows. This is related to the curse of dimensionality (Section 2.1.1), and makes it harder to prune out objects because the variance in distance decreases. This means that more objects will have to be checked directly against, because the constraints for using the parent-pivot distances (described in Section 2.5.1) will not work as well. See [CNBYM01] for a detailed discussion of this problem. Also note how the average distance grows along with the number of dimensions, which also leads to reduced pruning.

More or less the same applies for the uniformly distributed data objects, except that here there is only one spike because of the lack of clustered objects. Also notice how the average distance is longer for uniformly distributed data than for the clustered data when using the same number of dimensions. Just like with the change from 10 to 30 dimensions, this makes search harder, because this will lead to increased size of ball regions around the pivots which again leads to worse pruning.

If nothing else is mentioned, the sets used in the tests are clustered data sets in 10 dimensions, with 1000 initial elements and 20000 either insert or delete operations, where there is a 75% chance of insert and 25% chance of delete.

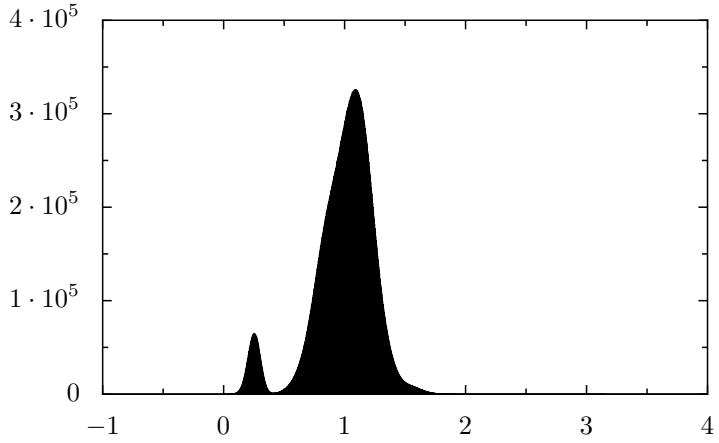


Figure 5.1: Distance distribution with 10 dimensions using 20 cluster centers.

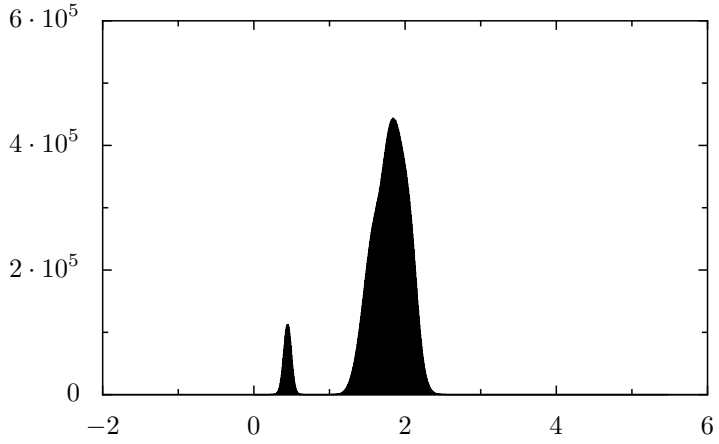


Figure 5.2: Distance distribution with 30 dimensions using 20 cluster centers.

5.2 Vantage Point Tree

The basic implementation of the VP-tree, described in [Uhl91], has a constant number of distance computations needed to build the tree for each set of objects. The implementation described in [Yia93], on the other side, has one major factor that determines how many distance computations are needed for building the tree, and that is the how many values you include with the getSamples() function. Because you have to find the median for each of the objects being picked by this function, there will be a lot of extra distance computations used for finding the best possible vantage point if a lot of samples are used. The higher the value, the better the chance of

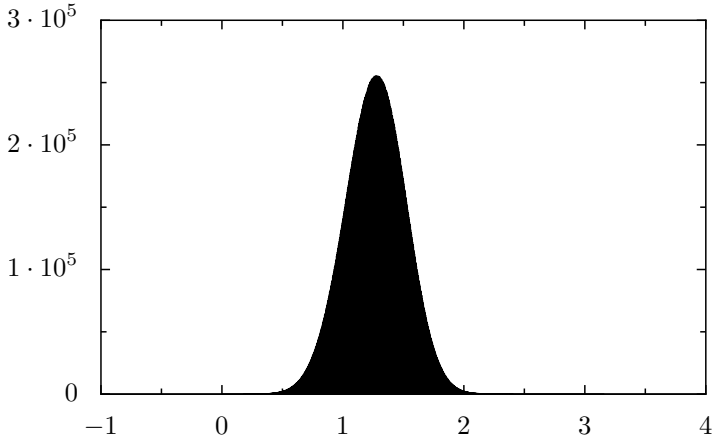


Figure 5.3: Distance distribution with 10 dimensions using uniformly distributed data objects.

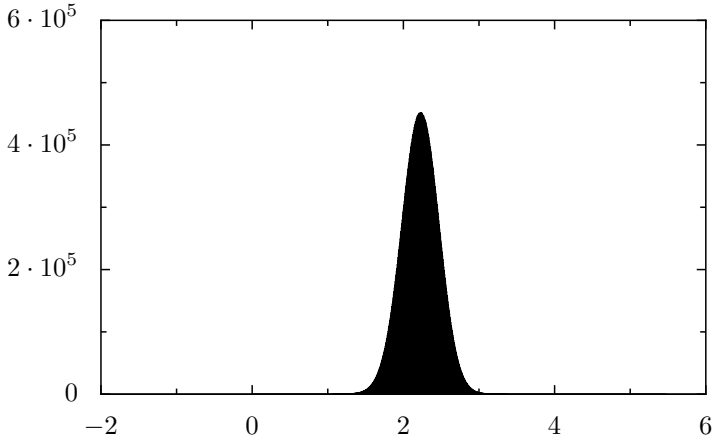
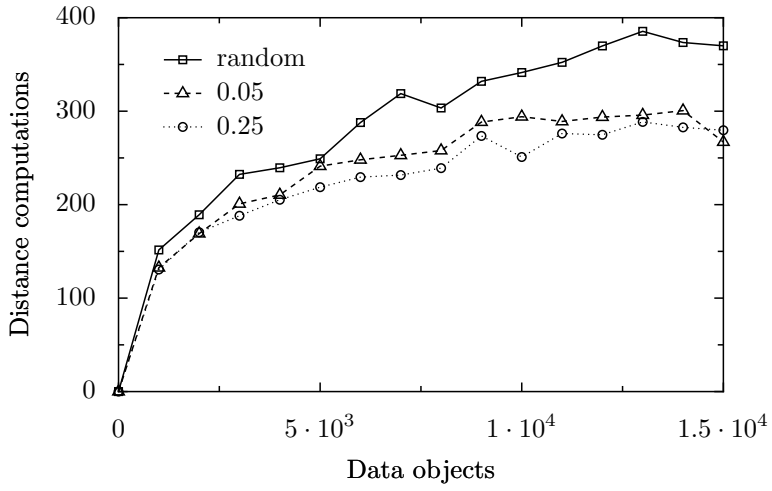


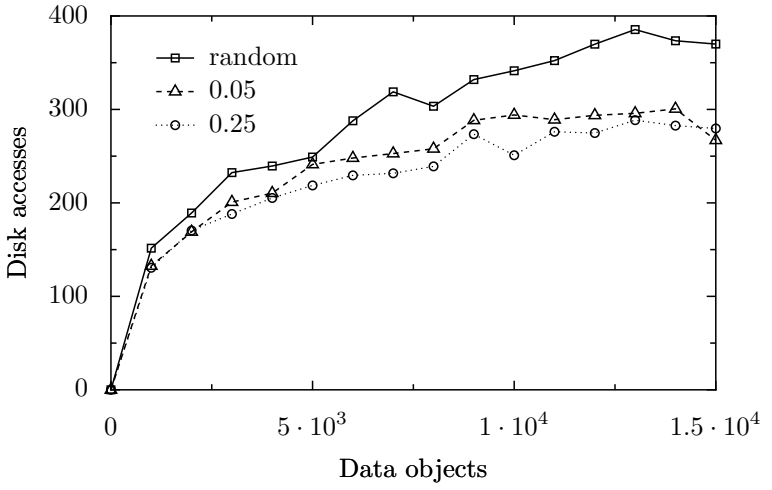
Figure 5.4: Distance distribution with 30 dimensions using uniformly distributed data objects.

finding the best vantage point, but the cost in building rises fast. Figures 5.5, 5.6, 5.7 and 5.8 shows the difference between random picking, 5% and 25%. Please note that, as this is ment more as a guidence for what is the best choice rather than a real comparison, these where not tested on the exact same datasets, just random datasets generated in the same way.

It is clear from Figures 5.5.1, 5.6.1, 5.7.1 and 5.8.1 that there are not much to save on distance computations when using a high sampling rate for finding a better vantage point than when just using the random method. And, considering that this is a VP-tree, the same is of course true for the



5.5.1: Distance computations

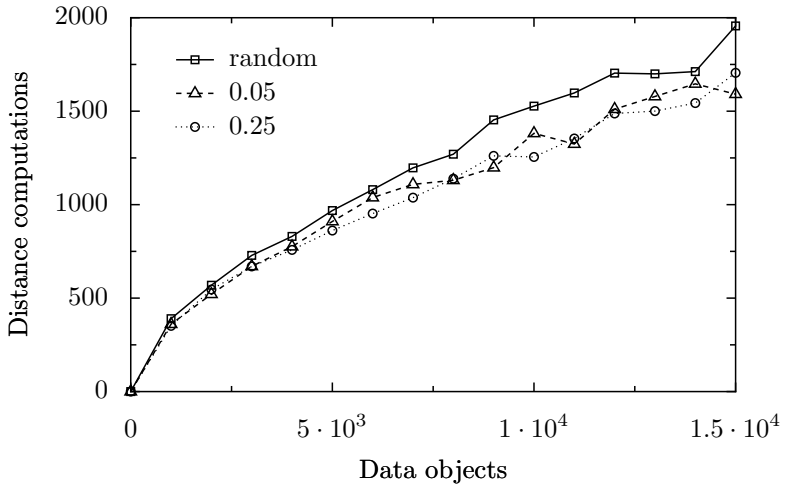


5.5.2: Disk IO

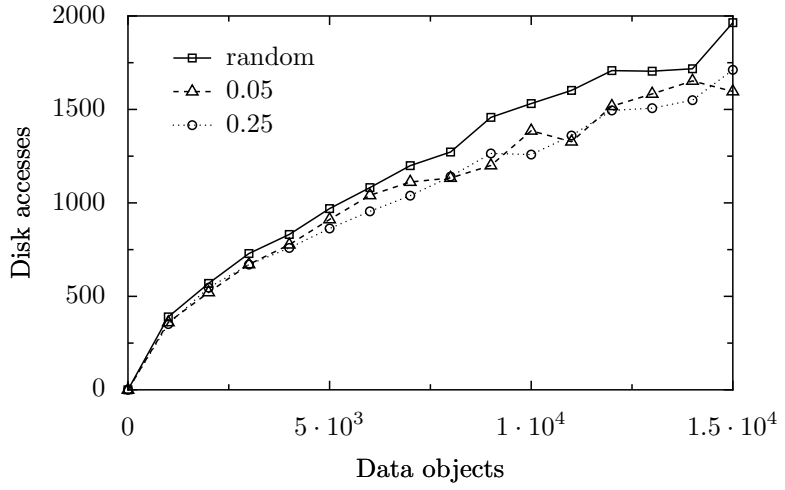
Figure 5.5: Range search in a VP-tree with a range of 0.1 in 10 dimensions.

number of disk accesses. This is probably because, at least on clustered data, there are very many good candidates to be chosen as good vantage points. Therefore, it does not really matter that much which one is chosen.

But one very interesting thing that can bee seen in Figure 5.7.1 and 5.8.1 is that a range of 0.3 performs much worse than 0.5 when it comes to distance computations, but much better when it comes to disk I/O. This is because of the Range-Pivot Distance Constraint (see Section 2.5.1 and Lemma 2.5.2), because, when the range is increased, the number of objects directly included is increased as well. While this does not make up for the number of nodes that must be searched between a range of 0.2 and 0.3, it



5.6.1: Distance computations

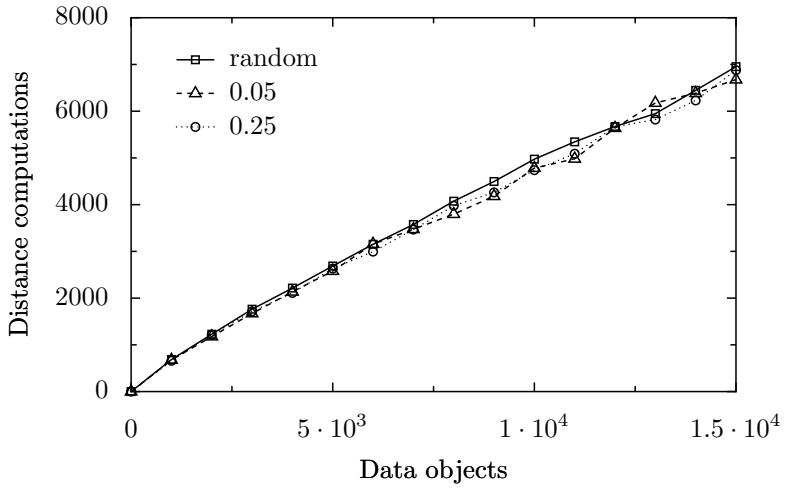


5.6.2: Disk IO

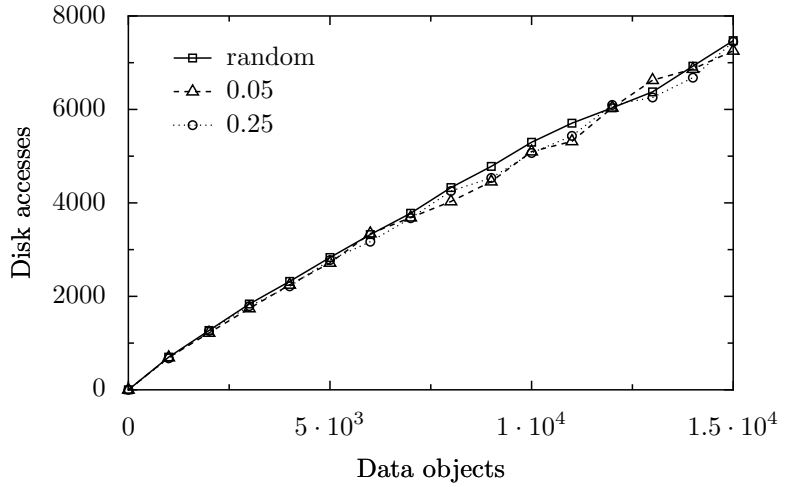
Figure 5.6: Range search in a VP-tree with a range of 0.2 in 10 dimensions.

certainly does between 0.3 and 0.5. However, as can be seen in Figure 5.8.2, a range of 0.5 finds more or less every single object, and it can be argued how interesting that is.

But what is really interesting here is the build cost, shown in Figure 5.9. Note first that the cost of building a VP-tree has nothing to do with the number of dimensions, which is because the list is just divided in two and no search for a vantage point is done. Therefore the curse of dimensionality has no effect on the build cost. But because the VP tree is going to be used in a dynamic data structure where partial and complete rebuilding is used a lot, the build time is very important in order to keep a functional dynamic



5.7.1: Distance computations



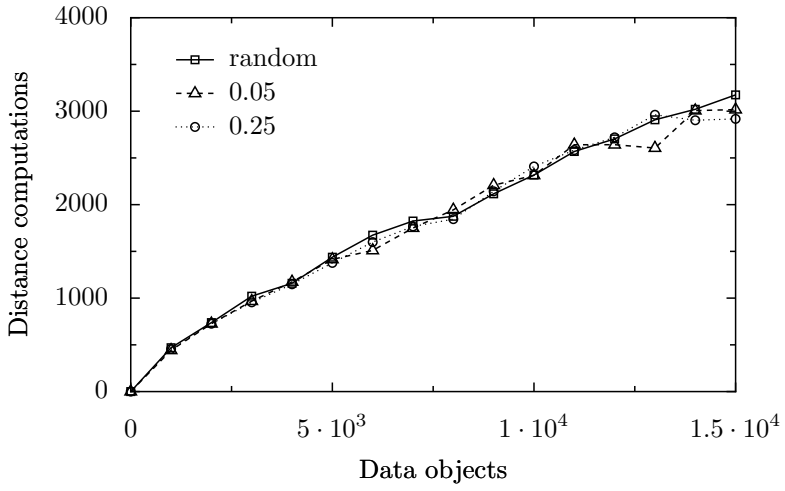
5.7.2: Disk IO

Figure 5.7: Range search in a VP-tree with a range of 0.3 in 10 dimensions.

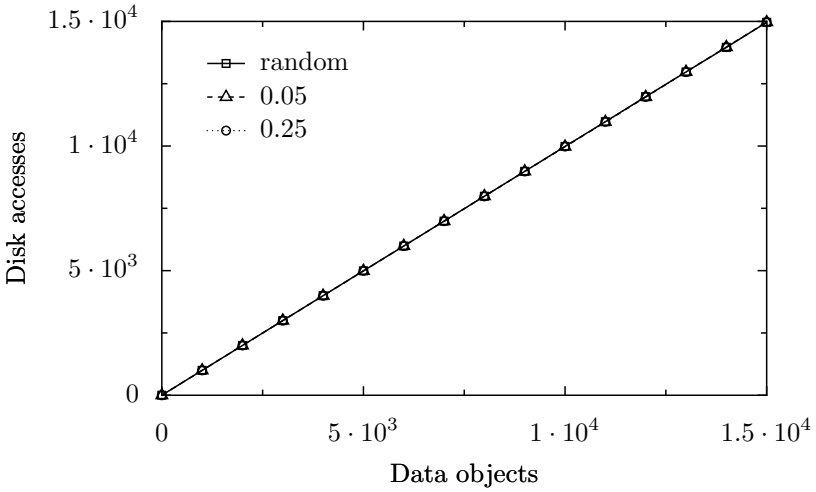
tree. And, as can be seen in Figure 5.9, the rise in build cost is enormous. It should therefore be safe to say that, considering that there is not even much gain in search time between the three, the cost in actually maintaining a tree is just too great to actually be of interest for the GBVP-tree. Therefore, from here on, only the random selection method will be used.

