Efficient data structures for range searching on a grid

Mark H. Overmars

RUU-CS-87-2
January 1987

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

Efficient data structures for range searching on a grid

Mark H. Overmars

Technical Report RUU-CS-87-2
January 1987

Department of Computer Science
University of Utrecht
3508 TA Utrecht
the Netherlands

Efficient data structures for range searching on a grid

Mark H. Overmars

January 1987

Abstract

We consider the 2-dimensional range searching problem in the case where all point lie on an integer grid. A new data structure is presented that solves range queries on a $U * U$ grid in $O(k + \log\log U)$ time using $O(n \log n)$ storage, where n is the number of points and k the number of reported answers. Although the query time is very good the preprocessing time of this method is very high. A second data structure is presented that can be built in time $O(n \log n)$ at the cost of an increase in query time to $O(k + \sqrt{\log U})$. Similar techniques are used for solving the line segment intersection searching problem in a set of axis-parallel line segments on a grid. The methods presented also yield efficient stuctures for dominance searching and searching with half-infinite ranges that use only $O(n)$ storage. A generalization to multi-dimensional space, using a normalization approach, yields a static solution to the general range searching problem that is better than any known solution when the dimension is at least 3.

1 Introduction.

One of the problems in computational geometry that has received a large amount of attention is the *range searching* problem. Given a set of n points in a d-dimensional space, the range searching problem asks to store these points such that for a given range $([A_1..B_1]\ldots[A_d..B_d])$ one can efficiently determine which points lie in the range, i.e., report the points $P = (P_1, \ldots, P_d)$ such that $A_1 \leq P_1 \leq B_1, A_2 \leq P_2 \leq B_2, \ldots$ and $A_d \leq P_d \leq B_d$. The range searching problem has many applications in such areas as database design and computer graphics. The problem has been studied in great detail. See for example [1,2,4,6,24,25]. In the case the structure is static, i.e., no points have to be inserted or deleted the best known result is by Chazelle[4] who constructs a structure that uses space $O(n\frac{\log^{d-1} n}{\log\log n})$ and has a query time of $O(k + \log^{d-1} n)$. Many trade-offs can be obtained between the query time and the amount of space required (see e.g. Scholten and Overmars[20]).

In this paper we study the range searching problem on a 2-dimensional grid, i.e., points have integer coordinates between 0 and $U-1$ where U is called the universe size. On a grid many problems can be solved more efficiently by using table lookup and perfect hashing techniques. Note that lowerbounds, like e.g. the ones for range searching in Chazelle[5], do not hold on a grid. Hence, solutions that are optimal ion arbitrary space might be improvable on a grid. Studying problems on a grid yields potential practical solutions because in many applications points indeed have integer coordinates (e.g. in computer graphics). Only recently a number of papers have discussed problems from computational geometry on a grid (see e.g. [10,11,12,13,14,15,17,22,23]). Problems considered so far are the nearest neighbor searching problem ([11,12]), the point location problem ([17]) and a number of off-line problems like e.g. computing the convex hull of a set of points or the intersections in a set of line segments ([10,13,14,15]).

In this paper we will treat the range searching problem on a grid, i.e., we are given a set of points in U^2 and we want to store them such that range queries can be performed efficiently. We will present two different solutions. Our first solution has a query time of $O(k + \log\log U)$, where k is the number of reported answers. Unfortunately the preprocessing time of this method is very high. A second solution uses only moderate preprocessing at the cost of an increase in query time to $O(k + \sqrt{\log U})$. Both structure use $O(n \log n)$ storage.

Both solutions start by looking at the dominance searching problem that asks for locating all points that dominate a given query point. Next this solution is transformed into a solution to the half-infinite range searching problem in which ranges have the form $([A_1..B_1], [A_2..\infty])$. Both problems can be solved in the time bounds stated above using only $O(n)$ storage. The solutions are based on a combination of the priority search tree of McCreight[16] and the y-fast and q-fast trie of Willard[22,23].

The paper is organised as follows. In Section 2 we recall some known results and structures that will be used throughout this paper. Readers who are familiar with structures like x-fast tries[22,23], priority search trees[16] and segment trees[3] can skip this section.

In Section 3 we treat the dominance searching problem and show how it can be solved efficiently on a grid. In Section 4 we generalize the structure for dominance searching to range queries with half-infinite ranges. In Section 5 we use this structure to solve the general range searching problem on a 2-dimensional grid. This solution has a high preprocessing time because it is based on perfect hashing. In Section 6 we demonstrate a second technique that solves the dominance searching problem, the half-infinite range searching problem and the general range searching problem on a grid using table lookup rather than perfect hashing.

These 2-dimensional structures can be generalized to work in d-dimensional space as well. In Section 7 it is shown how this can be done. Moreover, a normalization technique will be presented that turns range searching problems in arbitrary space into grid problems. In this way a new data structure for range

searching in arbitrary d-dimensional space ($d \geq 3$) is obtained yielding a query time of $O(k + \log^{d-2} n \log\log n)$ using $O(n \log^{d-1} n)$ storage.

The techniques presented for solving range searching on a grid can be used for solving other problems on a grid as well. In Section 8 we show that a similar method can be used to solve the *line segment intersection searching* problem: Given a set of horizontal line segments, store them such that for any vertical line segment we can efficiently determine which line segments it intersects. Using the segment tree as a basic structure we will show that this problem can also be solved in a query time of $O(k + \log\log U)$ on a grid.

Finally, in Section 9 we give some conclusions and a number of open problems.

Throughout this paper we assume that no two points have a same value for some coordinate. On a grid this is not very realistic. Fortunately, all techniques can easily be generalised to the case where more points have the same value in some coordinate. At relevant places it is briefly indicated how methods can be adapted.

An extended abstract of this paper appeared in [18].

2 Preliminaries.

In this section we will describe some known results and data structures that will be used throughout this paper. Those familiar with such structures as y-fast tries by Willard[22,23], priority search trees of McCreight[16] and segments trees as described in e.g. Bentley and Wood[3], can skip this section.

We first recall a result by Fredman e.a.[8] concerning perfect hashing on a grid.