5.3 Multiple Vantage Point Tree

As mentioned in section 4.2.2, an important property of the MVP-tree is the ability to set the value k for how many data objects a leaf node will



5.8.1: Distance computations



5.8.2: Disk IO

Figure 5.8: Range search in a VP-tree with a range of 0.5 in 10 dimensions.

contain. While there are also options for increasing the fan-out of internal node, as described in Section 4.2.2, is not used here. Using the algorithm described in [BO97], the build cost is dependent on one thing, and that is the number used for k. This is, first of all, because k determines the height of the tree. The more objects that can be put in the buckets in the leaf nodes, the shorter the tree will be, and therefore less partitioning must be done. However, on the leaf level, the optimal second vantage point is found compared to the first vantage point, so there is a little extra cost there.

The trees in Figure 5.10 shows the differences between the build time with $k = 10$ and $k = 80$.

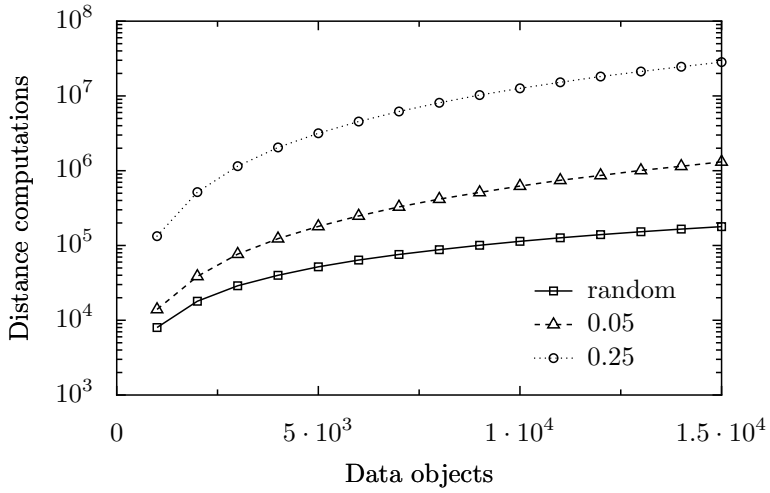


Figure 5.9: Build costs for building VP-trees.

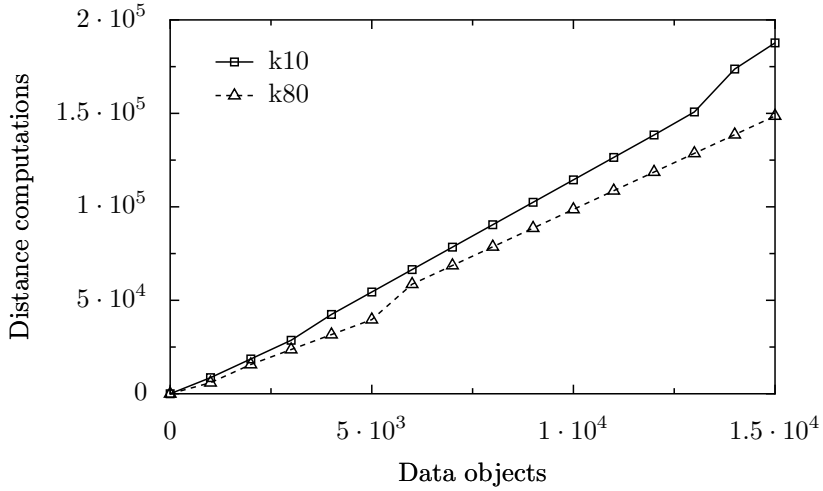


Figure 5.10: Building the tree with $k = 10$ and $k = 80$.

As can seen, the number of distance computations for building the tree is slightly better when using a higher value for k. This is because a high value for k means more objects in the leaves and therefore the height is lower. This means less distance computations used for dividing the data objects around pivot objects, and therefore less distance computations. This is partly made up for by the fact that it takes more distance computations to find the second pivot object in a leaf node than in an internal node, but as we can see, there still a slight difference.

Next up is the actual search for query objects. Figures 5.11, 5.12, 5.13

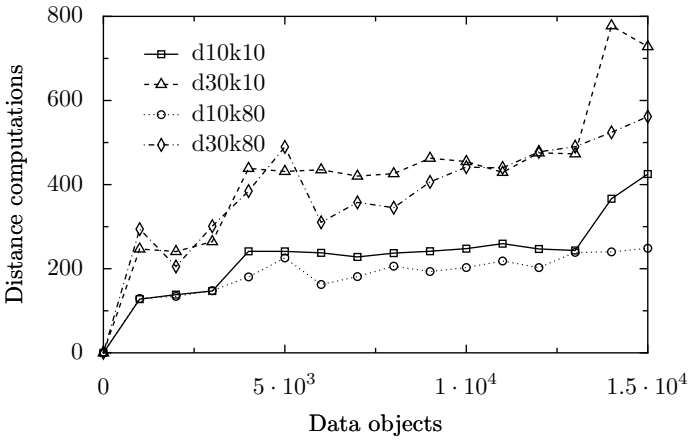
and 5.14 shows the range search for $k = 10$ and $k = 80$ and with the number of dimensions set to 10 and 30. Figure 5.15 shows the KNN-search for the same trees. Note again that the trees are not the same, and neither are the search objects, but they have been randomly generated by the same method, and the method described in Section 3.4.5 have been used to minimize the random errors.

The results for a range of 0.1, as can be seen in Figure 5.11, are not very interesting, because, as can be seen in Figure 5.11.3, hardly any objects are found at all. What is worth noting, though, is that for 30 dimensions the number of distance computations are quite a bit worse even when no objects are found. But as the number of disk I/O stays the same, this is probably because it is easier to filter out the leaf nodes using pivot filtering (Lemma 2.5.5) when the number of dimensions is small.

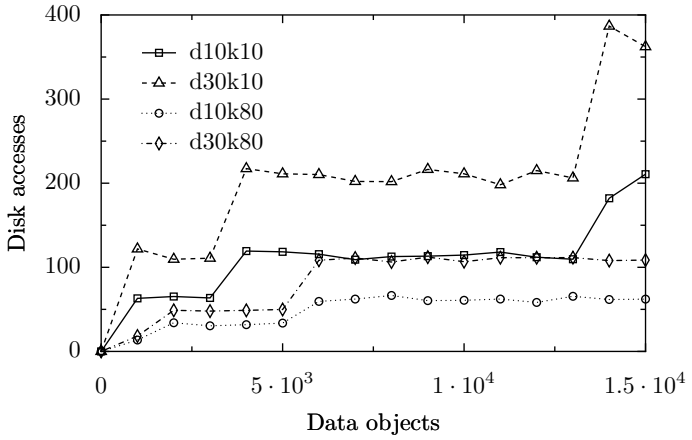
The results for a range of 0.2 (Figure 5.12) can be a bit surprising at first, because for both distance computations (Figure 5.12.1), and to a lesser extent, disk I/O (Figure 5.12.2), the tree using 10 dimensions performs worse than the tree using 10 dimensions. But the explanation can be seen in Figure 5.12.3, as the tree using 30 dimensions simply finds no objects here either, which probably results in fewer sub-trees being investigated because of filtering higher up in the tree. It is also becoming easy to see how a high value for k can have an impact on disk I/O, as the higher number of objects in the leaf nodes makes for few nodes that have to be accessed. A high value for k could also have an impact on the number of disc accesses, but it is very minimal.

The results for a range of 0.3 (Figure 5.13) are a bit different from that found with a range of 0.2. The number of distance computations, as seen in Figure 5.13.1, is still very similar between a k of 10 and 80, but now also the number of distance computations for 30 dimensions is actually higher for that of 10, even though the number of objects found (Figure 5.13.3) is still very small for 30 dimensions with compared with 10, so it is evident that the curse of dimensionality (Section 2.1.1) is beginning to show up. It is also worth mentioning that $k = 10$ is now actually just edging out $k = 80$ when it gets to distance computations, which could be because more objects are directly included from the pivot nodes because of the higher range. But, as seen in Figure 5.13.2, the number of disk I/O is still quite a bit smaller for $k = 80$ than for $k = 10$, again due to the fact that fewer nodes have to be accessed.

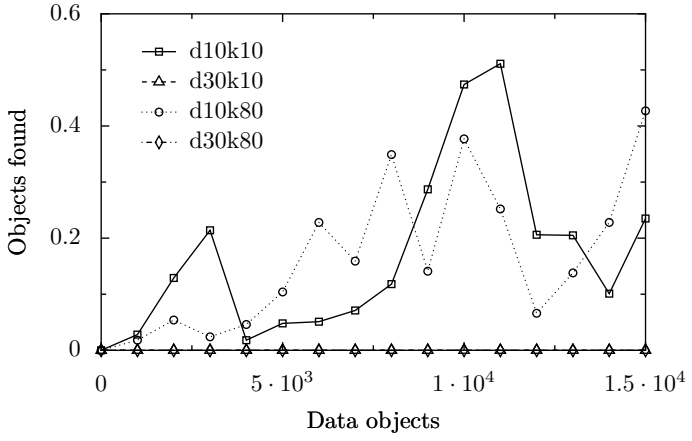
As can be seen in Figure 5.14, the trend continues. The number of distance computations (Figure 5.14.1 are now a lot smaller because of the higher automatic inclusion in internal nodes because of Lemma 2.5.2. Because more or less every node is included in the search (Figure 5.14.3), the curse of dimensionality does not show up here, as every node has to be accessed or directly included anyways. But the difference in disk I/O (Figure 5.14.2) becomes enormous when the number of objects gets close to 15000.



5.11.1: Distance computations

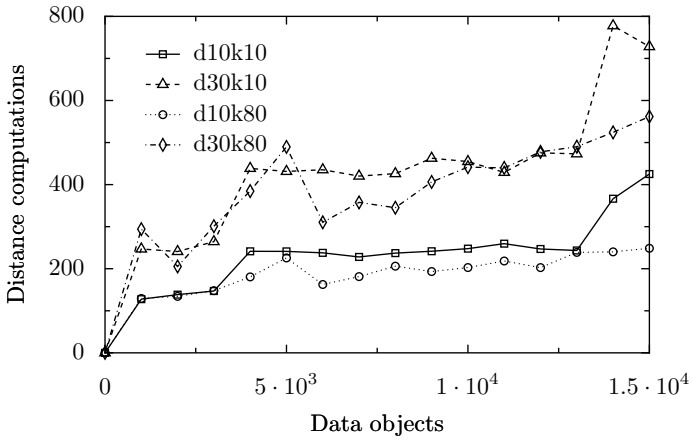


5.11.2: Disk I/O

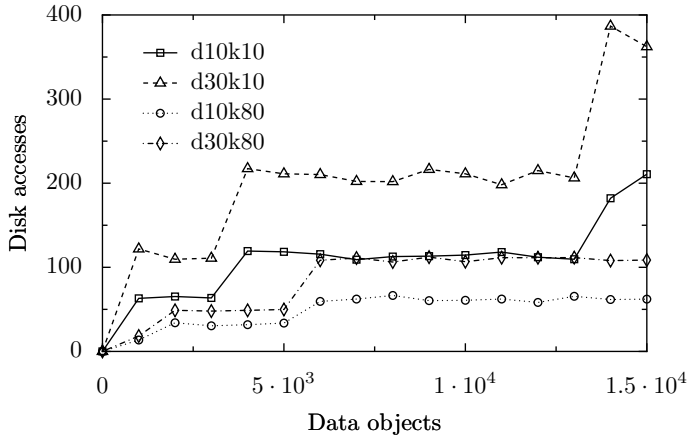


5.11.3: Objects found

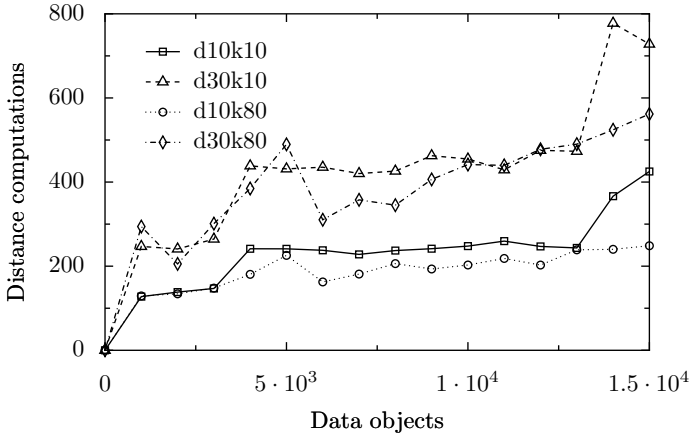
Figure 5.11: Range search in MVP-trees with $k = 10$ and $k = 80$ in 10 and 30 dimensions with range 0.1.



5.12.1: Distance computations

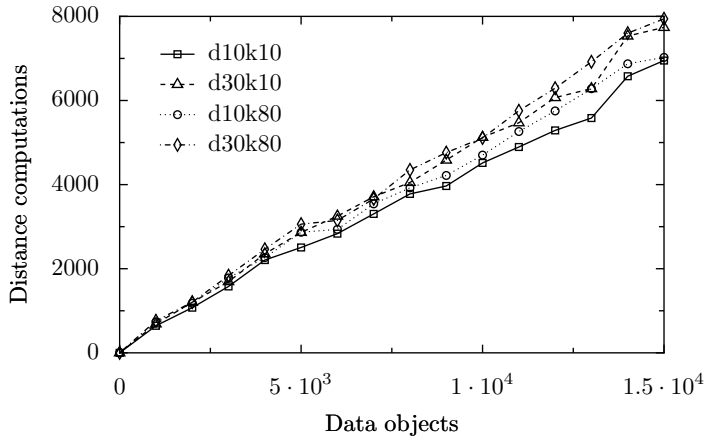


5.12.2: Disk I/O

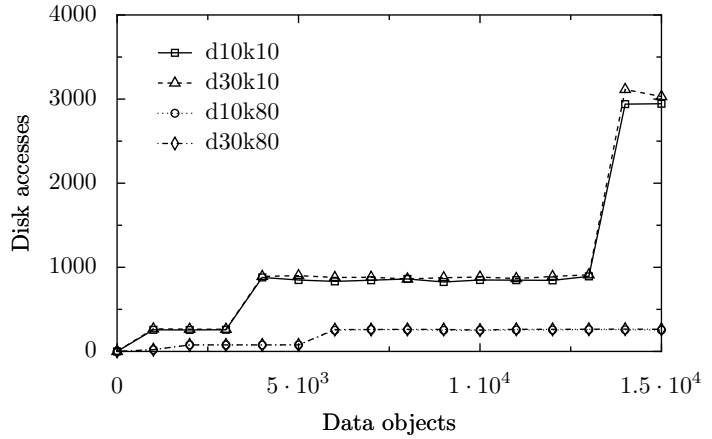


5.12.3: Objects found

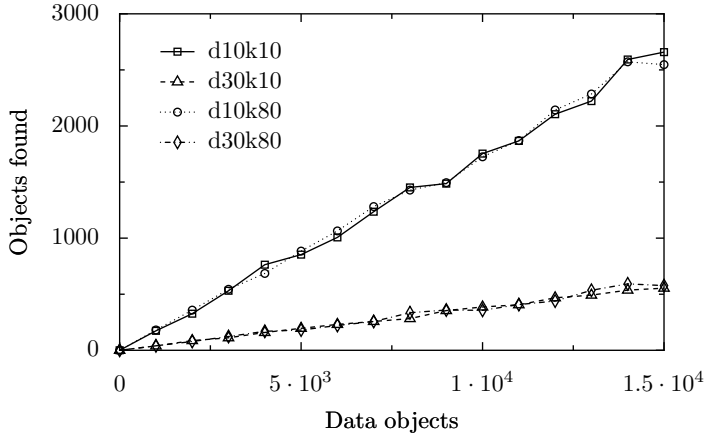
Figure 5.12: Range search in MVP-trees with $k = 10$ and $k = 80$ in 10 and 30 dimensions with range 0.2.



5.13.1: Distance computations

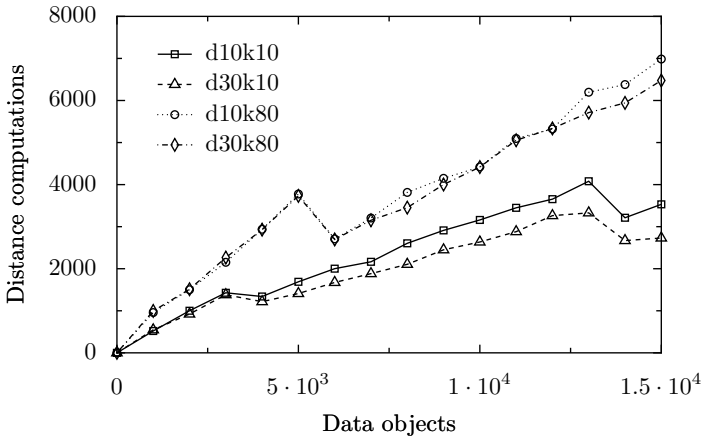


5.13.2: Disk I/O

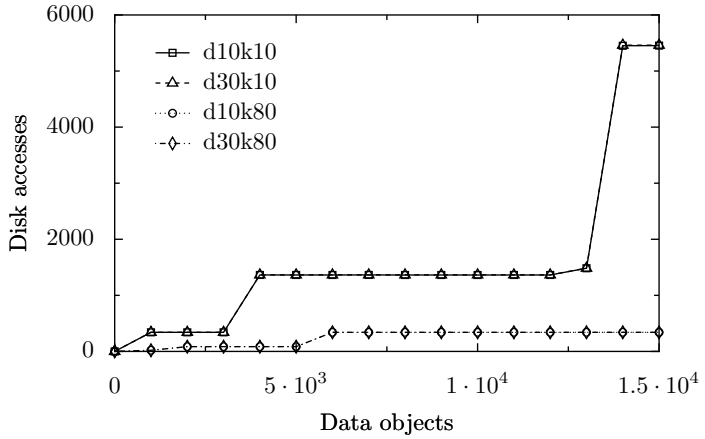


5.13.3: Objects found

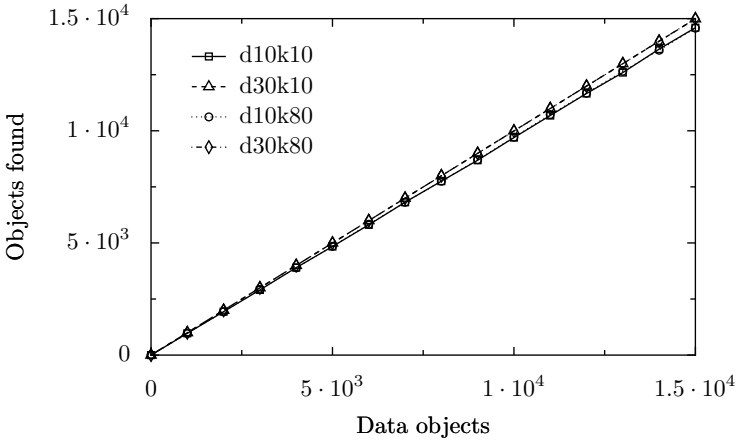
Figure 5.13: Range search in MVP-trees with $k = 10$ and $k = 80$ in 10 and 30 dimensions with range 0.3.



5.14.1: Distance computations

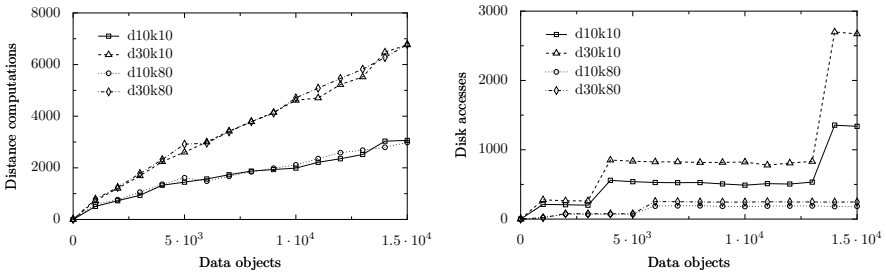


5.14.2: Disk I/O



5.14.3: Objects found

Figure 5.14: Range search in MVP-trees with $k = 10$ and $k = 80$ in 10 and 30 dimensions with range 0.5.



5.15.1: Distance computations

5.15.2: Disk I/O

Figure 5.15: KNN-search in MVP-trees with $k = 10$ and $k = 80$ in 10 and 30 dimensions with a KNN value of 10.

The gigantic spike is because the number of nodes needed for a new level in the tree is reached. Having 80 objects in the leaf means that the tree will fill up a lot slower, and therefore the spike will happen a lot later. In other words, this difference should just grow and grow as the number of objects in the tree gets larger.

When looking at distance computations for the KNN-search (Figure 5.15), the curse of dimensionality becomes quickly apparent, as both the tests running on 30 dimensions performs much worse than both the ones on 10 dimensions. Interestingly, the number of child nodes, being it $k = 10$ or $k = 80$, has nothing to say at all. This probably means that it converges toward the a close approximation of the best number of matches pretty fast. It is also clear that the difference between 10 and 30 dimensions will grow even more as the number of data objects grows. Sadly, this means that the MVP does not scale very well when the number of dimensions grow, and the number of distance computations needed will probably grow close to a linear scan if the number of dimensions increases further. However, when looking at disk I/O (Figure 5.15.2), it is clear that the number of objects a child node can include have a huge impact on the number of disk accesses needed for a KNN-search. It is also clear that the spikes are worse for 30 dimensions than they are for 10, so there can be raised questions about how well the MVP-tree scales in this area as well.

The results of the tests are clearly that a higher value for k is a good thing, as more data objects included with the child nodes leads to far less disk I/O, while the number of distance computations is more or less equal for anything but really large search ranges. However, there can be raised severe questions about how well the MVP-tree scales as the number of dimensions rises. While a higher k will delay the inevitable rise in disk I/O needed for searches, this will sooner or later rise as well.

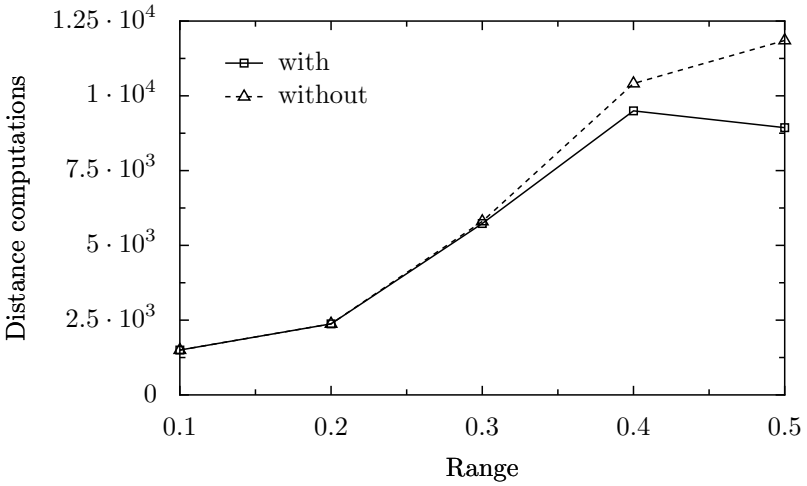
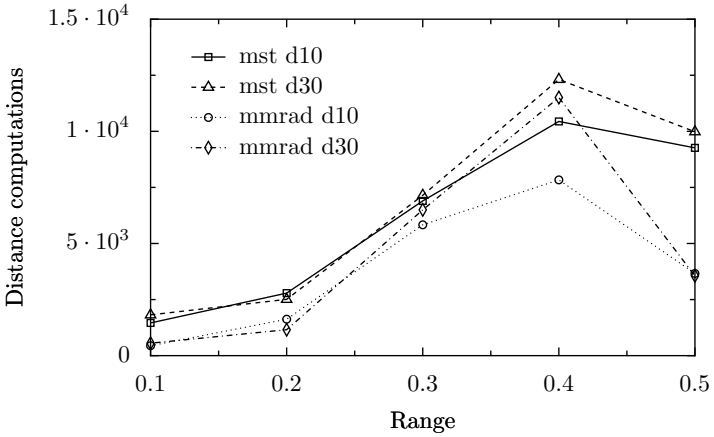


Figure 5.16: Range search in a M-tree showing the effect of the automatic inclusion from the Pivot-Range Distance Constraint.

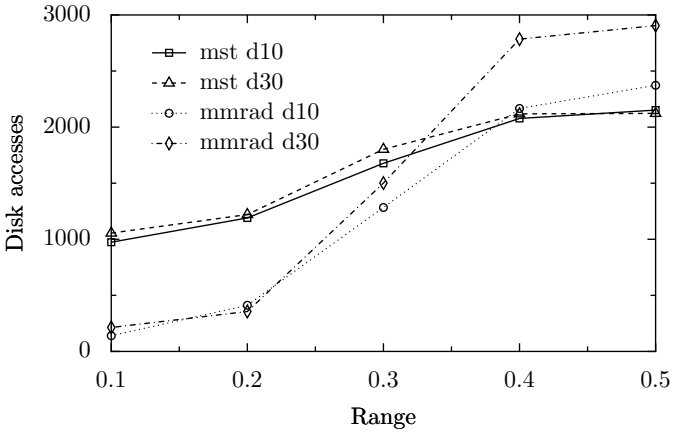
5.4 M-tree

The Pivot-Range Distance Constraint using Lemma 2.5.2, as described in Section 4.2.2, was implemented for the M-tree. To see if it has any noticeable effect at all on the M-tree, a test was performed to compare the the methods against each other. This is done on the exact same trees, but using the automatic inclusion in one of the range searches and not in the other. The effect can be seen in Figure 5.16, and, while it is clear that it has not exactly got an extreme effect on the performance of the M-tree, it does improve the search by a noticeable amount when the search radius gets large. And as the M-tree is not exactly known to scale too well when the range gets larger (see [PZB06] for some results), the inclusion of Lemma 2.5.2 does at least make the M-tree a bit more competitive in this area.

In section 4.2.2 it was also described that several different promote and partition algorithms where implemented for the M-tree. As the random selection of pivot objects during promotion did very badly on the initial tests, the algorithm was dropped for the actual tests shown here. The mMRAD algorithm and the MST algorithm where however used, and the results of range search using the algorithms on clustered data can be seen in Figure 5.17, while the results from the building and KNN-search can be seen in Table 5.1 and 5.2. As can be seen in Figure 5.17.2, the mMRAD partitioning algorithm does indeed partition the elements quite a bit better than the MST algorithm when it gets to distance computations, while disk I/O (Figure 5.17.2 is more even, with mMRAD being better on lower ranges and the MST better on higher. The results from the uniformly distributed data is shown in the appendix (Figure A.1), and even if the difference is not as



5.17.1: Distance computations



5.17.2: Disk I/O

Figure 5.17: Range search in a M-tree using the MST and mMRAD promotion algorithms in 10 and 30 dimensions on clustered data using a fan-out of 10.

clear there, it is still notable. KNN-search was also notably better for the mMRAD algorithm, both in distance computations and disk I/O. However, the build cost is surprisingly lower for the MST algorithm, which could be partly explained by the lower average tree height. This means that the MST probably creates a more balanced tree than the mMRAD algorithm, but, as [CPZ97] showed, balanced is not always better. While it is hard to say which one is best, the mMRAD algorithm was chosen for use in the rest of the tests because of the better search results.

To see how the fan-out of the M-tree have an impact on search performance, a test was done using different fan-outs. Surprisingly, using a high fan-out provided extremely slow running time, which prevented the test from

Algorithm	Dimensions	Distance computations	Height
MST	10	16136566	5.0
MST	30	15779262	5.0
mMRAD	10	17447868	5.1
mMRAD	30	22913110	6.0

Table 5.1: Building cost of M-trees using the MST and mMRAD algorithms using a fan-out of 10.

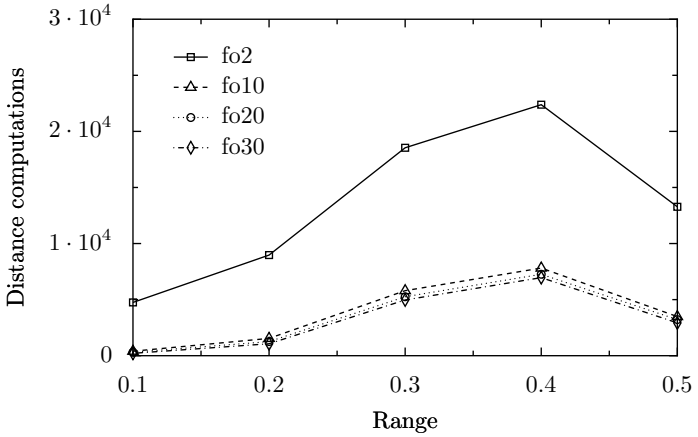
Algorithm	Dimensions	Distance computations	Disk I/O
MST	10	3523.633	1289.366
MST	30	6259.925	1725.570
mMRAD	10	2618.120	617.809
mMRAD	30	5694.433	1323.319

Table 5.2: KNN-search in M-trees using the MST and mMRAD algorithms using a fan-out of 10.

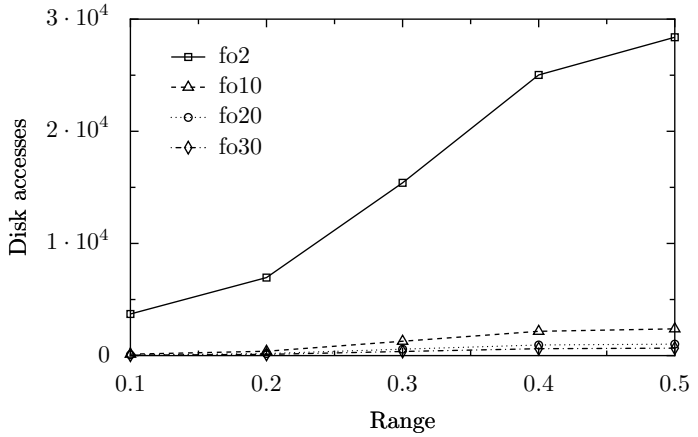
using high values on any meaningful number of data objects, but the results seems to indicate some convergence, so it may not have a huge impact. But a fan-out of 2, 10, 20 and 30 was tested, and the results of the range search for 10 dimensions are shown in Figure 5.18, while the build cost are shown in Table 5.3 and the KNN-search in Table 5.4. Not surprisingly, a fan-out of only 2 did far worst, both in distance computations (Figure 5.18.1) and disk I/O (Figure 5.18.2), while 20 does a bit better than 10 and 30 a little bit better than 20. Surprisingly, a fan-out of 30 uses a few more distance computation than a fan-out on 20 on the KNN-search, but it is notably better in regards of disk I/O. The build costs are also lower for a higher fan-out. The conclusion is that a higher fan-out, at least as high as has been tested here, is indeed a good thing. But, considering how small the improvements between a fan-out of 20 and 30 are, this probably converges. However, there could be a more to gain when the fan-out is high enough for the height of the tree to be reduced by one more often than with a fan-out of 30, which with its average height of 3.9 is only 0.1 better than a fan-out of 20.

5.5 General Balanced Vantage Point Tree

As the General Balanced Vantage Point Tree (GBVP-tree) is a dynamic structure, the main point was to try to test its dynamic capabilities. As it was already concluded in Section 5.2 that only the randomized version would be practical a dynamic structure, it is the only one used here. The main test method was to first bulk load the tree with some initial elements,



5.18.1: Distance computations



5.18.2: Disk I/O

Figure 5.18: Range search in a M-tree for different fan-outs in 10 dimensions on clustered data.

Fan-out	Distance computations	Height
2	155527828	23.3
10	17444112	5.3
20	11859002	4.0
30	11796590	3.9

Table 5.3: Building costs for M-trees using different fan-outs on clustered data in 10 dimensions.

Fan-out	Distance computations	Disk I/O
2	10632.929	8132.106
10	2524.599	594.749
20	2199.902	276.510
30	2228.295	187.549

Table 5.4: KNN-search in M-trees using different fan-outs on clustered data in 10 dimensions.

before starting to insert and delete objects as described in Section 4.5. After a given number of insertions and deletions, range search and KNN-search is performed on the tree.

First of all, some experimentation with different values of k was done. While this should give more or less the same results as the tests on the MVP-tree (Section 5.3),

Some experimentations was also done with the b and c values that influences when a GB-tree is rebuilt. Results where 75% insertions and 25% deletetions are done are shown in Figures 5.19 and 5.20. Table 5.7 shows the same results in a more readable format, while Table 5.5 shows the build costs and heights and Table 5.6 shows the KNN-search. Determening what values should be given to b and c is not trivial, especially c, as a high value for c will mean many complete rebuilds but very few partial, while a low value will give the opposite effeft. As b only influences on the number of total rebuilds it is easier to dermine the value, but without some experimentation it is hard to say what is the best balance between total and partial rebuilds. Please not that the automatic inclusion of subnodes that have to be within range was turned off for this particular test. While this shouldn not have any impact on the results in this test, this means the results here should not be directly compared against any other test in this thesis. This also means that the values for distance computations and disk I/O will be the same, and therefore only one is shown.

As can be seen from Figure 5.19, changing the value of b has got a slight effect, but not by much. As seen in Table 5.7, it is the delete method using marking that suffers the most as the value for b gets larger, even though, as Table 5.5 shows, the numbers of partial rebuildings due to the GBT criteria are far higher than for the one using rebuilding. But the fact that the ones using rebuilding still have a lot more distance computations than suggests that a lot of rebuilding is done because of the deletion of internal nodes, and this in the end leads to both a better result but at the cost of an increased number of distance computations during building. However, both methods still use a large number of partial rebuildings, while hardly any total rebuildings, which is because the value for c is so low. A larger value for c would probably lead to a worse result for the mark method.

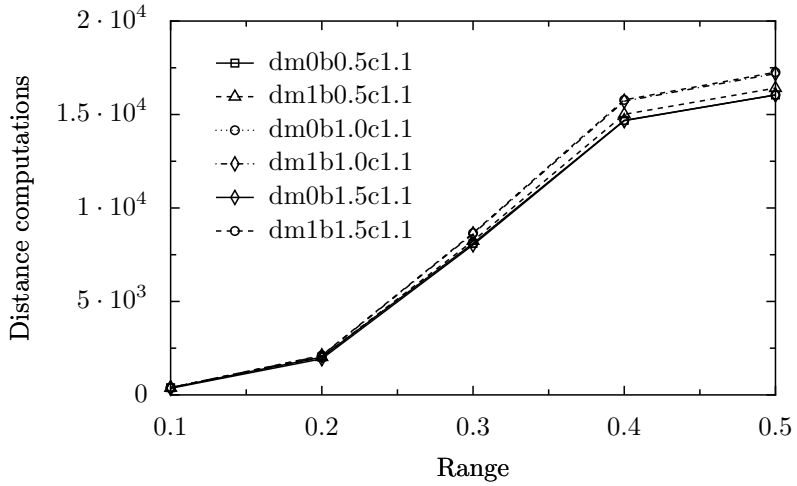


Figure 5.19: Range search in GBVP-trees with $c = 1.1$ and different values for b. Initial number of objects was 1000 and 30000 was inserted/deleted.