Theorem 2.1 *([8]) Given a set of n keys in a bounded universe of known size U, they can be stored using $O(n)$ storage such that for a given key K in this universe one can determine in $O(1)$ time whether K is in the set of keys.*

Unfortunately this structure has a very high preprocessing time of $O(n^3 \log U)$. The method really is a hashing scheme in the sense that it can only answer the question whether a given key is present or not. When a key is not present it is unable to find its nearest neighbor, like e.g. search trees can.

Willard[22] defined a so-called *x-fast trie* that is able to perform such nearest neighbor queries on a 1-dimensional grid efficiently. We will briefly describe this structure. Let V be a set of n keys in a universe of size U. For convenience, we assume the universe consists of the integers $0..U-1$. On the universe we build a trie, i.e., a balanced binary tree in which each internal node corresponds to some part of the universe. The root corresponds to the whole universe and the two sons of an internal node δ correspond to equal halves of the part corresponding to δ. Clearly, the depth of such a trie is bounded by $O(\log U)$ and the trie uses $O(U)$ storage. To reduce the amount of storage required we continue the splitting of the universe only until the part of a node δ contains 0 or 1 key. If the part contains

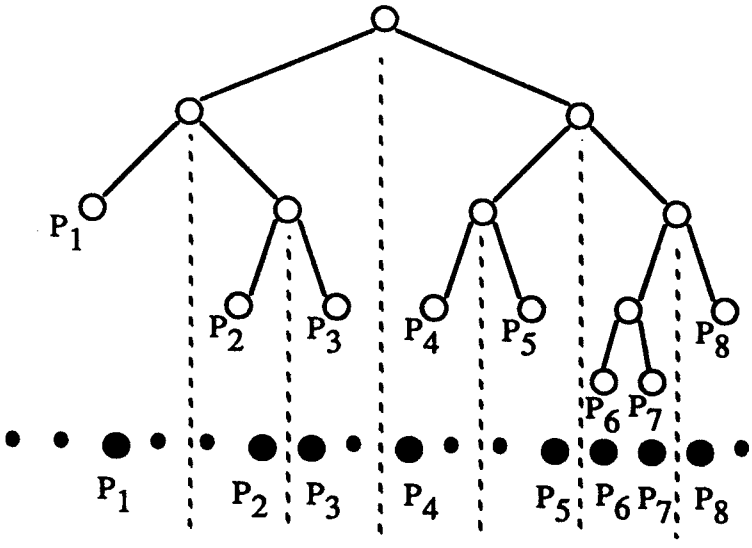

Figure 1: x-fast trie.

1 key, we make δ into a leaf and store the key there. The leaves we link in a list such that, once we find a leaf we can easily find its two neighbors. See figure 1 for an example.

To enable fast searching we augment the structure in the following way. For each level i of the trie we build a structure H_i for perfect hashing (see Theorem 2.1.) of the nodes present at this level. Clearly, on level i there are at most 2^i nodes, hence, the universe size for H_i is bounded by 2^i. In this way we can determine in O(1) time whether a given node δ on some level i is present or not, using H_i. It can easily be seen that the structure uses $O(n \log U)$ storage.

To perform a query on the x-fast trie with a key K we will look for the leaf K comes in when searching in the trie, but, rather than walking from the root down the tree till we reach a leaf, we perform binary search on the depth of the tree. To this end we compute the node on the middle level through which the search path would go. Because the structure is a trie this can be done in O(1) time. Next we use the H structure at this level to determine whether this node exists. If it does exist, we know that the leaf we are looking for lies on a lower level of the trie and we continue our search there. If the node is not present, the leaf lies in the top half of the tree and we continue there. Hence, in O(1) time we halved the depth of the tree that remains to be searched. In the remaining subtree we recursively do the same, until the node we check is a leaf. For details see [22]. This leads to a query time of $O(\log\log U)$.

To reduce the amount of storage required we use a technique called *pruning* (see van Emde Boas[21]). Rather than storing one key at each leaf we store $\log U$ keys at each leaf of the trie. These $\log U$ keys we store in a normal balanced search tree (e.g. an AVL-tree). This clearly reduces the storage to $O(n)$. To search with

a key K we first search in the trie. In $O(\log\log U)$ time we find a leaf of this trie. Next we search the AVL-tree stored at this leaf in the normal way. As this AVL-tree contains $\log U$ keys this takes another $O(\log\log U)$ time. Willard[22] calls this pruned trie a *y-fast trie*. We conclude:

Theorem 2.2 *([22]) Let V be a set of n keys in a universe of size U. We can store V using $O(n)$ storage such that, given a key K we can determine in $O(\log\log U)$ time the largest key in $V \leq K$ (or the smallest key $\geq K$).*

Because this structure uses perfect hashing the preprocessing is again very high (bounded only by $O(U^3 \log U)$). Another structure, also due to Willard[23] (independently devised by Karlsson[11]), exists, called a *p-fast trie,* that can be build more efficiently at the cost of a small increase in query time. We will again briefly describe this structure.

Let $f(U) = 2^{\sqrt{\log U}}$. Rather than building a binary trie we construct a trie in which each node has $f(U)$ sons, each son corresponding to an equal part of the universe. Again we only continue splitting until the part corresponding to a node contains 0 or 1 key. Clearly, this trie has depth $O(\sqrt{\log U})$. To search with a key K we start at the root and search down the trie. Because the structure is a trie, for each internal node on the search path of K we can determine in $O(1)$ time in which son the search has to continue. Hence, we find the required answer in time $O(\sqrt{\log U})$. Unfortunately the structure uses $O(n\sqrt{\log U} f(U))$ storage.

To reduce the amount of storage required, Willard[23] introduced the so-called *q-fast trie*. This is a pruned variant of the p-fast trie. We only split until a part contains at most $\sqrt{\log U} f(U)$ keys. These keys we store in an AVL-tree. Searching this AVL-tree costs time $O(\sqrt{\log U})$. For details see [23].

Theorem 2.3 *([23]) Let V be a set of n keys in a universe of size U. We can store V using $O(n)$ storage such that, given a key K we can determine in $O(\sqrt{\log U})$ time the largest key in $V \leq K$ (or the smallest key $\geq K$). The structure can be built in time $O(n\sqrt{\log U})$.*

(When $\sqrt{\log U} > \log n$, the structure can be built in time $O(n \log n)$. Of course, in this case we better use a simple balanced search tree instead.)

In fact, this structure can be made dynamic but, as we are only dealing with static solutions in this paper, this is not important for our purposes.