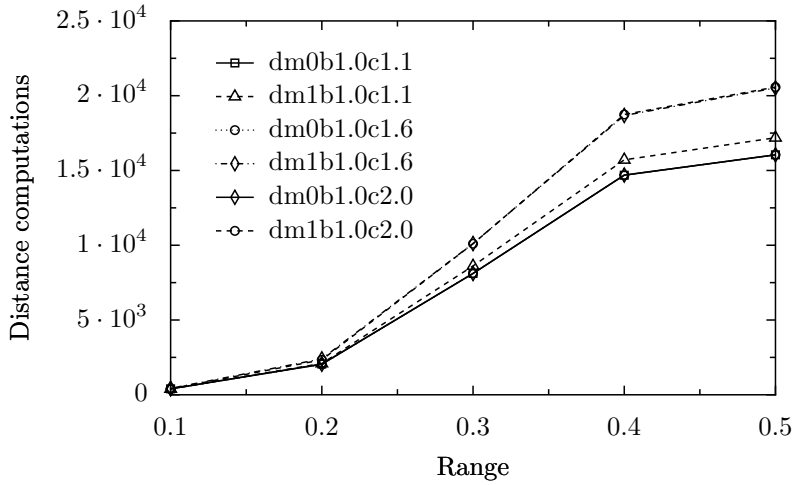


Figure 5.20: Range search in GBVP-trees with $b = 1.0$ and values 1.1, 1.6 and 2.0 for c. Initial number of objects was 1000 and 30000 was inserted/deleted.

More or less the same is true for Figure 5.20, where there, just like with the changes of b, there is just a slight difference between the different versions. However, there is a larger gap here, as the fact that as c gets no partial rebuilding due to the GBT-criteria is done. As can be seen in Table 5.5, the height difference between trees using the two different delete methods gets much larger when the partial rebuildings stop. This shows much better how the GBT rebuilding influences the trees than before, as it

DM	b	c	Dist comp	Tot reb	Part reb	Height
Rebuilding	0.5	1.1	3613100	1.6	766.3	16.0
Mark	0.5	1.1	3568020	1.2	1426.0	16.0
Rebuild	1.0	1.1	2301954	0.0	877.9	16.0
Mark	1.0	1.1	1768444	0.0	1962.1	16.6
Rebuild	1.5	1.1	2243548	0.0	890.4	16.0
Mark	1.5	1.1	1695818	0.0	1944.8	16.7
Rebuild	1.0	1.6	1710514	0.0	0.1	20.4
Mark	1.0	1.6	906040	0.0	29.7	24.0
Rebuild	1.0	2.0	2004332	1.0	0.0	19.7
Mark	1.0	2.0	902922	0.0	0.0	27.9

Table 5.5: Building cost of GBVP-trees with different values for b and c. Initial number of objects was 1000 and 30000 was inserted/deleted.

DM	b	c	Dist Comp/Disk I/O
Rebuilding	0.5	1.1	3756.714
Mark	0.5	1.1	3671.471
Rebuild	1.0	1.1	3628.518
Mark	1.0	1.1	3901.629
Rebuild	1.5	1.1	3553.222
Mark	1.5	1.1	3913.219
Rebuild	1.0	1.6	3587.378
Mark	1.0	1.6	4592.849
Rebuild	1.0	2.0	3763.263
Mark	1.0	2.0	4527.828

Table 5.6: KNN-search in GBVP-trees with different values for b and c. Initial number of objects was 1000 and 30000 was inserted/deleted.

is clear that performance in the tree using marking decreases. However, it is also indicated just how much the rebuilding due to internal deletion can say for a tree, as the the ones using that delete method does not grow much in height at all. The cost difference of building the trees also gets much larger, and both are expectedly less expensive to build when GBT rebuilding no longer happens. The fact that the trees using rebuilding on delete performs just as well on the average range search when no GBT rebuilding happens is quite interesting when considering the fact that it is less expensive to build. The average height is a bit higher, but it does not seem to matter much. As usual, the KNN-search follows more or less the exact same trend as the range search did, with worse results for the trees using marking for larger values of c.

DM	b	c	r0.1	r0.2	r0.3	r0.4	r0.5
Rebuild	0.5	1.1	383.7	2021.9	8105.1	14686.7	16045.9
Mark	0.5	1.1	376.1	2022.5	8239.0	15008.9	16413.7
Rebuild	1.0	1.1	397.6	2056.2	8115.1	14682.6	16046.3
Mark	1.0	1.1	391.3	2093.7	8615.1	15711.6	17187.9
Rebuild	1.5	1.1	353.2	1932.7	8025.1	14671.3	16046.7
Mark	1.5	1.1	385.4	2115.1	8657.4	15781.7	17261.7
Rebuild	1.0	1.6	381.8	2033.7	8112.9	14681.9	16047.0
Mark	1.0	1.6	428.5	2403.0	10114.4	18677.6	20517.6
Rebuild	1.0	2.0	397.3	2039.4	8112.1	14692.1	16046.3
Mark	1.0	2.0	401.2	2335.4	10100.8	18742.6	20575.7

Table 5.7: Range search showing average values in GBVP-trees with different values for b and c. The ranges are from 0.1-0.5 (marked r0.1-r0.5). Initial number of objects was 1000 and 30000 was inserted/deleted.

DM	b	c	r0.1	r0.2	r0.3	r0.4	r0.5
Rebuild	0.5	1.1	1642	8518	15956	16318	16318
Mark	0.5	1.1	1712	8736	17294	17729	17729
Rebuild	1.0	1.1	1873	8645	15914	16318	16318
Mark	1.0	1.1	1747	9678	17871	18226	18226
Rebuild	1.5	1.1	1415	8594	15921	16318	16318
Mark	1.5	1.1	1658	9827	17866	18248	18248
Rebuild	1.0	1.6	1607	8814	15942	16318	16318
Mark	1.0	1.6	2089	10851	20276	20761	20761
Rebuild	1.0	2.0	1669	8857	15847	16318	16318
Mark	1.0	2.0	1772	10757	20243	20796	20796

Table 5.8: Range search showing the worst values in GBVP-trees with different values for b and c. The ranges are from 0.1-0.5 (marked r0.1-r0.5). Initial number of objects was 1000 and 30000 was inserted/deleted.

The worst case results of the same test is given in Table 5.8. This shows more or less the same results as the ones for the average case, but again it is interesting to see how close the trees with rebuilding on deletion of internal nodes do when the value for c grows. It is also interesting to see how fast they converge into an upper bound, which makes them even more even. This do not happen for the trees using marking, where the worst case gets worse along with the average results.

To get a closer look at how much the delete algorithm influences the results, a test where only insertions were done was made. The results of the building is shown in Table 5.9, while the results of the range search is

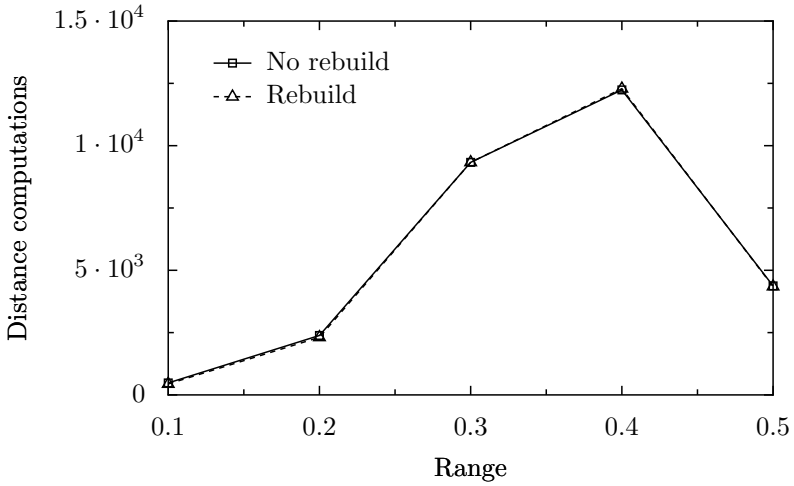


Figure 5.21: Range search in a GBVP-tree with $c = 1.2$ and values 1.0, 1.5 and 2.0 for b. Initial number of objects was 1000 and 20000 was inserted/deleted.

Balanced	Distance computations	Total rb	Partial rb	Height
No	628464	0.0	0.0	27.7
Yes	914212	0.0	802.2	18.0

Table 5.9: Building cost of the GBVP-tree with and without rebuilding using only insertions.

shown in Figure 5.21 and KNN-search in Table 5.10. As can be seen from the results, the difference in height is there indeed, with the average of the maximum height of the trees shrinking from 27.7 to 18.0 when rebuilding is turned on. However, when looking at the average number of distance computations, there is not any difference at all. However, the KNN-search does improve a bit for the balanced tree compared to the unbalanced.

The result of the tests is that the trees are indeed more balanced when using the GBT criteria. However, the results are not overly positive for the GBVP-tree when it actually comes to query results, as there is hardly any

Balanced	Distance computatations
No	4596.066
Yes	4196.411

Table 5.10: KNN-search of the GBVP-tree with and without rebuilding using only insertions.

k	Distance computations	Total rebuilds	Partial rebuilds	Height
2	1254886	0.0	0.6	16.2
10	1232240	0.5	0.0	14.0
80	1819274	12.1	0.0	10.0

Table 5.11: Building cost of GBMVP-trees with different values for k with 1000 initial objects and 20000 insert/delete operations.

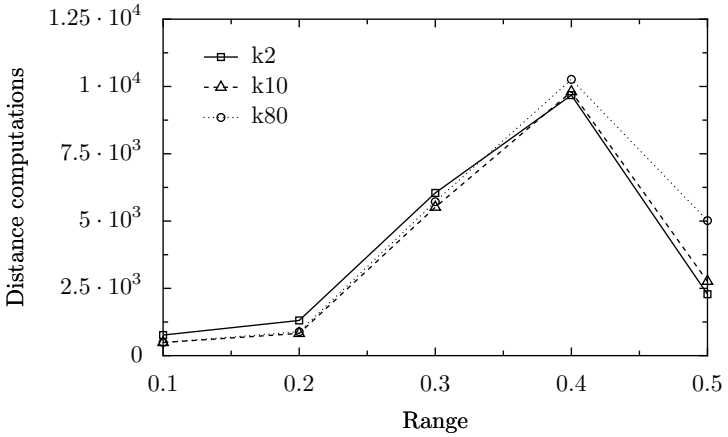
improvement at all. This was especially concerning for the last test, where the trees using no rebuilding at all performs just as well as a trees using it.

5.6 General Balanced Multiple Vantage Point Tree

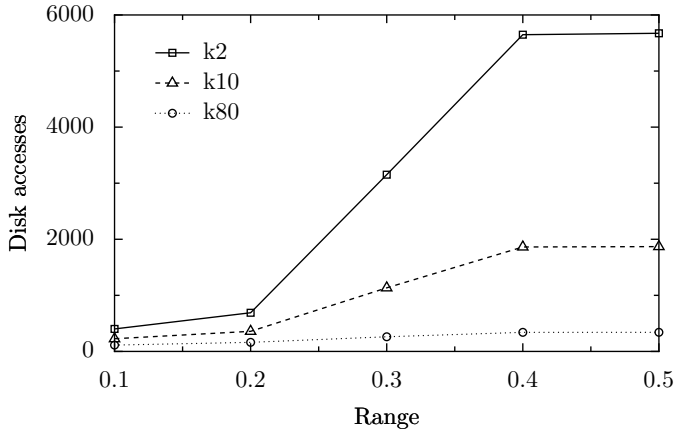
There is one initial problem with testing the GBMVP-tree, and that is setting the right value for k. Because, if the number for k is large, it means that a lot of objects are in each leaf node, and therefore the height of the tree grows slower. That, in turn, means less use for partial rebuilds. Table 5.11 shows the build costs for building trees with $k = 2$, $k = 10$ and $k = 80$, as well as information about rebuildings and the heights of the trees. The number of distance computations and disk I/O using range search can be seen in Figure 5.22. For this test was done on clustered data, and used $b = 1.0$ and $c = 1.2$.

As can be seen in Figure 5.22.1, the number of distance computations for the various values of k are very similar. $k = 80$ is quite a bit worse when the range is high, which is probably due to the lower height giving less use for the Object-Pivot Constraint, and $k = 2$ is slightly worse when the range is low. However, as can be seen in Figure 5.22.2, the number of disk accesses varies wildly, with $k = 80$ having the lowest number of disc accesses by far, but this should not be surprising considering the results in Section 5.3.

While the figures in themselves do not give any particular new information, it is interesting when looking at the build costs in table 5.11. As can be seen, neither $k = 2$ or $k = 10$ used many rebuilds at all, and therefore gained quite similar results as what can be seen in Section 5.3, where a higher value for k gives a slight boost in build efficiency. $k = 80$, on the other side, used many total rebuilds, which resulted in a very high average build cost. This is probably because the tree grows so slow that the number of deletions gets very high compared to the number of free children in the tree. This will lead the formulas described in Section 2.8 to initiate a complete rebuild faster if the growth of the tree is slow, although the number of partial rebuilds will go down. This is an unfortunate property with the GBMVP-tree in this regard, because it means it is more difficult to adapt a MVP-tree to the GBT paradigm than a VP-tree. However, the fact that the increase in build



5.22.1: Distance computations



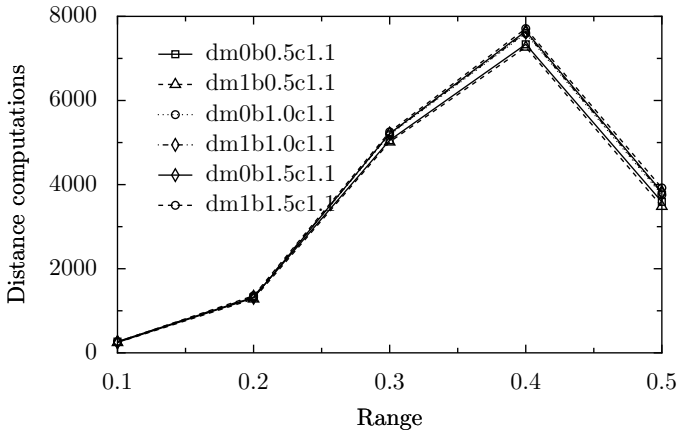
5.22.2: Disk I/O

Figure 5.22: Range search in GBMVP-trees with $k = 2$, $k = 10$ and $k = 80$ in 30 dimensions with range $0.1 - 0.5$.

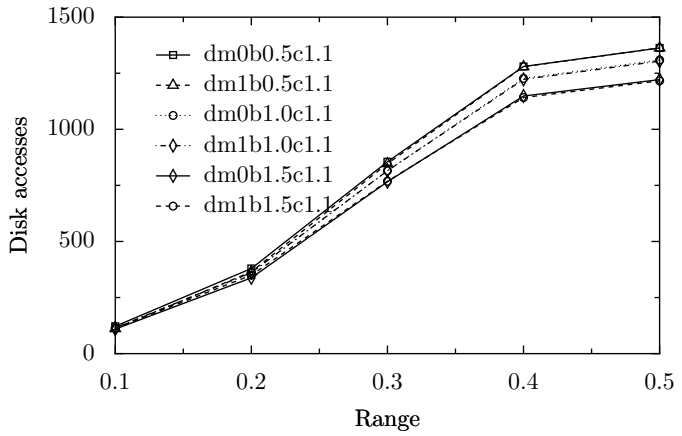
time did not give it a better average search time was not a good sign for the GBMVP-tree.

A similar test was done on 100000 samples on both clustered and uniformly distributed data, and the results are shown in Section A.2.2. In this test, the higher values for k gave a bit more gain than when only using 20000, suggesting that the difference will only grow as the number of objects do. The two delete methods where also tested, but turned out to perform more or less equal.

One interesting question is if the lack of improved search results when regarding distance computations was because of the different values for k or because the slight difference in height does not matter that much. Just



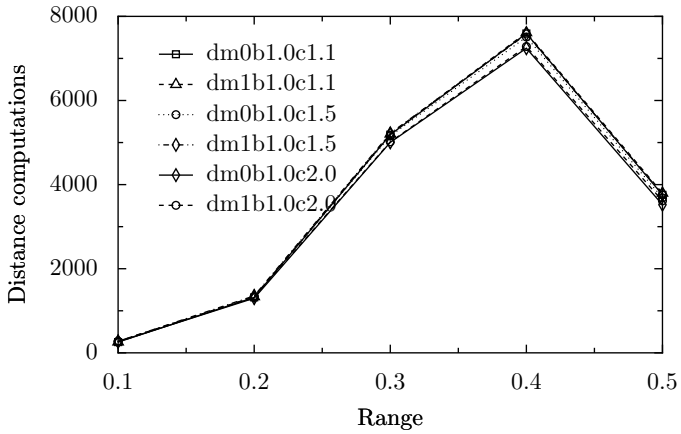
5.23.1: Distance computations



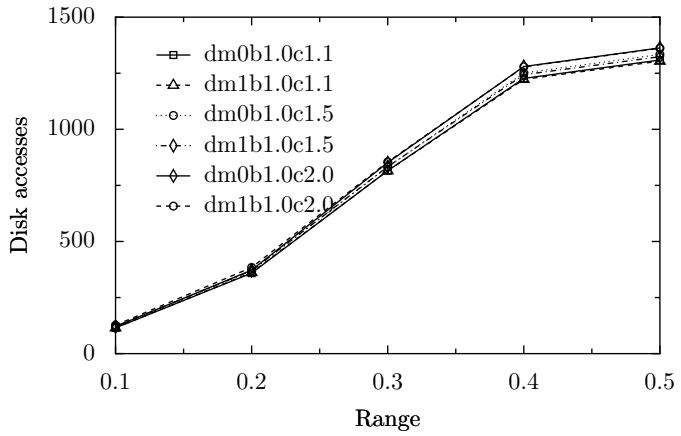
5.23.2: Disk I/O

Figure 5.23: Range search in GBMVP-trees for different values of b, with $c = 1.1$. The data is clustered and in 10 dimensions.

like with the GBVP-tree, the GBMVP-tree was tested with several different values for b and c to see if they could change the number of rebuilds, and how this again can change the search costs. Table 5.14 shows the results of building, and it is clear that the build cost rises heavily when the number of rebuilds grows. Because of $k = 25$, the tree grew so slow that only total rebuilds happened. Some height improvement is seen in the trees using balancing, and the results of the range search in Figure 5.23 and Figure 5.24 shows that there is indeed some for some of the trees, but is only only visible on high ranges. As expected, the KNN-search in Table 5.15 indicates that the delete method using marking does best when the number of rebuilds are high, while the method using rebuilding gets an advantage when b is higher, which makes the tree make less total rebuilds.



5.24.1: Distance computations



5.24.2: Disk I/O

Figure 5.24: Range search in GBMVP-trees for different values of c, with $b = 1.0$. The data is clustered and in 10 dimensions.

To test the importance of the delete algorithm, two trees using the two different delete algorithms where tested against each other. As described in Section 3.3, the first one of these two methods uses rebalancing when one of the pivot objects (vantage points) are deleted, while the second one only marks the the node as deleted. Included in the test are also two trees using the two deletion algorithms, but these trees use no GB balancing at all. In other words, the first of the two trees using no GBT balancing only receives its balancing when internal nodes are deleted, while the second method has no balancing at all. To get the tree high enough to see any real results, a value for k as low as 2 was used. The result of the range search on clustered data in 10 dimensions can be seen in Figure 5.25, while the building costs and heights can be seen in Table 5.16 and the results of the KNN-search in

DM	b	c	r0.1	r0.2	r0.3	r0.4	r0.5
Rebuild	0.5	1.1	267.2	1320.5	5055.1	7335.8	3590.0
Mark	0.5	1.1	244.6	1281.0	5014.1	7268.4	3482.3
Rebuild	1.0	1.1	255.5	1318.9	5186.8	7592.9	3756.7
Mark	1.0	1.1	259.3	1335.9	5203.8	7613.2	3798.4
Rebuild	1.5	1.1	248.8	1307.9	5213.7	7648.3	3825.6
Mark	1.5	1.1	262.0	1358.4	5254.8	7709.9	3921.9
Rebuild	1.0	1.5	270.4	1332.8	5156.5	7509.3	3683.0
Mark	1.0	1.5	272.7	1355.1	5219.7	7577.2	3772.5
Rebuild	1.0	2.0	265.7	1299.9	5004.5	7242.3	3521.9
Mark	1.0	2.0	276.218	1323.8	5015.9	7283.5	3613.1

Table 5.12: Range search showing average distance computation values in GBMVP-trees with different values for b and c. The ranges are from 0.1-0.5 (marked r0.1-r0.5). Done on clustered data in 10 dimensions.

DM	b	c	r0.1	r0.2	r0.3	r0.4	r0.5
Rebuild	0.5	1.1	901	5276	10068	10340	9134
Mark	1.0	1.1	810	5186	10126	10190	8974
Rebuild	1.0	1.1	998	5434	10465	10386	9322
Mark	1.0	1.1	865	5389	10542	10416	9135
Rebuild	1.5	1.1	868	5446	10451	10465	9236
Mark	1.5	1.1	977	5637	10535	10547	9345
Rebuild	1.0	1.5	909	5296	10563	10266	9152
Mark	1.0	1.5	954	5415	10531	10264	9270
Rebuild	1.0	2.0	953	5220	10027	9987	8899
Mark	1.0	2.0	804	5220	10449	9944	8988

Table 5.13: Range search showing the worst values in GBVP-trees with different values for b and c. The ranges are from 0.1-0.5 (marked r0.1-r0.5). Done on clustered data in 10 dimensions.

Table 5.17.

As can be seen, there is a very slight improvement for both the number of distance computations (Figure 5.25.1) and disk I/O (Figure 5.25.2) for the trees using the GBT rebuilding, but the increase is not exactly dramatic. Expectedly, there is some rise in the cost of building the trees as well, although, just like the distance computations, it is not dramatic either. Surprisingly, the average height of the method using no rebalancing was actually lower than the one using it on delete. The same was also true when running a test with $k = 4$ (Figure A.5, Table A.1 and Table A.2). However, it does not apply for the trees using the GBT rebalancing, where the average

DM	b	c	Dist comp	Tot reb	Part reb	Height
0	0.5	1.1	989684	6.4	0.0	12.0
1	0.5	1.1	989824	6.3	0.0	12.0
0	1.0	1.1	559976	1.0	0.0	14.0
1	1.0	1.1	558308	1.0	0.0	14.0
0	1.5	1.1	494408	0.0	0.0	14.0
1	1.5	1.1	494854	0.0	0.0	14.0
0	1.0	1.5	672622	2.7	0.0	13.0
1	1.0	1.5	640208	2.6	0.0	13.8
0	1.0	2.0	951624	5.3	0.0	12.0
1	1.0	2.0	951624	5.3	0.0	12.0

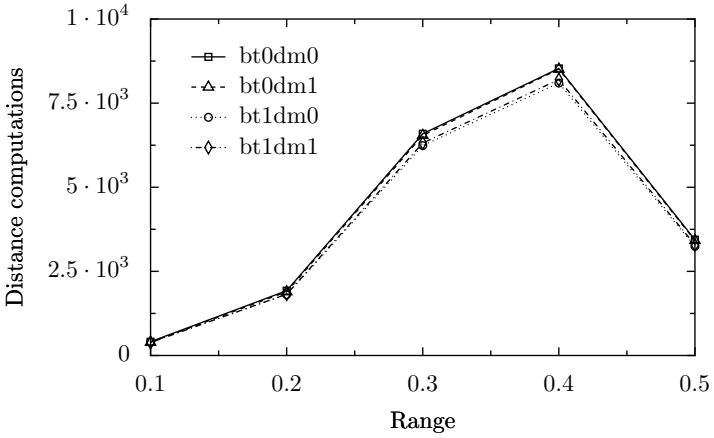
Table 5.14: Building cost of GBMVP-trees showing how different values for b and$c$ can influence the build cost. Done on clustered data in 10 dimensions.

DM	b	c	Dist comp	Disk I/O
Rebuild	0.5	1.1	2236.944	509.323
Mark	0.5	1.1	2228.294	500.980
Rebuild	1.0	1.1	2233.220	480.472
Mark	1.0	1.1	2262.824	483.025
Rebuild	1.5	1.1	2291.732	455.578
Mark	1.5	1.1	2189.255	452.942
Rebuild	1.0	1.5	2165.155	481.652
Mark	1.0	1.5	2312.922	496.121
Rebuild	1.0	2.0	2191.281	500.384
Mark	1.0	2.0	2202.600	509.915

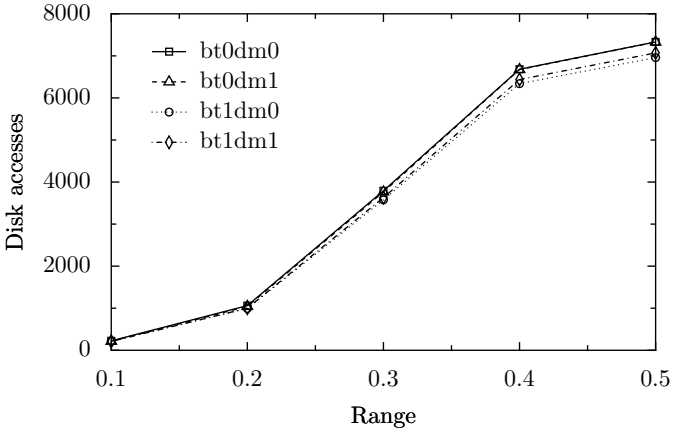
Table 5.15: KNN-search in GBMVP-trees showing how different values for b and$c$ can influence the build cost. Done on clustered data in 10 dimensions.

BT rebuild	Delete method	Dist comp	Tot reb	Part reb	Height
No	Rebuild	570846	0.0	0.0	22.0
No	Mark	570930	0.0	0.0	21.2
Yes	Rebuild	648976	0.0	15.6	18.4
Yes	Mark	631952	0.0	10.2	19.2

Table 5.16: Building cost of GBMVP-trees showing how the General Balanced Trees (BT) and delete methods can change the build cost and height of the trees.



5.25.1: Distance computations



5.25.2: Disk I/O

Figure 5.25: Range search in GBMVP-trees on clustered data using GBT rebuilding and no rebuilding (bt) and delete methods (dm, where 0 is with rebuilding when deleting pivot objects and 1 is with marking) and $k = 2$.

BT rebuild	Delete method	Dist comp	Disk I/O
No	Rebuild	3369.632	1849.009
No	Mark	3291.13	1800.374
Yes	Rebuild	3149.643	1717.498
Yes	Mark	3111.619	1702.302

Table 5.17: KNN search costs for GBMVP-trees showing how the General Balanced Trees (BT) and delete methods can change the search costs.

height is slightly a little bit lower for the ones using rebalancing on delete.

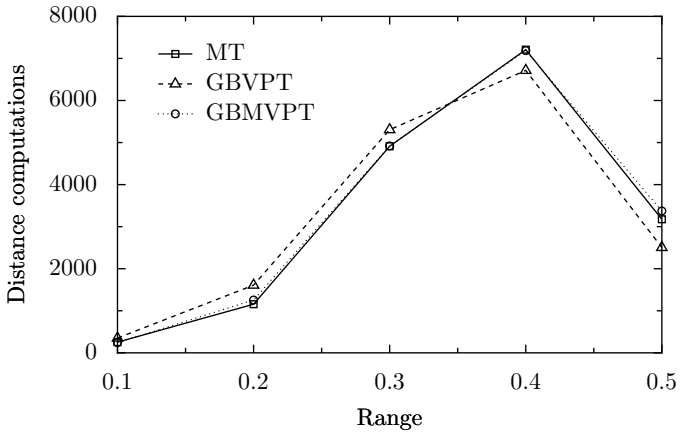
5.7 Comparisons

Some tests where done where the three different dynamic structures where set up against each other. All the three dynamic tree structures, the M-tree, the GBVP-tree and the GBMVP-tree, were tested against each other using the same datasets, insertions deletions and query objects. They where both tested on tests using the L_2 metric and the edit distance function. To make each tree perform as well as it could, the parameters that were found to be most impressive in the previous sections were used for each tree. However, it is also important that the GBT rebuilding is actually used by any of the trees in the test.

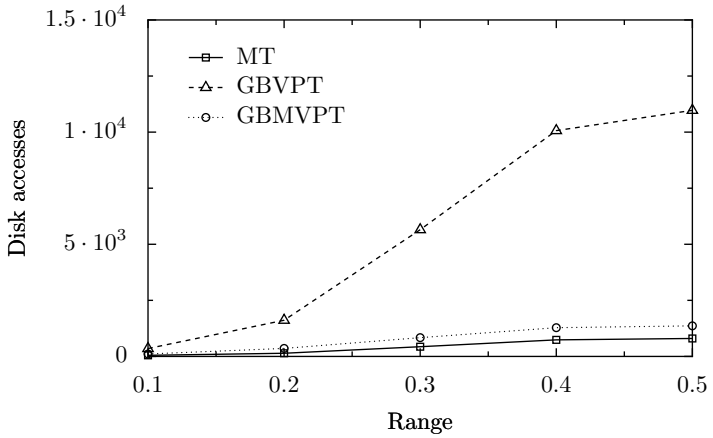
The GBVP-tree uses a random selection of vantage points to make sure that the build costs stay within proportions. As a higher percentage did not really give any advantage anyways, this should not have too much to say on search performance. The GBMVP-tree was set to only $k = 25$. While a higher value for k may have given better search performance, the value was chosen for several reasons. Firstly, it is the value where most information was known about how to set the b and c values. Secondly, it is because $k = 25$ performed very well in the tests in Section 5.6 without having the builds costs go through the roof.

Just like the GBMVP-tree, the M-tree uses a fan-out of 25, but here on every level, not just the leaves. Considering the tests in Section 5.4, a the results seemed to converge at around 25, so it should probably be a good value while still have good run-time performance. It also uses the mMRAD algorithm for promotion and generalized hyperplane for partitioning.

The results of the range search using the L_2 metric is shown in Figure 5.26. As can be seen, the GBMVP-tree and the M-tree performs more or less equal numbers of distance computations (Figure 5.26.1), while the M-tree uses about half as much disk I/O. This is probably because the M-tree uses a much larger fan-out on all levels, therefore reading many more objects on each node access, which also results in a much lower tree (Table 5.18). According to Table 5.19, the KNN-search shows about the same, with the M-tree having a slight lead in distance computations and under half as many disk accesses, and the worst case (Table 5.20 follows the same trend). The GBVP-tree performs just a little bit worse when looking at distance computations, but, as expected, the disk I/O is much higher due to the fact that only one object is read on each disk access and the fact that the tree is much taller. However, when looking at the building costs (Table 5.18), the M-tree is by far the most expensive, with the GBMVP-tree using over 10 times as few distance computations. That the difference was this big was quite surprising.



5.26.1: Distance computations



5.26.2: Disk I/O

Figure 5.26: Range search using the L_2 metric distance function on different dynamic tree structures.

To see how much the delete algorithm for the M-tree influenced this, a test using only inserts where done. The results of the building are seen in Table 5.21, and while the M-tree does perform better compared to the other structures, it still uses far more distance computations. Note that the parameters for the GBVP-tree and GBMVP-tree where exactly not the same as in Table 5.19, so a direct comparison can not be made. The rest of the results from that test can be found in Appendix A.3.

Sadly, the edit distance search did not work very well, mostly due to the fact that the using single words as keys turned out to not be discriminating enough. As can be seen in Figure 5.27, when the search range increases, more or less every object is checked and included in the search. The KNN-search

Tree	Distance computations	Height
M-tree	10193250	4.0
GBVPT	1432850	17.8
GBMVPT	986632	12.0

Table 5.18: Build costs and height for different dynamic trees using L_2 metric distance on clustered data in 10 dimensions.

Tree	Distance computations	Disk I/O
M-tree	2073.072	215.049
GBVPT	2962.537	2962.537
GBMVPT	2189.212	494.463

Table 5.19: KNN search for different dynamic trees using the L_2 metric distance on clustered data in 10 dimensions.

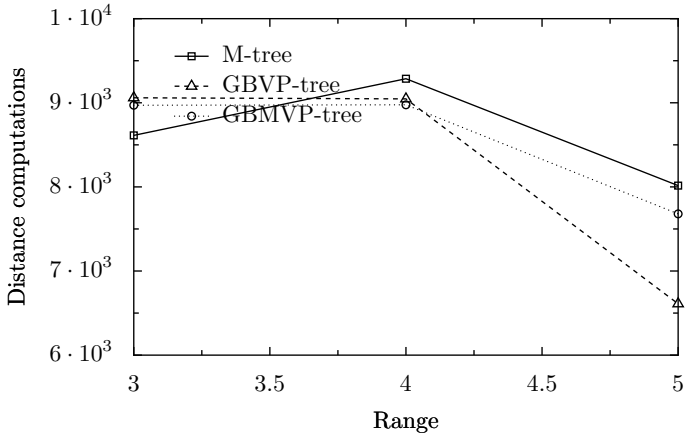
(Table 5.22) also confirms this. There is still some interesting results though, as the GBMVP-tree beat the M-tree in the number of distance computations when the range went up, with the GBVP-tree not far behind, and it also did best in the KNN-search. This could however simply be from the fact that the M-tree has more nodes than the other trees, which combined with the fact that more or less every node in the tree is visited as no pruning is done, means that the M-tree will do worse. Even so, the M-tree did best when concerning disk I/O, but that could again be explained by the low height of the tree. But, as can be seen in Table 5.22, the build costs for the M-tree were once again extremely high compared to the other two trees.

Tree	r0.1	r0.2	r0.3	r0.4	r0.5
M-tree	1157	6632	10269	10066	9500
GBVPT	1348	7425	9770	9786	8285
GBMVPT	1173	6732	9867	9802	9174

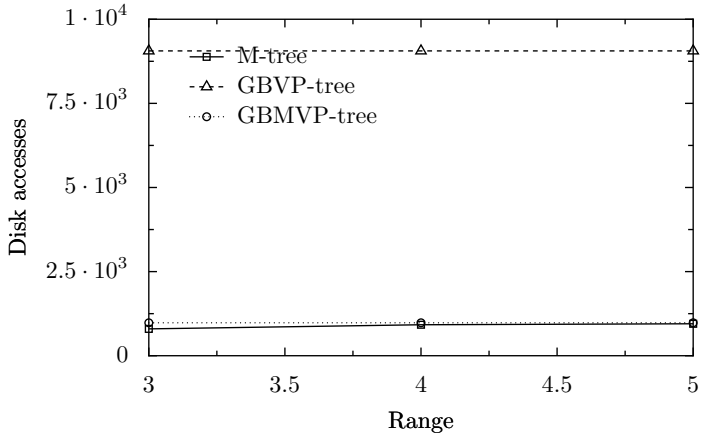
Table 5.20: Worst results for range search in different dynamic trees using the L_2 metric distance on clustered data in 10 dimensions.

Tree	Distance computations	Height
M-tree	2596576	4.0
GBVPT	639116	22.0
GBMVPT	568370	14.8

Table 5.21: Build costs and height for different dynamic trees using L_2 metric distance on clustered data in 10 dimensions.



5.27.1: Distance computations



5.27.2: Disk I/O

Figure 5.27: Range search using the edit distance function on different dynamic tree structures.

Tree	Distance computations	Height
M-tree	11161838	4.0
GBVPT	1500720	20.0
GBMVPT	610270	16.4

Table 5.22: Build costs and height for different dynamic trees using the edit distance.

Tree	Distance computations	Disk I/O
M-tree	9301.2	879.2
GBVPT	9059.4	9059.4
GBMVPT	8973.8	978.2

Table 5.23: KNN search costs for GBMVP-trees showing how the General Balanced Trees (BT) and delete methods can change the search costs.

Chapter 6

Discussion and Conclusion

6.1 Discussion

As can be seen from the results in Chapter 5, the results by using the GB paradigm with standard metric trees had both promising and less than promising results. While it is clear that the GB-paradigm do indeed reduce the hight of the trees when used correctly, it is also clear that it is harder to adapt the trees successfully to the new environment than initially thought, and the actual improvements in search time were sadly minimal.

There are several reasons for why this does not work as good as expected, and several runs down to the problems with similarity search. For a binary search tree, the worst case search time for an exact search will always be the maximum height of the tree. Therefore, it is very important for the tree to stay balanced to reduce the worst case search. However, while determining what the average search time is for a tree using both insertions and deletions is far from trivial, [THCS01] gives a proof that the average search time of a binary tree using only insertions will be $log_2(n)$, so the average search time in a binary search tree will not be a good estimate of the effect of balancing a binary search tree.