Our method for solving the range searching problem on a grid will be based on a technique by Edelsbrunner[6] that is based on the so-called *priority search tree* by McCreight[16]. We will briefly describe their solution here.

A priority search tree is a structure for solving half-infinite range queries in arbitrary 2-dimensional space (i.e., not neccessary on a grid). Let V be a set of n points in the plane. For convenience, we assume that no two points have an equal value for some coordinate. We store the points, ordered with respect to their first coordinate in the leaves of a balanced binary search tree T. With each

internal node we associate a point of the set in the following way: With the root we associate the point in T with largest y-coordinate. With each internal node δ we store the point in T_δ (the subtree rooted at δ) with largest y-coordinate that has not been stored higher up in the tree. It is clear that this structure uses $O(n)$ storage.

To perform a query with range $([A_1..B_1],[A_2..\infty])$ we search with both A_1 and B_1 in the tree. For each node on the search path we check whether it lies in the range and, if so, report it. All other answers must lie in between the two search paths. Let δ be a node that lies in between the two search paths, whose father lies on one of the search paths. We will show how to report all answers in T_δ. All points in T_δ have first coordinate between A_1 and B_1. Hence, we only have to test whether the second coordinate is larger than A_2. δ contains the point in T_δ with largest y-coordinate. Hence, if this point does not lie in the range, no point in T_δ does and we are done. If the point does lie in the range we report it and continue looking at the two sons of δ. It is easily shown that the amount of work for finding all the answers in T_δ is bounded by $O(1)$ plus the number of answers found. Hence, the total query time is $O(\log n)$ for searching with A_1 and B_1, another $O(\log n)$ because we have to spend $O(1)$ time in $O(\log n)$ subtrees plus the total number of answers. For details see [16].

Theorem 2.4 *([16]) Given a set of n points in the plane, they can be stored, using $O(n)$ storage, such that half-infinite range queries can be answered in time $O(k+\log n)$ where k is the number of reported answers.*

Edelsbrunner[6] uses this structure to solve general 2-dimensional range queries in the following way. We construct a balanced binary search tree T in which we store all points, sorted by y-coordinate, in the leaves. With an internal node δ that is left son of its father we store a priority search tree P_δ of all points in T_δ. If δ is right son of its father we store an inverse priority search tree, i.e., a priority search tree for ranges that extend downwards to infinity. The structure takes $O(n\log n)$ storage.

Now assume we want to perform a query with range $([A_1..B_1],[A_2..B_2])$. We search with both A_2 and B_2 in T. For some time both search paths will be the same but at some internal node δ A_2 will lie in the left subtree and B_2 will lie in the right subtree. Let δ_l be the left son of δ and δ_r the right son. Clearly all answers lie in T_{δ_l} or T_{δ_r}. All points in T_{δ_l} have second coordinate $< B_2$. Hence, we can as well perform a query on them with range $([A_1..B_1],[A_2..\infty])$. For this we can use the structure P_{δ_l}. Similar, for the answers in T_{δ_r} we can as well perform a half-infinite range query with $([A_1..B_1],[-\infty..B_2])$ on P_{δ_r}. Hence, the total query time is bounded by $O(k+\log n)$.

Theorem 2.5 *([6]) Given a set of n points in the plane, they can be stored, using $O(n\log n)$ storage, such that range queries can be answered in time $O(k+\log n)$ where k is the number of reported answers.*

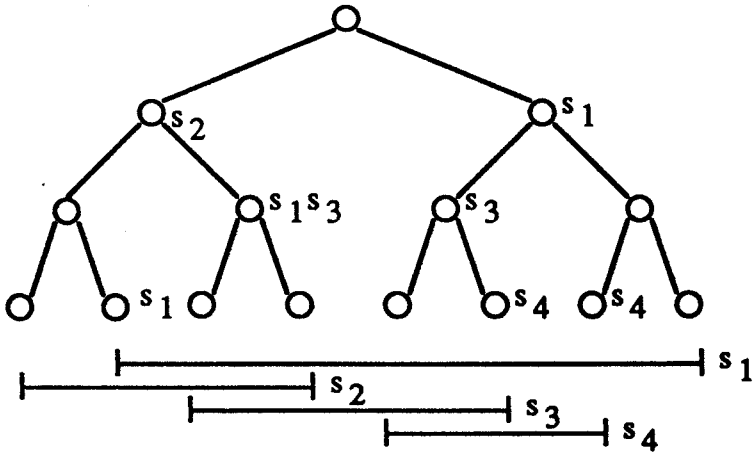

Figure 2: A segment tree.

Finally let us briefly recall the *segment tree*. See [3] for details. Let V be a set of n intervals $[A_i..B_i]$. We want to store these intervals for solving a so-called *stabbing query:* Given a point p, determine all intervals that contain p.

To this end we construct a balanced binary tree on all endpoints of intervals. These endpoints we store in the leaves. With each internal node δ we store the interval $I_\delta = [L_\delta..R_\delta]$ where L_δ is the smallest point in T_δ and R_δ is the largest point in T_δ. We will associate intervals in the set V to internal nodes in the following way: An interval I we associate to δ iff I contains I_δ completely and I does not contain $I_{\text{father}(\delta)}$. Each interval might be associated to more than one node but never to more than $O(\log n)$ nodes. See figure 2 for an example.

To perform a query with a point p we search with p in the tree. For each node δ on the search path we report all intervals associated with δ. In this way all intervals containing p are reported exactly once. See [3] for details.

Theorem 2.6 *([3]) Given a set of n intervals, they can be stored using $O(n \log n)$ storage, such that stabbing queries can be answered in time $O(k + \log n)$ where k is the number of reported answers.*

In fact, this is not the best possible result. Using a structure called an *interval tree* the problem can be solved in time $O(k + \log n)$ using only $O(n)$ storage (see e.g. [7,19]).

3 Dominance searching.

Given two points $P = (P_1, P_2)$ and $P' = (P'_1, P'_2)$ we say that P *dominates* P' iff $P_1 \geq P'_1, P_2 \geq P'_2$ and $P \neq P'$. The *dominance searching problem* asks to store a

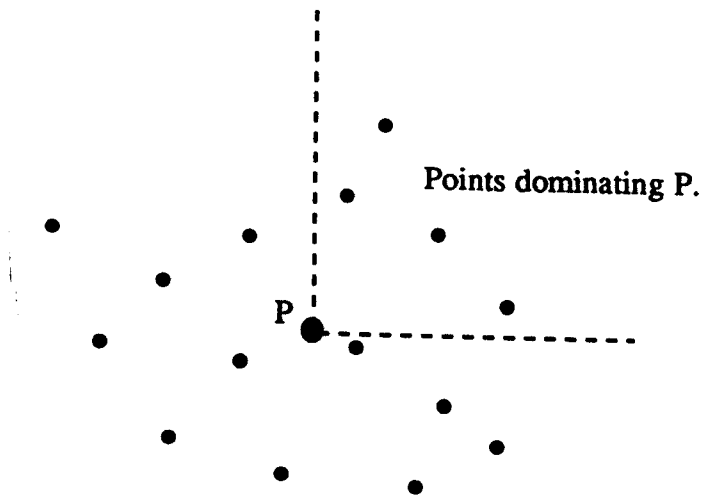

Figure 3: Dominance searching.

set of points V such that for any query point P we can efficiently determine those points that dominate P. In other words, we look for the points in the North-East quadrant from P. See figure 3.

To solve the dominance searching problem on a grid, we store all points in a x-fast trie with respect to their x-coordinate. We transform this x-fast trie into a priority search trie by associating to each internal node the point in its subtree with largest y-coordinate not stored higher up in the trie.

To allow for very efficient searching we store some extra information with the leaves of the trie. With each leaf δ we associate a priority search tree PT_δ containing all points stored at nodes on the search path towards δ. Clearly, PT_δ contains at most $\log U$ points. Moreover, we store a list R_δ of pointers to nodes that are rightson of a node on the search path but do not lie on the search path themselves. These nodes we store in R_δ in decreasing order of y-coordinate of the points stored in the nodes. See figure 4 for an example of the associated information in a leaf.

The x-fast trie itself takes $O(n \log U)$ storage. There are n leaves that each need $O(\log U)$ storage for the PT tree and the list R. Hence, the total structure uses $O(n \log U)$ storage.

To perform a query with a point (x, y) we search for the leaf δ containing the smallest x-value $\geq x$. This can be done in $O(\log\log U)$ time using the special search method for x-fast tries. (See Section 2.) Next we have to check all points stored at nodes on the search path whether they dominate the query point. As we do not visit all these nodes during the search for the leaf, we have to test them in a different way. To this end we use the priority search tree PT_δ stored at δ. (Clearly priority search trees can be used for solving dominance queries.) PT_δ contains the

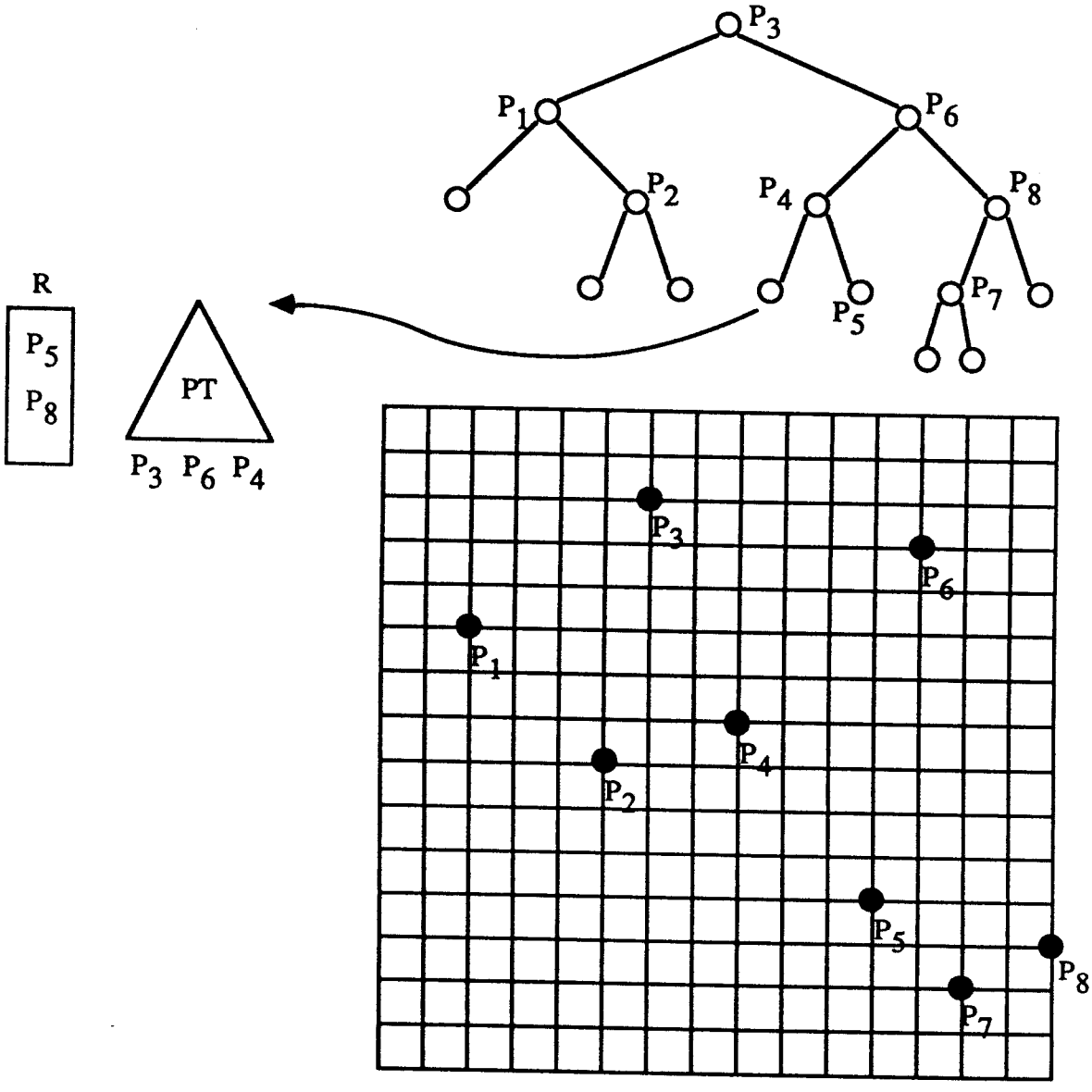

Figure 4: PT_δ **and** R_δ **structures.**

$O(\log U)$ points on the search path. Hence, the query takes $O(\log\log U)$ time (plus the number of answers found).