Similarity search in metric trees is a bit different. Because many of them, including both the VP-tree and the MVP-tree includes a lot of randomization during the building, and because of the branching during search due to the nature of similarity search, determining what exactly is the worst case search in a tree for a given range is a bit more difficult. While it is clear that balancing is good, exactly how much impact does it have on both the average search and the worst case performance?

The randomization has also got another disadvantage when it gets to rebuilding, as there is no real way of knowing whether or not a rebuilt tree will actually be better or not than the tree that was chosen for rebuild. With a binary search tree, it can easily be made sure that, given the existing objects, a perfect new tree can be made with little extra cost. This is not the

case with many metric tree structures, and especially trees like the VP-tree which simply selects its vantage points on random. The tree could actually go from having good vantage points to having really bad ones that do not partition the data in any good way. But of course, as this is random, given enough samples this should even out.

It is clear from the results in Chapter 5 that the GBT criteria do indeed help to reduce the height of the trees, but no definite improvements were found in either the average or the worst case performance. In this respect the results where a failure. However, in theory, the worst case results were indeed reduced, which is a success for the GBT paradigm.

The setting of the b and c variables turned out to be far from trivial as well, and it looks like this must be adjusted for each tree structure. Also, there is a question about how much a delete should count when the leaves can have more than two data objects stored in them. Tests with the GBMVP-tree (Section 5.6) showed that the number of total rebuildings got higher along with k. This was because the tree grew very slow, and therefore the criteria for total rebuilding is reached faster. However, this does not mean that the tree needs the rebuilding more than a tree using a lower value for k. In fact, it is the complete opposite, as a tree using the a lower value for k will become unbalanced faster because it grows faster. For the same reason, the number of partial rebuilds is reduced to nothing when k grows, which is again due to the fact that the tree grows so slow compared to the number of deletes. This could maybe be solved by giving the deletions a weight dependent on the value of k, so a delete in tree where k is high should be given a lesser weight than one where k is small. The same could also be done for insertions, only in the opposite way.

One very interesting observation was how good the GBVP-tree and GBMVP-tree performed compared to the M-tree. While the M-tree has seen several improved versions over the years, it is still impressive for these two experimental versions to perform almost just as well as this well-proven tree. And, as the versions using no rebuilding did not perform much worse, this may be an indication that even very simple dynamic versions of static structures can perform almost as well as the more complicated originally dynamic ones.

But finally, there is one very interesting question that is worth mentioning, and that is if these kinds of tree structures really are the best way to structure a metric access method. First of all, non-tree based methods like the hash based methods, including the SH [CGZ01], D-index [VDZ03] and eD-index [DGZ03] have shown very promising results, and it is a bit concerning when a simple structure like LC [CN00] can beat even the most advanced of trees. Additionally, [CNBYM01] discusses how compact tree structures probably is a better choice than pivot based tree structures when the number of dimensions grows, which mean that the trees based on Generalized Hyperplane Partition will work better than the Ball Partitioning

methods like the VP and MVP-tree.

6.2 Conclusion

The General Balanced Trees paradigm [And99] was in this thesis tested out on two different tree structures, the Vantage Point Tree and the Multiple Vantage Point Tree, that are originally static metric tree structures. To make this possible, the trees where extended to be dynamic trees with both insert and delete methods. Instead of using complex criteria and methods for how to balance these trees in a dynamic environment, the trees instead used the simple global criteria from [And99], leading to total and partial rebuilding of the trees when needed.

Creating dynamic trees from static structures proved to be non-trivial, as several problems had to be overcome. The worst problem was how to handle deletion, because internal nodes could not be removed or easily replaced in neither the VP or the MVP-tree. Two different methods for deletion was proposed and in this thesis, with one being more dependent on the rebuilding by the GBT criteria than the other one. The result was that one one that was heavily dependent performed when the parameters where set so that a lot of rebuilding was done, while the other performed better when rebuilding was less frequent.

Setting the b and c parameters also turned out to be a problem, as the different trees had different characteristics, and the way the trees grow have a large impact on how the parameters must be set. A property that especially turned out to be a problem was the fan-out, because in trees growing slow in height due to a high fan-out, either on every level or just in the leaves, a delete will have a much greater impact on the global criteria than in one growing fast. The result was that, even with c close to the minimum value 1, some trees would more or less never do partial rebuilding. However, results indicated that they did total rebuilds more often than needed.

Testing the trees showed that the trees did indeed become more balanced when the GBT rebuilding was done. However, the tests also showed that this did not improve search results by much. But a slight improvement was found in several of the results, and this should improve even more if used for large data sets than it was possible to use for this thesis.

After testing the GBVP-tree and GBMVP-tree to see how they performed with different settings, the two trees where tested against the M-tree to see how they were performed. Because the M-tree lacked a delete algorithm, one based on the delete algorithm in R-trees was proposed and implemented. The trees where tested on both vectors of real numbers using the L_2 metric distance function and English words using the edit distance function under different conditions, testing both the abilities of performing insertions and deletions. While the the GBVP-tree and the GBMVP-tree

did not perform any better than the M-tree, they did perform comparably well, and the idea of how simple, global criteria can be used for rebalancing metric trees instead of complex, local ones could be worth further investigation.

6.3 Future work

While the results from this thesis showed some promise for the GBT paradigm, several problems needs to be solved for the paradigm to be used efficiently. This including the problem with how to handle deletions, as the two ways proposed in this thesis are probably not the optimal way to do it. Section 6.1 also implies that a Generalized Hyperplane based structure might be a better idea than using ball partitioning, and it could be interesting to see how one would perform.

There are several possibilities with the GBT paradigms that could be explored in order to create more efficient structures. First of all, the paradigms in [And99] should be expanded into handle trees with a larger fan-out than just binary trees. This should be able to improve the method by quite a bit, because it has been shown that several metric tree structures can improve when a larger fan-out is used. While the MVP-tree does quite a bit better with a high value for k, [BO97] shows that a fan-out on internal nodes improves the searching as well. The same is true for many other tree structures as well. Also, this could solve the problem encountered with the hybrid MVP-tree used in this thesis.

Second, more state of the art structures should be joined with the GBT paradigms to see if better dynamic tree structures can be created. While the VP-tree and MVP-tree are structures that are frequently used as comparisons for other structures, and therefore gives a good indication of the possibilities of a new structure, many other structures have later shown much better results compared to them, with some of them mentioned in Section 2.6.

Finally, the values used for b and c, the two variables that affect when a total and partial rebuild will happen, should be explored more. While some experimentation has been done with different values in this thesis, it was not very successful, and this is a matter which requires more investigation.

Bibliography

[And99] Andersson. General balanced trees. *ALGORITHMS: Journal of Algorithms*, 30, 1999.

[AVL62] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. 1962.

[Bay71] Rudolf Bayer. Binary b-trees for virtual memory. In E. F. Codd and A. L. Dean, editors, *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, November 11-12, 1971*, pages 219–235. ACM, 1971.

[Bay72] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.

[Bel61] Richard Bellman. *Adaptive Control Processes : a Guided Tour*. Princeton University Press, 1961.

[BGRS99] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? *Lecture Notes in Computer Science*, 1540:217–235, 1999.

[BK73] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, April 1973.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An index structure for high-dimensional data. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A., 1996. Morgan Kaufmann Publishers.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[BO97] Tolga Bozkaya and Meral Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. pages 357–368, 1997.

[BO99] Tolga Bozkaya and Meral Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems*, 24(3):361–404, 1999.

[Bri95] Sergey Brin. Near neighbor search in large metric spaces. In *The VLDB Journal*, pages 574–584, 1995.

[BY97] Ricardo Baeza-Yates. Searching: An algorithmic tour. In Allen Kent and James G. William, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker, Inc, 1997.

[BYCMW94] Ricardo A. Baeza-Yates, Walter Cunto, Udi Manber, and Sun Wu. Proximity matching using fixed-queries trees. In *CPM '94: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 198–212, London, UK, 1994. Springer-Verlag.

[CGZ01] Pasquale Savino Claudio Gennaro and Pavel Zezula. Similarity search in metric databases through hashing. In *Proceedings of the 2001 ACM workshops on Multimedia: multimedia information retrieval*, pages 1–5. ACM Press, 2001.

[Chi94] Tzi-cker Chiueh. Content-Based Image Indexing. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 582–593, Santiago, Chile, 1994.

[CMN99] Edgar Chávez, José L. Marroquín, and Gonzalo Navarro. Overcoming the curse of dimensionality, 1999.

[CN00] E. Chávez and G. Navarro. An effective clustering algorithm to index high dimensional metric spaces. In *Proceedings of the 6th International Symposium on String Processing and Information Retrieval (SPIRE'2000)*, pages 75–86. IEEE CS Press, 2000.

[CNBYM01] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Proximity searching in metric spaces. *ACM Computing Surveys*, 2001. To appear.

[CP98] P. Ciaccia and M. Patella. Bulk loading the m-tree, 1998.

[CP00a] P. Ciaccia and M. Patella. Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *Proceedings of the 16th International Conference on Data Engineering (ICDE 2000)*, March 2000.

[CP00b] Paolo Ciaccia and Marco Patella. The m^2-tree: Processing complex multi-feature queries with just one index. In *DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries*, 2000.

[CPRZ97] Paolo Ciaccia, Marco Patella, Fausto Rabitti, and Pavel Zezula. Indexing metric spaces with m-tree. In *Sistemi Evolui per Basi di Dati*, pages 67–86, 1997.

[CPZ97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *The VLDB Journal*, pages 426–435, 1997.

[DGZ03] Vlastislav Dohnal, Claudio Gennaro, and Pavel Zezula. Similarity join in metric spaces using ed-index. In *DEXA*, pages 484–493, 2003.

[Doh04] Vlastislav Dohnal. *Indexing Structures for Searching in Metric Spaces*. PhD thesis, Masaryk University, February 2004.

[FsCCM00] Ada Wai-Chee Fu, Polly Mei shuen Chan, Yin-Ling Cheung, and Yiu Sang Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *VLDB Journal*, 9(2):154–173, 2000.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47 – 57. ACM Press, 1984.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proc. 21st Int. Conf. Very Large Data Bases, VLDB*, pages 562–573. Morgan Kaufmann, 11–15 1995.

[HSE+95] James Hafner, Harpreet S. Sawhney, Will Equitz, Myron Flickner, and Wayne Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 17(7):729–736, 1995.

[KM83] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, 9:631–634, 1983.

[KS00] Norio Katayama and Shin'ichi Satoh. Similarity image retrieval with significance-sensitive nearest-neighbor search, '2000.

[Kue93] Geoff Kuenning. ispell-3.3.02. http://www.lasr.cs.ucla.edu/geoff/ispell.html (Accessed 23.05.06), 1993.

[LCGMW02] C. Li, E. Chang, H. Garcia-Molina, and G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces, 2002.

[LJF94] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal: Very Large Data Bases*, 3(4):517–542, 1994.

[MOV94] Maria Luisa Micó, José Oncina, and Enrique Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recogn. Lett.*, 15(1):9–17, 1994.

[MS94] B. Moret and H. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree, 1994.

[NBS90] Ralf Schneider Norbert Beckmann, Hans-Peter Kriegel and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322 – 331. ACM Press, 1990.

[NR72] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 137–142. ACM Press, 1972.

[NS] Beomseok Nam and Alan Sussman. A comparative study of spatial indexing techniques for multidimensional scientific datasets.

[PZB06] Vlastislav Dohnal Pavel Zezula, Giuseppe Amato and Michal Batko. *Similarity Search - The Metric Space Approach.* Springer, 2006.

[PZR96] Paolo Ciaccia Pavel Zezula and Fausto Rabitti. M-tree: A dynamic index for similarity queries in multimedia databases. Technical report, Technical Report 7, HERMES ESPRIT LTR Projects, 1996.

[Rig04] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial*

evaluation and semantics-based program manipulation, pages 15–26, New York, NY, USA, 2004. ACM Press.

[Rot91] Gunter Rote. Computing the minimum hausdorff distance between two point sets on a line under translation. *Information Processing Letters*, 38(3):123–127, 1991.

[Sko04] Tomas Skopal. Pivoting m-tree: A metric access method for efficient similarity search, 2004.

[SW90] Dennis Shasha and Tsong-Li Wang. *New techniques for best-match retrieval*, volume 8. ACM Press, April 1990.

[THCS01] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, September 2001.

[TTSF00] Caetano Traina Jr., Agma Traina, Bernhard Seeger, and Christos Faloutsos. Slim-Trees: High performance metric trees minimizing overlap between nodes. *Lecture Notes in Computer Science*, 1777:51–65, 2000.

[Uhl91] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[USB98] J. Goldstein U. Shaft and K. Beyer. Nearest neighbors query performance for unstable distributions. Technical report, October 1998.

[VDZ03] Pasquale Savino Vlastislav Dohnal, Claudio Gennaro and Pavel Zezula. D-index: Distance searching index for metric data sets. In *Multimedia Tools and Applications*, volume 21, pages 9–33. Kluwer Academic Publishers, September 2003.

[XZY03] Jeffrey Xu Yu Xiangmin Zhou, Guoren Wang and Ge Yu. M+-tree: a new dynamical multidimensional index for metric spaces. In *Proceedings of the Fourteenth Australasian database conference on Database technologies 2003*, volume 17, pages 161 – 168. Australian Computer Society, Inc., 2003.

[Yia93] Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1993.

[YMT96] Apostolos N. Papadopoulos Yannis Manolopoulos, Alexander Nanopoulos and Yannis Theodoridis. *R-Trees: Theory and Application*. Springer, 1996.

[ZWZY05] Xiangmin Zhou, Guoren Wang, Xiaofang Zhou, and Ge Yu. Bm+-tree: A hyperplane-based index method for high-dimensional metric spaces. In Lizhu Zhou, Beng C. Ooi, and Xiaofeng Meng, editors, *DASFAA*, volume 3453, pages 398–409. Springer, 2005.

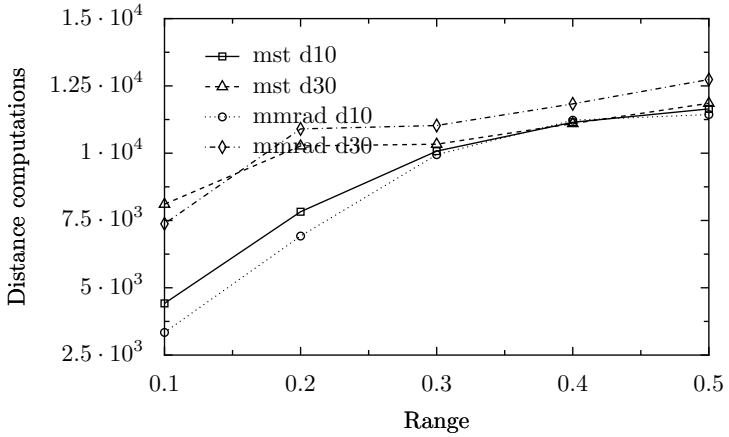
Appendix A

Additional Results

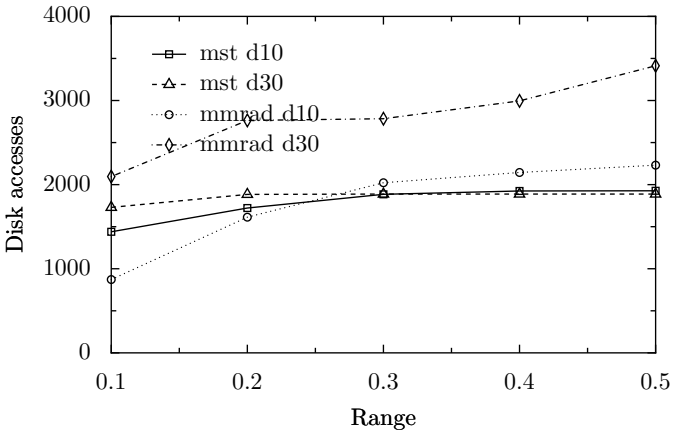
This section contains results not included in Chapter 5 because they did not add anything new to the table. Some are referred to for additional information in Chapter 5, but many are here just for the interested reader to be able to check more sides of the results of tests on the trees.

A.1 M-tree

A.1.1 MST vs. mMRAD



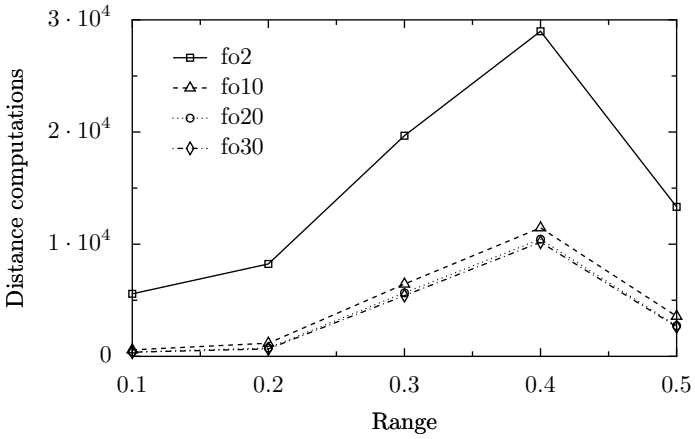
A.1.1: Distance computations



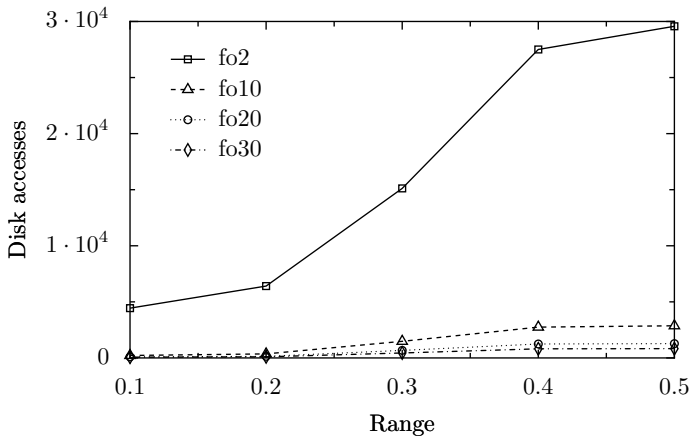
A.1.2: Disk I/O

Figure A.1: Range search in a M-tree using the MST and mMRAD promotion algorithms in 10 and 30 dimensions on uniformly distributed data.