All other possible answers must lie in a subtree to the right of the search path towards δ. Searching for answers in such a subtree takes time $O(1)$ plus the number of answers found using the same method as described in Section 2. Looking in each subtree for answers would take at least $O(\log U)$ time. As we cannot afford to spend this amount of time we will take care that we only look in subtrees where we will surely find some answers. To this end we use the list R_δ. Let β be the first node in this list. Check the subtree rooted at β for possible answers. If we find none, we are done. All other subtrees to the right of the search path must contain points with y-coordinate smaller than the y-coordinate of the point p stored at β. Hence, if p is not an answer (i.e., it lies below the y-coordinate of the query point) then none of the other points can be an answer. When we do find answers in the subtree rooted at β, we continue with the next node in R_δ, etc. It is easy to see that in this way we find all answers in the subtrees to the right of the search path in time $O(1)$ plus the total number of answers (we look in at most one subtree where we do not find answers).

Theorem 3.1 *Given a set of n points in U^2, we can store them using $O(n \log U)$ storage such that dominance queries can be answered in time $O(k+\log\log U)$ where k is the number of reported answers.*

Proof. Follows from the above discussion. □

To reduce the amount of storage required we use pruning. We replace the x-fast trie by a y-fast trie. This leads to a trie with $O(n/\log U)$ leaves. Each leaf δ corresponds to at most $\log U$ points. These $\log U$ points we also store in the priority search tree PT_δ. Again we associate points to internal nodes. For a number of leaves there will remain some points that are not stored somewhere at internal nodes. These points we store, ordered by y-coordinate, in a list $LIST_\delta$ at δ.

In this way, the tree itself needs $O(n)$ storage. Moreover, each leaf needs $O(\log U)$ storage for PT_δ, R_δ and $LIST_\delta$, and, as there are $O(n/\log U)$ leaves, the total amount of storage required is bounded by $O(n)$.

Queries are performed in almost the same way. First we determine the leaf δ where x comes in and we query PT_δ. Next we look in subtrees to the right of the search path using R_δ. When, while finding answers in a subtree we reach a leaf β, we also have to look at the points in $LIST_\beta$. As the points in $LIST_\delta$ occur in order of y-coordinate this can be done in time $O(\text{number of answers})$. It is obvious that the changes do not increase the query time in order of magnitude.

Theorem 3.2 *Given a set of n points in U^2, we can store them using $O(n)$ storage such that dominance queries can be answered in time $O(k + \log\log U)$ where k is the number of reported answers.*

We assumed above that no two points had the same value in some coordinate. When two or more points have the same y-coordinate there occurs a problem in choosing which node is to be associated to some internal node. There might be more possible choices. It is easy to see that it does not matter which point we take in such a case and that the method described will just run fine. When more points have the same x-coordinate we have to be a bit more careful. A simple method for solving the problem is to store only the point p with highest y-coordinate. The other points with the same x-coordinate as p we store, ordered by decreasing y-coordinate in a list associated to p. Clearly, we only have to report some of these points when we report p. So whenever we have to report p we walk along the list for other answers. All points in the list have their x-coordinate larger than the x-coordinate of the query point. So the only criteria is whether their y-coordinate is larger than the y-coordinate of the query point. So we only have to report some first portion of the points in the list. It is easy to see that this does not increase the query time or amount of storage required.

4 Half-infinite ranges.

We will now adapt the structure described in the previous section to handle half-infinite range queries of the form $([A_1..B_1],[A_2..\infty])$. Again we use a x-fast trie storing all points, ordered by x-coordinate, in its leaves and associate points to internal nodes to turn it into a priority search trie. Also we store structures PT_δ with each leaf δ containing the points on the search path towards δ. But rather than storing one list R_δ we store a number of lists $R_\delta^1, \ldots, R_\delta^{\lfloor \log U \rfloor}$ where R_δ^i contains pointers to nodes on a level $> i$ that are right son of a node on the search path towards δ but do not lie on the search path themselves. These nodes are stored in R_δ^i in decreasing order of the y-coordinate of the associated point. In other words, R_δ^i is R_δ restricted to nodes on levels $> i$. (Level 1 is the root of the trie, level 2 its sons, etc.) In particular $R_\delta^1 = R_\delta$. Similar we store lists $L_\delta^1, \ldots, L_\delta^{\lfloor \log U \rfloor}$ where L_δ^i contains those nodes on level $> i$ that are left son of a node on the search path but do not lie on the search path themselves, again ordered by decreasing y-coordinate.

As each list uses $O(\log U)$ storage and there are $O(\log U)$ lists for each leaf, the total amount of storage required for the structure is bounded by $O(n \log^2 U)$.

To perform a query with range $([A_1..B_1],[A_2..\infty])$ we search with both A_1 and B_1 in the x-fast trie, finding the leaves they come in within $O(\log\log U)$ time. Let these two leaves be δ_1 and δ_2. We query both PT_{δ_1} and PT_{δ_2} to find the points on the search paths that lie in the range. Next, we have to look in the subtrees in between the search paths to locate answers. To do so we first determine the level i at which the search paths of A_1 and B_1 split (go to different sons). This can easily be done in $O(\log\log U)$ time using binary search on the depth of the x-fast trie. The subtrees we have to look in are exactly the ones stored in $R_{\delta_1}^i$ and $L_{\delta_2}^i$.

Both of these two lists we treat in exactly the same way as we treated the list R_δ in Section 3, guaranteeing that we only look in subtrees where we do find answers. This immediately leads to:

Theorem 4.1 *Given a set of n points in U^2, we can store them using $O(n \log^2 U)$ storage such that half-infinite range queries can be answered in time $O(k + \log\log U)$ where k is the number of reported answers.*

Again, we can reduce the amount of storage required using pruning. But this time we have to prune a bit more. We continue the splitting process in the x-fast trie until each leaf δ corresponds to about $\log^2 U$ points. These $\log^2 U$ points we store in PT_δ. Again we associate points to internal nodes and store all points that are left over at δ in a list $LIST_\delta$ ordered by y-coordinate. In this way we obtain a tree with $O(n/\log^2 U)$ leaves, each requiring $O(\log^2 U)$ storage for storing PT_δ, $LIST_\delta$ and the L_δ and R_δ lists.