A.1.2 Different Fan-Outs

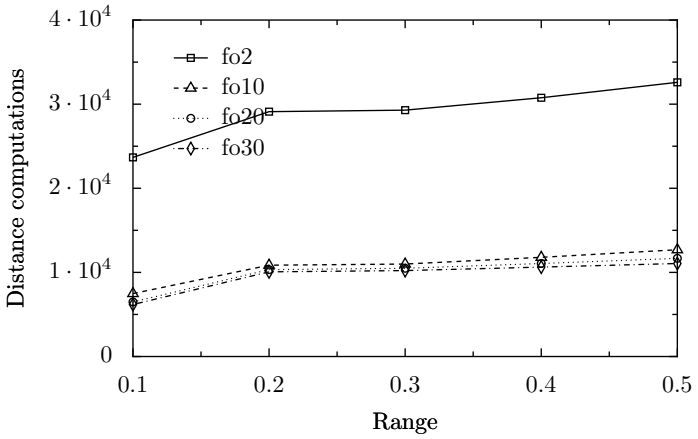


A.2.1: Distance computations

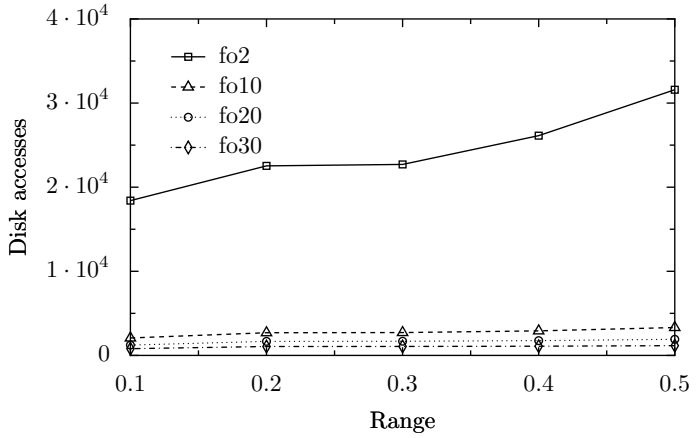


A.2.2: Disk I/O

Figure A.2: Range search in a M-tree for different fan-outs in 10 dimensions on clustered data.

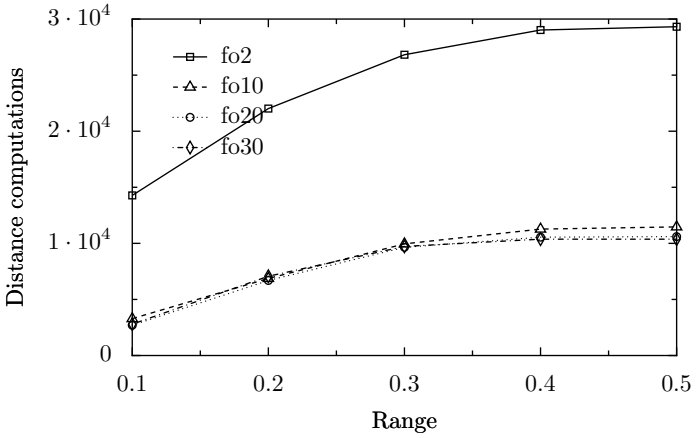


A.3.1: Distance computations

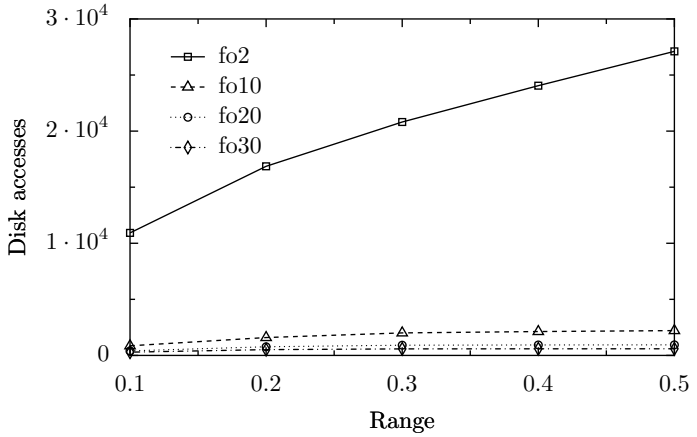


A.3.2: Disk I/O

Figure A.3: Range search in a M-tree for different fan-outs in 10 dimensions on uniformly distributed data.



A.4.1: Distance computations

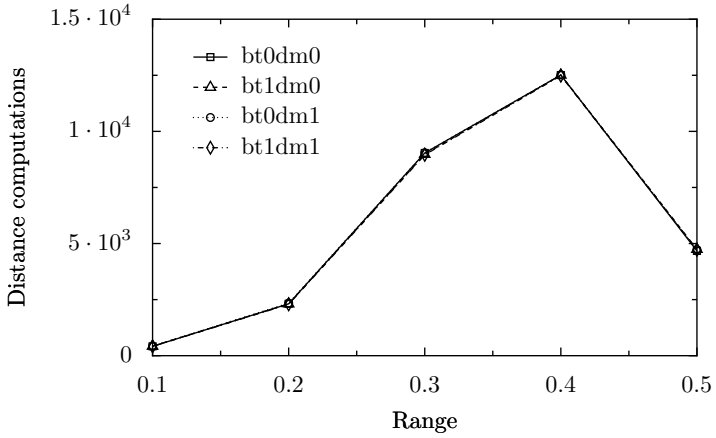


A.4.2: Disk I/O

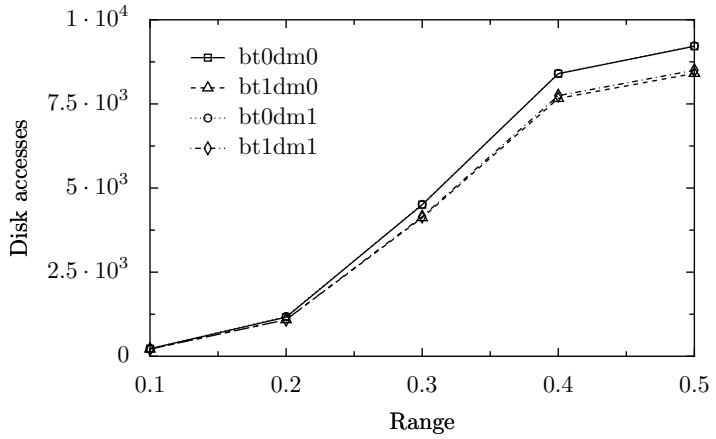
Figure A.4: Range search in a M-tree for different fan-outs in 10 dimensions on uniformly distributed data.

A.2 GBMVP-tree

A.2.1 Rebuilding



A.5.1: Distance computations



A.5.2: Disk I/O

Figure A.5: Range search in GBMVP-trees on clustered data using GBT rebuilding and no rebuilding (bt) and delete methods (dm, where 0 is with rebuilding when deleting pivot objects and 1 is with marking) and $k = 4$.

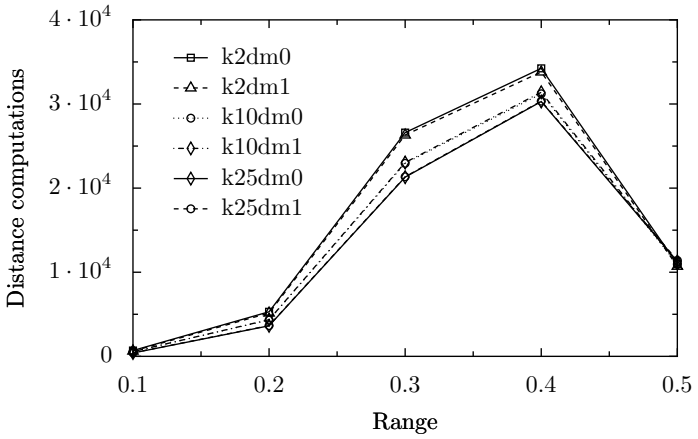
BT rebuild	Delete method	Dist comp	Tot reb	Part reb	Height
No	Rebuild	616024	0.0	0.0	21.0
No	Mark	616718	0.0	0.0	20.2
Yes	Rebuild	901990	0.0	2.2	17.4
Yes	Mark	892500	0.0	2.1	17.6

Table A.1: Building cost of GBMVP-trees showing how the General Balanced Trees (BT) and delete methods can change the build cost and height of the trees.

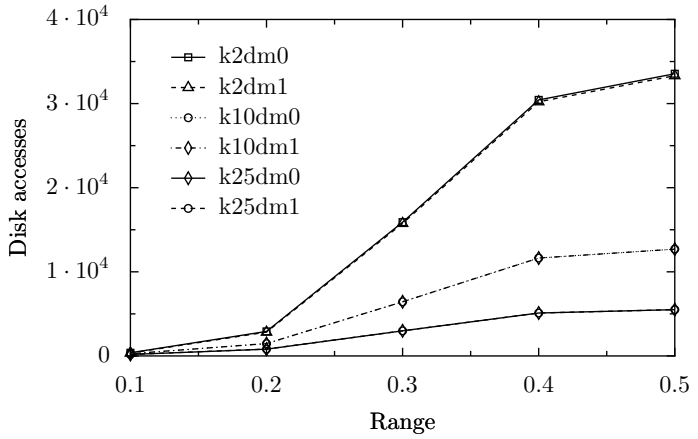
BT rebuild	Delete method	Dist comp	Disk I/O
No	Rebuild	4046.34	1985.864
No	Mark	3963.033	1948.41
Yes	Rebuild	3827.003	1749.381
Yes	Mark	3809.02	1752.832

Table A.2: KNN search costs for GBMVP-trees showing how the General Balanced Trees (BT) and delete methods can change the search costs with $b = 0.5$ and $c = 1.2$.

A.2.2 Different values for k



A.6.1: Distance computations



A.6.2: Disk I/O

Figure A.6: Range search in GBMVP-trees on clustered data in 10 dimensions showing the difference between $k = 2$, $k = 10$ and $k = 25$ with the two different delete methods.

k	Delete method	Dist comp	Tot reb	Part reb	Height
2	Rebuild	3399264	0.0	50.1	23.4
2	Mark	3497070	0.0	37.6	22.4
10	Rebuild	3141390	0.1	1.0	19.6
10	Mark	3050950	0.0	0.0	20.4
25	Rebuild	3148720	2.0	0.0	16.0
25	Mark	3144748	2.0	0.0	16.0

Table A.3: Building cost of GBMVP-trees showing how the different values for k and delete methods can change the build cost and height of the trees with $b = 1.0$ and $c = 1.2$. 1000 initial objects and 100000 objects inserted/deleted and the number of dimensions is 10.

k	Delete method	Dist comp	Disk I/O
2	Rebuild	8852.645	4874.354
2	Mark	8831.279	4857.166
10	Rebuild	6990.469	2248.051
10	Mark	7025.858	2257.199
25	Rebuild	5901.253	1164.662
25	Mark	5921.760	1169.121

Table A.4: KNN search costs for GBMVP-trees showing how the the different values for k and delete methods can change the search costs with $b = 1.0$ and $c = 1.2$. 1000 initial objects and 100000 objects inserted/deleted and the number of dimensions is 10.

k	Delete method	Dist comp	Tot reb	Part reb	Height
2	Rebuild	3382508	0.0	105.5	24.4
2	Mark	3618810	0.0	70.0	22.4
10	Rebuild	3311586	0.1	0.8	20.0
10	Mark	3048420	0.0	0.1	20.2
25	Rebuild	3133380	2.0	0.0	16.0
25	Mark	3135446	2.0	0.0	16.0

Table A.5: Building cost of GBMVP-trees showing how the different values for k and delete methods can change the build cost and height of the trees with $b = 1.0$ and $c = 1.2$. 1000 initial objects and 100000 objects inserted/deleted and the number of dimensions is 30.

k	Delete method	Dist comp	Disk I/O
2	Rebuild	24374.471	13100.876
2	Mark	23225.800	12102.937
10	Rebuild	20267.769	5102.676
10	Mark	20502.727	5224.53
25	Rebuild	18426.064	2427.647
25	Mark	18652.145	2470.522

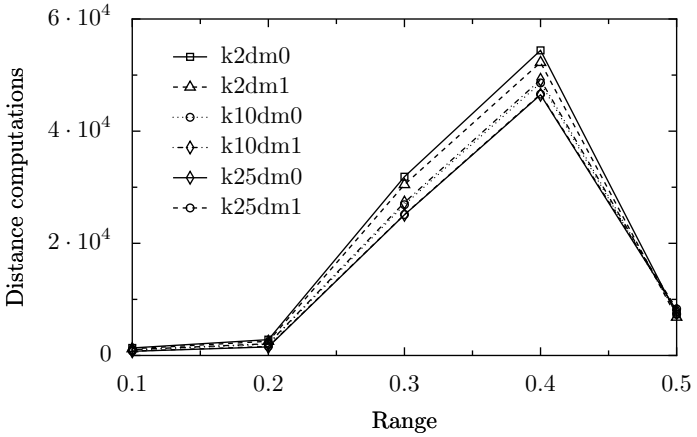
Table A.6: KNN search costs for GBMVP-trees showing how the the different values for k and delete methods can change the search costs with $b = 1.0$ and $c = 1.2$. 1000 initial objects and 100000 objects inserted/deleted and the number of dimensions is 30.

k	Delete method	Dist comp	Tot reb	Part reb	Height
2	Rebuild	3559784	0.0	94.8	23.0
2	Mark	3360336	0.0	81.8	23.6
10	Rebuild	3338250	1.8	1.3	19.4
10	Mark	3201248	1.6	1.4	19.6
25	Rebuild	3153868	10.1	0.0	16.0
25	Mark	3138070	9.8	0.0	16.0

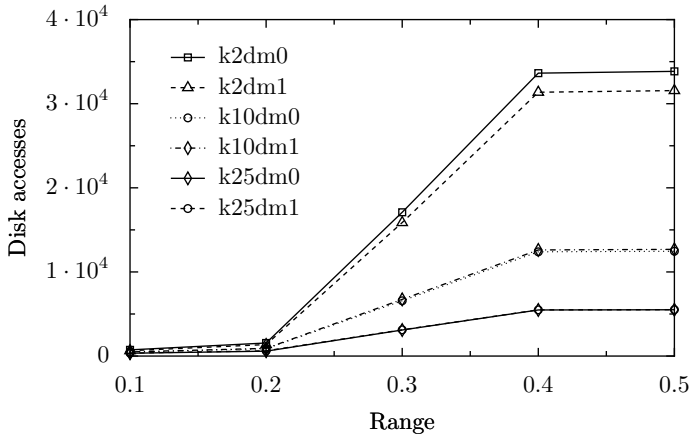
Table A.7: Building cost of GBMVP-trees showing how the different values for k and delete methods can change the build cost and height of the trees with $b = 1.0$ and $c = 1.2$. 1000 initial objects and 100000 objects inserted/deleted and the number of dimensions is 30 and the data is uniformly distributed.

k	Delete method	Dist comp	Disk I/O
2	Rebuild	24312.430	13225.097
2	Mark	25097.032	13966.604
10	Rebuild	18603.071	6010.079
10	Mark	18382.985	5998.635
25	Rebuild	16420.876	3093.605
25	Mark	16387.421	3087.385

Table A.8: KNN search costs for GBMVP-trees showing how the the different values for k and delete methods can change the search costs with $b = 1.0$ and $c = 1.2$. 1000 initial objects and 100000 objects inserted/deleted and the number of dimensions is 10 and the data is uniformly distributed.

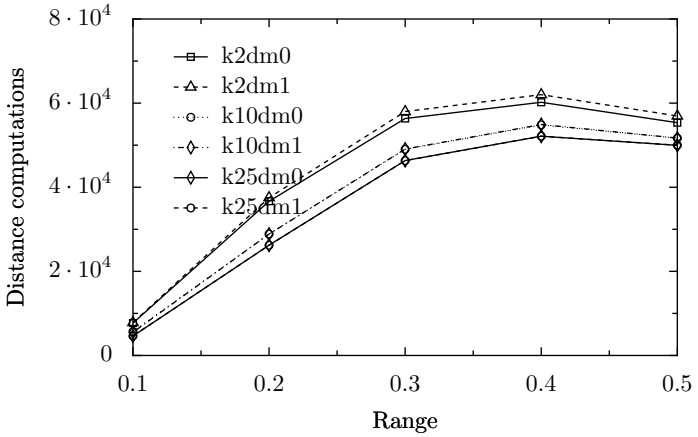


A.7.1: Distance computations

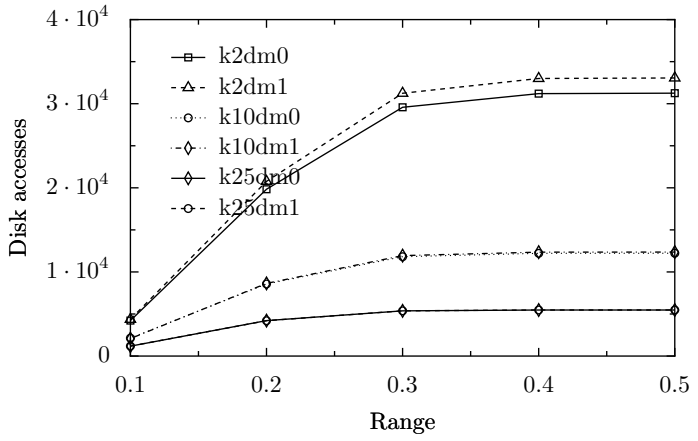


A.7.2: Disk I/O

Figure A.7: Range search in GBMVP-trees on clustered data in 30 dimensions showing the difference between $k = 2$, $k = 10$ and $k = 25$ with the two different delete methods.

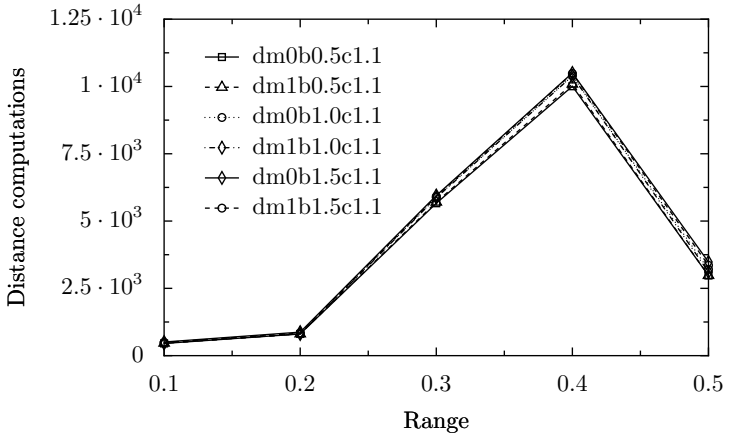


A.8.1: Distance computations

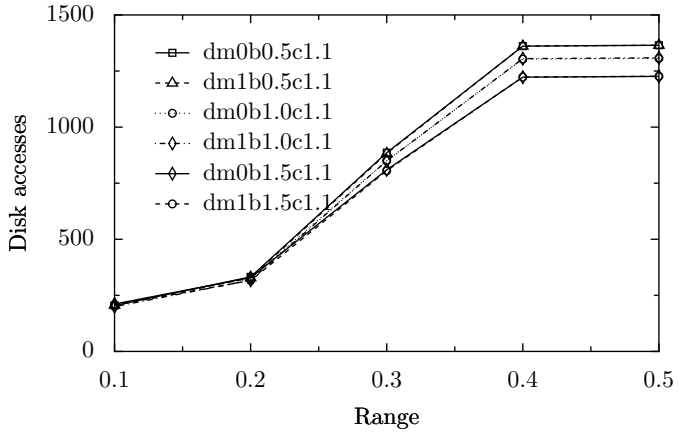


A.8.2: Disk I/O

Figure A.8: Range search in GBMVP-trees on uniformly distributed data in 10 dimensions showing the difference between $k = 2$, $k = 10$ and $k = 25$ with the two different delete methods.



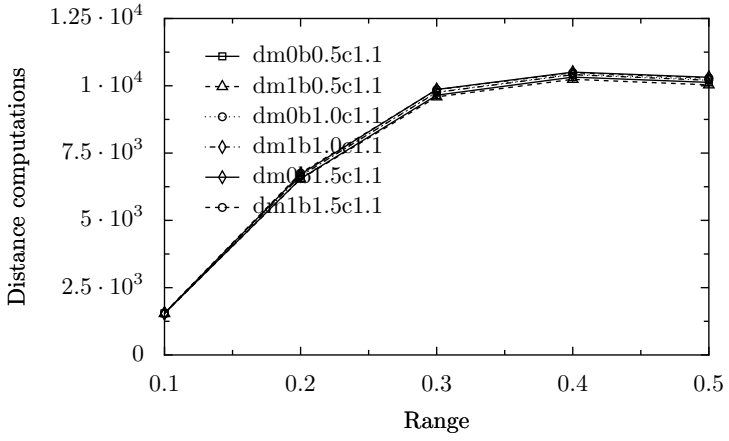
A.9.1: Distance computations



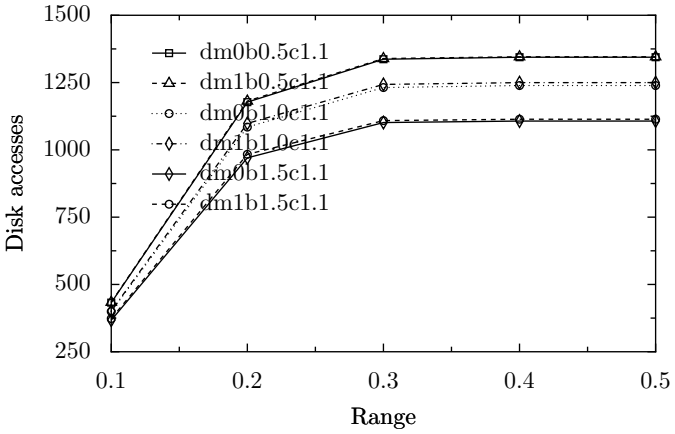
A.9.2: Disk I/O

Figure A.9: Range search in GBMVP-trees for different values of b, with $c = 1.1$. The data is clustered and in 10 dimensions.

A.2.3 Different values for b and c

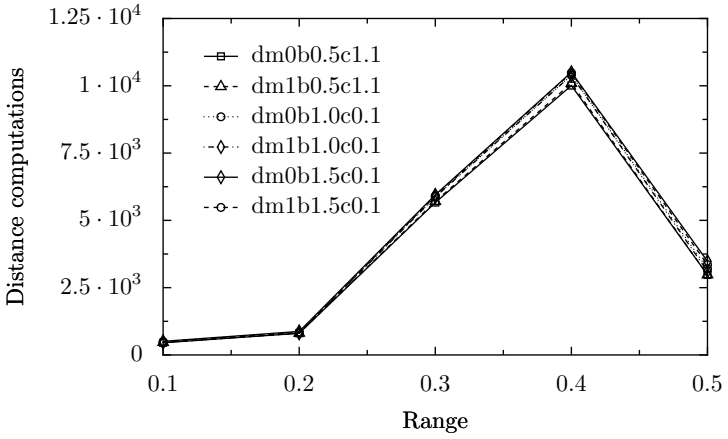


A.10.1: Distance computations

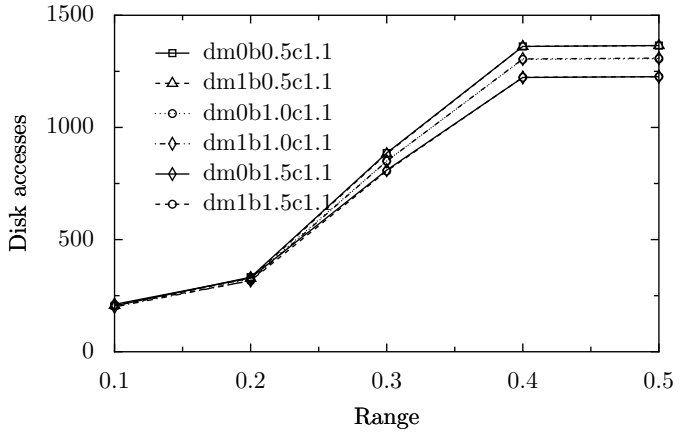


A.10.2: Disk I/O

Figure A.10: Range search in GBMVP-trees for different values of b, with $c = 1.1$. The data is clustered and in 10 dimensions.

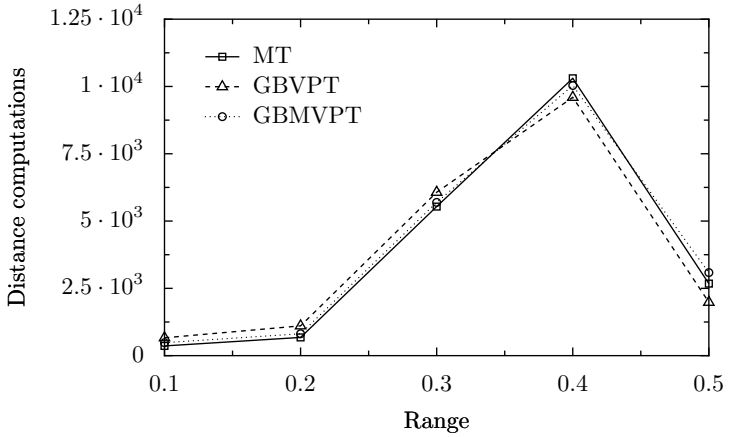


A.11.1: Distance computations

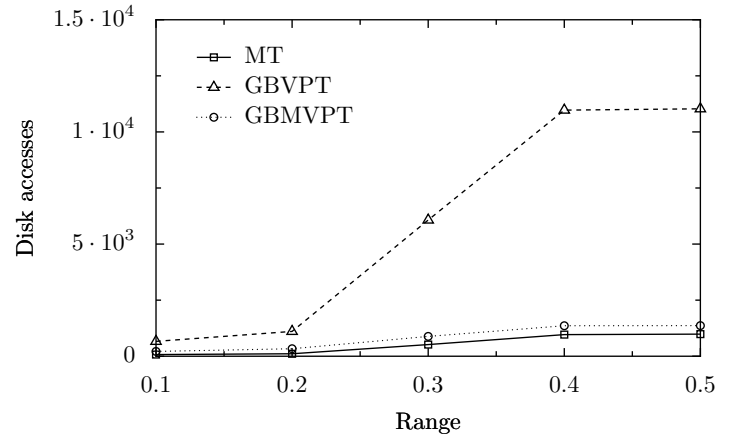


A.11.2: Disk I/O

Figure A.11: Range search in GBMVP-trees for different values of b, with $c = 1.1$. The data is clustered and in 10 dimensions.



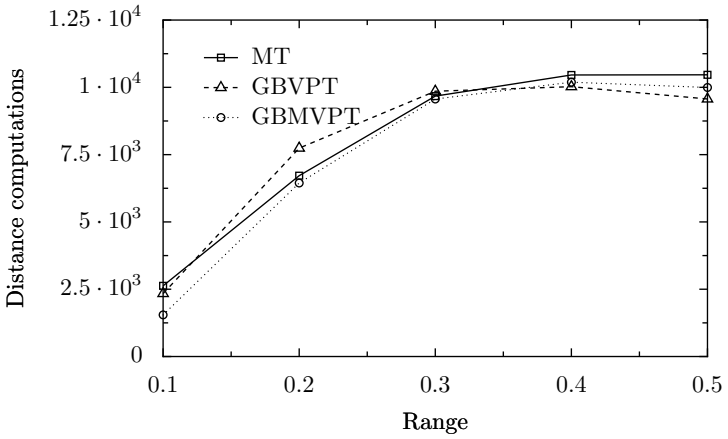
A.12.1: Distance computations



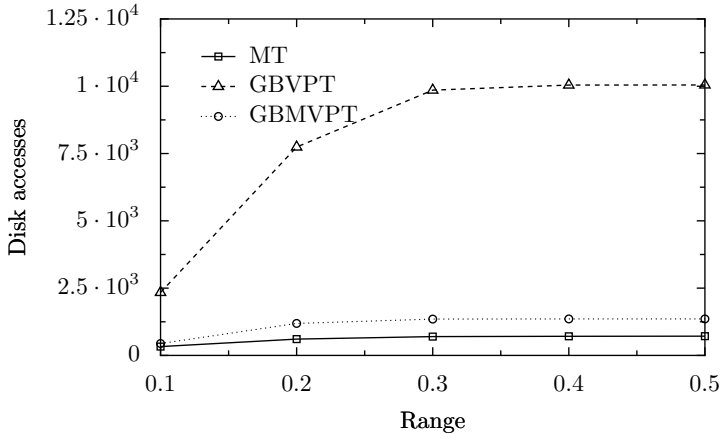
A.12.2: Disk I/O

Figure A.12: Range search using the L_2 metric distance function on different dynamic tree structures on clustered data with 30 dimensions.

A.3 Comparisons

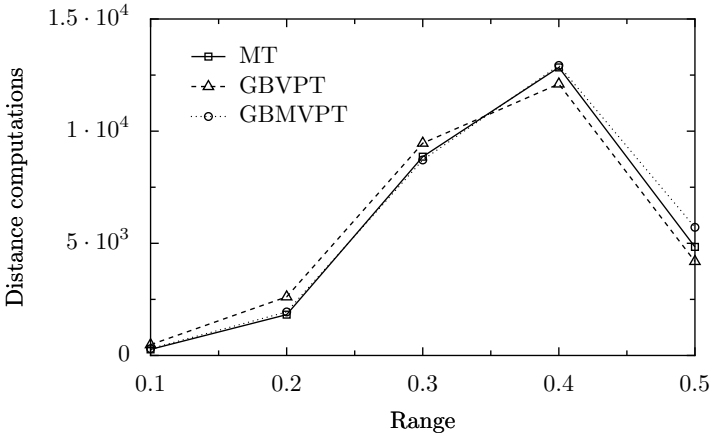


A.13.1: Distance computations

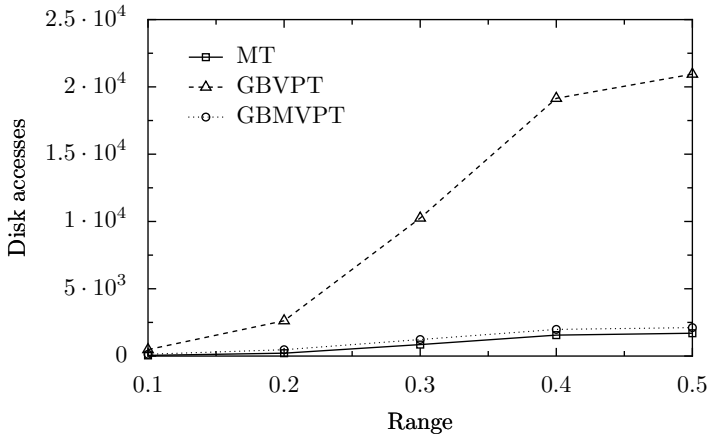


A.13.2: Disk I/O

Figure A.13: Range search using the L_2 metric distance function on different dynamic tree structures on uniformly distributed data with 10 dimensions.

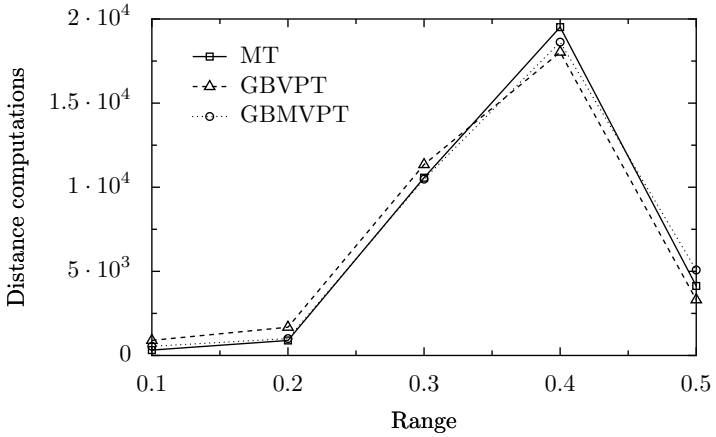


A.14.1: Distance computations

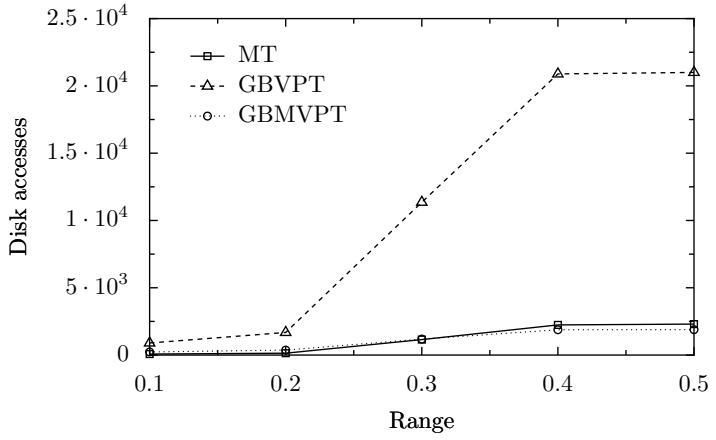


A.14.2: Disk I/O

Figure A.14: Range search using the L_2 metric distance function on different dynamic tree structures on clustered data with 10 dimensions. Only insertions were done.

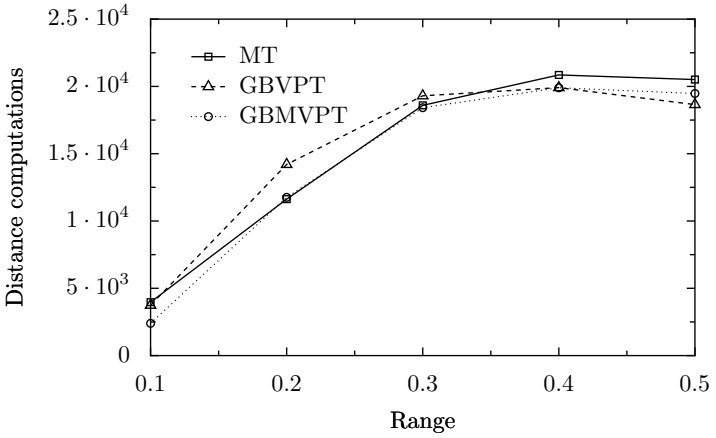


A.15.1: Distance computations

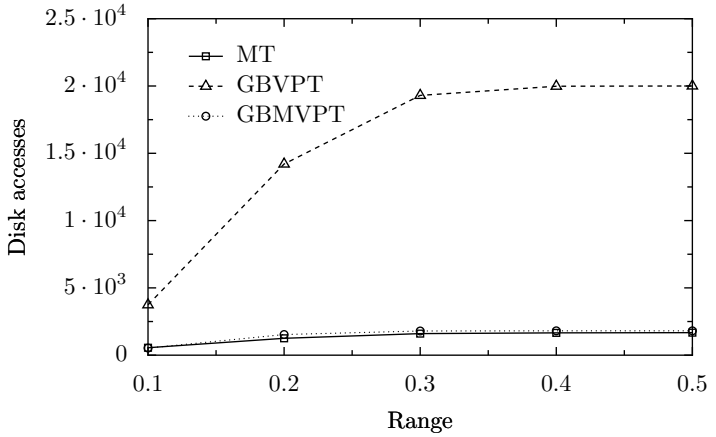


A.15.2: Disk I/O

Figure A.15: Range search using the L_2 metric distance function on different dynamic tree structures on clustered data with 30 dimensions. Only insertions were done.



A.16.1: Distance computations



A.16.2: Disk I/O

Figure A.16: Range search using the L_2 metric distance function on different dynamic tree structures on uniformly distributed data with 10 dimensions. Only insertions were done.