Queries are performed in the same way. We first find the leaves δ_1 and δ_2 where the search paths for A_1 and B_1 end. We query PT_{δ_1} and PT_{δ_2} which requires $O(\log(\log^2 U)) = O(\log\log U)$ time. Finally we look in the appropriate subtrees. When we reach a leaf during the search in a subtree we have to walk along the stored list $LIST$ as well. It is easy to see that this does not increase the query time in order of magnitude. We conclude:

Theorem 4.2 *Given a set of n points in U^2, we can store them using $O(n)$ storage such that half-infinite range queries can be answered in time $O(k + \log\log U)$ where k is the number of reported answers.*

Handling special cases can be done in exactly the same way as described at the end of Section 3.

5 Range searching.

To solve the range searching problem on a grid we use a variation of the technique of Edelsbrunner[6] as described in Section 2. We store all points, ordered with respect to their y-coordinate, in a x-fast trie. With each internal node δ that is leftson of its father, we store a structure P_δ for half-infinite range queries as described in the previous section, of all points in the subtree T_δ rooted at δ. When δ is right son of its father we associate a similar structure for half-infinite range queries that extend downwards to infinity.

To perform a range query with range $([A_1..B_1], [A_2..B_2])$ we search with A_2 and B_2 in the trie to determine the node δ where the search paths of A_2 and B_2 split. This can easily be done in time $O(\log\log U)$. Next we perform a query on $P_{\text{lson}(\delta)}$ with range $([A_1..B_1], [A_2..\infty])$ and a query on $P_{\text{rson}(\delta)}$ with range $([A_1..B_1], [-\infty..B_2])$. By the same arguments as in [6] this yields the correct answers to the query.

Theorem 5.1 *Given a set of n points on U^2, we can store them, using $O(n \log U)$ storage, such that range queries can be answered in time $O(k + \log\log U)$, where k is the number of reported answers.*

Proof.

The query time follows from the above discussion. At each level each point is stored in exactly one associated stucture. Hence, all associated structures on some level together use $O(n)$ storage. The depth of the trie is $O(\log U)$. The storage bound follows. □

Note that it is essential that the amount of storage required for the half-infinite range query structure is independent of U.

The total amount of storage required can be reduced to $O(n \log n)$. To this end we use a normal balanced binary search tree rather than a x-fast trie as a main structure. As the depth of a tree is bounded by $O(\log n)$, this structure uses $O(n \log n)$ storage. The problem is to determine in $O(\log\log U)$ time the node δ where the search paths of A_2 and B_2 split. This node δ is the highest node in the tree that contains a splitting value inside the range $[A_2..B_2]$. Consider each node δ at height h with splitting value v as a point (v, h). So we are given a set of n points on a $U * \lceil \log n \rceil$ grid and we ask for the highest point with first coordinate between A_2 and B_2. It can easily be seen that the structure described in Section 4 for half-infinite range queries (going upwards to infinity) can be used to solve such a query in time $O(\log\log U)$. (The point we search for is either stored on the search path, i.e., in the PT structures at the leaves our search ends in, or it is stored in the root of the first subtree in the appropriate R or L list. Searching the PT trees in this particular case can be done in time $O(\log\log U)$.) We conclude:

Theorem 5.2 *Given a set of n points on U^2, we can store them, using $O(n \log n)$ storage, such that range queries can be answered in time $O(k + \log\log U)$, where k is the number of reported answers.*

When more points have a same y-coordinate, we just store one of them in the main tree. In the associated structures we, of course, store all of them. It is easy to see that this does not influence the time complexity and amount of storage required.

6 Alternative solutions.

A disadvantage of the method described in the previous sections is that the preprocessing time is very high because of the use of perfect hashing techniques. When we want to reduce the preprocessing time we have to avoid the use of perfect hashing. Perfect hashing is only used to locate the leaf a particular point comes in. We will show that it is possible to do this in different ways.

Let T be the structure for half-infinite range queries on a grid as described in Section 4. We adapt T in the following ways: First of all, we remove all perfect hashing structures. Secondly, we transform the trie into a tree, splitting at each internal node the set in two equal halves, rather than splitting the universe. Hence, we are left with a normal priority search tree but with the extra information at the leaves. Again we do pruning but only such that each leaf contains about $\log^2 n$ rather than $\log^2 U$ points. To be able to determine the level at which the search paths of A_1 and B_1 split, we add to each leaf δ a balanced search tree B_δ storing all nodes on the search path towards δ. Searching with B_1 in the B_δ-tree stored at the leaf δ where A_1 comes in, we can determine the splitting level in time $O(\log\log n)$. This adds $O(\log n)$ storage to each leaf. But because of the pruning, the total amount of storage required for T remains $O(n)$.

The structure can be built in time $O(n \log n)$ by starting at the leaves and working our way upwards, later filling in the associated information at the leaves. When the points are already sorted with respect to x-coordinate the construction can even be caried out in time $O(n)$. Details are left to the reader.

Once we have located the leaves A_1 and B_1 come in, the structure can solve the half-infinite query in time $O(k + \log\log n)$ by the same arguments as in Section 4. So the question is how to locate the leaves. There are different solutions for this. In Section 4 we in fact used a y-fast trie to do this. A very simple method uses an array $LEAF$ of $0..U-1$ that stores for each possible x-coordinate the corresponding leaf in T. Searching with A_1 and B_1 in $LEAF$ takes only $O(1)$ time. This results in:

Theorem 6.1 *Given a set of n points in U^2, there exists a structure to store them using $O(U+n)$ storage that can be built in time $O(U + n\log n)$ such that half-infinite range queries can be answered in time $O(k + \log\log n)$ where k is the number of reported answers.*

Fries e.a.[9] independently obtained a similar result using a related but different technique.

Unfortunately, the amount of storage required is depending on U. This makes it very expensive with respect to storage (only bounded by $O(nU)$) to use the structure as associated structure in the tree for general range queries.

A second possibility is to use a q-fast trie (see Section 2) to locate the leaves. This structure, that takes $O(n)$ storage, makes it possible to find the leaves in time $O(\sqrt{\log U})$. As in general $\sqrt{\log U} > \log\log n$ this leads to the following result:

Theorem 6.2 *Given a set of n points in U^2, there exists a structure to store them using $O(n)$ storage that can be built in time $O(n\log n)$ such that half-infinite range queries can be answered in time $O(k + \sqrt{\log U})$ where k is the number of reported answers.*

When the points are already ordered with respect to their x-coordinate, the y-fast trie can be constructed in time $O(n)$ and, hence, the whole structure can be built in $O(n)$ time.

To solve the general range searching problem we build a binary tree M which stores all points, ordered with respect to their y-coordinate in the leaves. To determine the node where two search paths split, we again associate a structure B_δ to each leaf δ as described above. Again we use a q-fast trie to locate the leaf a particular point goes into. The tree we augment with the new structure for half-infinite range queries in the way described in Section 5.

To perform a query with range $([A_1..B_1],[A_2..B_2])$ we search with A_2 in the q-fast trie to locate its leaf δ in M. Next we search with B_2 in B_δ to locate the splitting node. After we found this node we proceed as in Section 5. It is obvious that such a query takes time $O(k+\sqrt{\log U})$.

To construct the structure we first sort the points by y-coordinate and build the main tree M and the q-fast trie. Next, for each leaf, we collect the nodes on its search path and build the B-structure out of these. To construct the associated structures for the internal nodes, we first sort the points by x-coordinate. Next working our way from the root to the leaves, level by level, we can maintain the points for each node ordered by x-coordinate. In this way, constructing the structures associated to nodes on one level takes time $O(n)$ and, hence, the total construction takes time $O(n \log n)$.

Theorem 6.3 *Given a set of n points in U^2, there exists a structure to store them using $O(n \log n)$ storage that can be built in time $O(n \log n)$ such that range queries can be answered in time $O(k+\sqrt{\log U})$ where k is the number of reported answers.*

7 Generalization to multi-dimensional space.

Up to now we only considered the range searching problem on a two-dimensional grid. To generalize the techniques to a multi-dimensional space we use a method of Willard and Lueker[25]. Assume we are given a set of n points on a d-dimensional grid U^d. A d-dimensional range trie is defined as follows: When $d=2$ it is a trie as described in Section 5 or 6. If $d>2$ we construct a balanced binary tree that contains all points of the set sorted with respect to their first coordinate in the leaves. With each internal node δ we associate a $d-1$ dimensional range trie T_δ of all points stored in the subtree rooted at δ, restricted to their last $d-1$ coordinates.

To perform a range query with query range $([A_1..B_1]\ldots[A_d..B_d])$, we search with both A_1 and B_1 in the tree. In this way we can identify $O(\log n)$ subtrees that together span the interval between A_1 and B_1. On the structures associated with the roots of these subtrees we perform a query with the remaining $d-1$ coordinates of the range. For details on the method see Willard and Lueker[25].

Let $T(d,n)$ be the amount of time required for solving a range query on such a d-dimensional range trie and let $M(d,n)$ denote the amount of space required. It is easy to verify that

$$T(d,n) = O(\log n)T(d-1,n)$$

and

$$M(d,n) = O(\log n)M(d-1,n).$$

Using as 2-dimensional structure the trie described in Section 5, we obtain

Theorem 7.1 *Given a set of n points in U^d, there exists a structure to store the points using $O(n\log^{d-1} n)$ storage such that range queries can be performed in time $O(k + \log^{d-2} n\log\log U)$, where k is the number of reported answers.*

Proof.
Follows from the above discussion. □

As in the 2-dimensional case this structure has a very high preprocessing time. To reduce the preprocessing time we use the structure described in Section 6 for the 2-dimensional trees.

Theorem 7.2 *Given a set of n points in U^d, there exists a structure to store the points using $O(n\log^{d-1} n)$ storage that can be built in time $O(n\log^{d-1} n)$ such that range queries can be performed in time $O(k + \log^{d-2} n\sqrt{\log U})$.*

Proof.
The query time and the amount of storage required follow from the above discussion. The preprocessing time follows from Theorem 6.3 and [25]. □

When $d \geq 3$ we can drop the restriction that points lie on a grid by using a normalization technique related to the method used in e.g. Karlsson and Overmars[14]. So assume we are given a set V of n points in an arbitrary d-dimensional space. We construct d balanced search trees $S_1, \ldots, S_d$. In S_i we store all different values for the i-th coordinate of points in the set V. With each value we store its rank in the tree. Now we replace each point $p = (p_1, \ldots, p_d)$ in the set by $R(p) = (R_1(p_1), \ldots, R_d(p_d))$, where $R_i(p_i)$ is the rank of p_i among the i-th coordinates of all points in the set. We have now got a set $R(V)$ of so-called *normalized* points on a d-dimensional grid n^d. For these points we construct a range trie.

Now assume we want to perform a range with $([A_1..B_1], \ldots, [A_d..B_d])$. For all $1 \leq i \leq d$ we search with A_i and B_i in the tree S_i to determine the smallest value $p_i \geq A_i$ and the largest value $p'_i \leq B_i$. Next, we replace A_i by the rank of p_i and B_i by the rank of p'_i. In this way we normalized our range. It is easy to see that a point p lies in the range if and only if $R(p)$ lies in the normalized range. Hence, to answer the range query, we perform a query with our normalized range

on the normalized set of points. (Of course, with each normalized point we have to store the original point because we have to report the original points, not the normalized ones.)

Theorem 7.3 *Given a set of n points in d-dimensional space ($d \geq 3$), there exists a structure to store the points using $O(n \log^{d-1} n)$ storage such that range queries can be performed in time $O(k + \log^{d-2} n \log\log n)$.*

Proof.

This follows mainly from Theorem 7.1. with the following notes. The d structures S_i together cost $O(dn)$ storage. Because d is a constant this does not influence the overall storage bound. To perform a query we first query the S_i structures. This takes $O(d \log n)$ time. Next we perform a range query on a n^d grid. As $d \geq 3$ the extra $O(d \log n)$ does not influence the overall time bound. □

Theorem 7.4 *Given a set of n points in d-dimensional space ($d \geq 3$), there exists a structure to store the points using $O(n \log^{d-1} n)$ storage that can be built in time $O(n \log^{d-1} n)$ such that range queries can be performed in time $O(k + \log^{d-2} n \sqrt{\log n})$.*

Proof.

Follows in a similar way from Theorem 7.2. □

8 Line segment intersection.

In this section we will apply a similar technique as for range searching to solve the following searching problem: Given a set of n horizontal line segments with end points on a grid, store them such that, given a vertical query line segment s, we can efficiently determine all horizontal line segments that intersect s. This query problem is an important step in for example wire routing problems. Another problem, which is equivalent to it, asks to store a set of axis-parallel rectangles such that, given a query rectangle, one can efficiently determine all rectangle that intersect the query rectangle in the boundary. (Clearly the rectangle problem is a generalization of the line segment problem. On the other hand, the rectangle problem can be solved by using two instances of the line segment problem. One instance stores all horizontal boundaries of the rectangles and we perform two queries with the vertical boundaries of the query rectangle on it. The other stores all vertical boundaries and we perform a query with both horizontal boundaries of the query rectangle.)

To solve the line segment intersection problem we first restrict ourselves to the situation in which the query line segments are half-infinite, i.e., they start at some point and run upwards to infinity.

The data structure we use is a combination of a x-fast trie and a segment tree. All x-coordinates of left and right endpoints of the horizontal line segments we store in the leaves of a x-fast trie. With each internal node δ we associate the interval I_δ covered by the subtree rooted at δ like in ordinary segment trees. A line segment s is stored at a node δ when its x-projection covers I_δ completely but does not cover $I_{\text{father}(\delta)}$. The line segment at a node δ we store in decreasing order of their y-coordinate in a list $LIST_\delta$.

To be able to perform queries efficiently, we associate with each leaf δ a list L_δ that contains all nodes on the search path towards δ in decreasing order of the highest line segment stored at these nodes. In other words, node β appears before node β' in L_δ if the first element in $LIST_\beta$ has a larger y-coordinate than the first element in $LIST_{\beta'}$.

It is easy to see that each segment is stored at $O(\log U)$ nodes. Moreover there are $O(n)$ leaves each requiring $O(\log U)$ storage for the L-list. Hence, the total amount of storage required is bounded by $O(n \log U)$.

To perform a query with a ray upwards, starting at (x, y), we search with x in the x-fast trie for the leaf δ where x comes in. This takes $O(\log\log U)$ time using the special search method for x-fast tries. All segments stored at nodes on the search path towards δ intersect the ray if and only if they lie above y. (Clearly, all segments that intersect the ray are stored somewhere on the search path.) Let β be the first node in L_δ and let s be the first line segment in $LIST_\beta$. When s lies below y we are done. (s is the highest segment in $LIST_\beta$ so no other element in $LIST_\beta$ can lie above y. Moreover, β was the first node in L_δ so all other nodes on the search path towards δ contain segments that lie below s and, hence, lie below y.) When s lies above y, we report it and we walk along $LIST_\beta$, reporting all segments we find, until we reach a line segment below y. We repeat this with the next node in L_δ, etc. It is easy to see that in this way we report all answers in time $O(1)$ plus the number of answers found.

Theorem 8.1 *Given a set of n horizontal line segments in U^2, we can store them using $O(n \log U)$ storage such that those line segments intersecting a given ray running vertically upwards can be determined in time $O(k + \log\log U)$ where k is the number of reported answers.*

The amount of storage required can easily be reduced to $O(n \log n)$. To this end we use as tree rather than a trie as a basic structure. As this tree has a depth of $O(\log n)$ it uses $O(n \log n)$ storage. To locate the leaf our query point goes into we use a separate y-fast trie that stores all x-coordinates of endpoints of line segments with pointers to the corresponding leaves in the tree. This y-fast trie uses only $O(n)$ storage.

To solve the line segment intersection query problem, we construct a x-fast trie of all y-coordinates of the horizontal line segments. This x-fast trie we augment in the following way: To each internal node δ that is left son of its father we associate

a structure P_δ as described above storing all line segments whose y-coordinate lies in T_δ, the subtree rooted at δ. When δ is a right son of its father we associate a similar structure for rays running downwards to infinity. It is easy to see that the structure requires $O(n \log n \log U)$ storage.

To perform a query with a line segment $s = \overline{(x, y_1)(x, y_2)}$ we search with both y_1 and y_2 in the trie to locate the node δ where the search paths split. All answers to the query are stored in $P_{\text{lson}(\delta)}$ and $P_{\text{rson}(\delta)}$. So we can perform a query with $\overline{(x, y_1)(x, \infty)}$ on $P_{\text{lson}(\delta)}$ and a query with $\overline{(x, -\infty)(x, y_2)}$ on $P_{\text{rson}(\delta)}$ to locate these answers.

Finding the splitting node δ takes time $O(\log\log U)$. Both queries take time $O(\log\log U)$ plus the number of answers found. The amount of storage required can easily be reduced to $O(n \log^2 n)$ using the same method as described in Section 5.

Theorem 8.2 *Given a set of n horizontal line segments in U^2, we can store them using $O(n \log^2 n)$ storage such that those line segments intersecting a given vertical line segment can be determined in time $O(k + \log\log U)$ where k is the number of reported answers.*

Also the methods of Section 6 can be applied to the line segment intersection problem to reduce the preprocessing time. Rather than using a y-fast trie to guide the search to the leaf of the augmented segment tree we use a q-fast trie. Details are left to the reader. This leads to the following result:

Theorem 8.3 *Given a set of n horizontal line segments in U^2, there exists a structure to store the line segments using $O(n \log^2 n)$ storage that can be built in time $O(n \log^2 n)$ such that those line segments intersecting a given vertical line segment can be determined in time $O(k + \sqrt{\log U})$ where k is the number of reported answers.*

9 Concluding remarks and open problems.

In this paper we have presented two different approaches for solving the range searching problem on a 2-dimensional grid in a very efficient way. One method resulted in a stucture with a query time of $O(k + \log\log U)$. Unfortunately, due to the fact that the method uses perfect hashing techniques, the preprocessing time is very high. The second method, not based on perfect hashing and, hence, with smaller preprocessing time, obtained a query time of $O(k + \sqrt{\log U})$. Both structures use $O(n \log n)$ storage. Similar techniques were applied to the line segment intersection problem in a set of orthogonal line segments, resulting in very efficient structures as well.

Using standard ways of generalization, also solutions for the multi-dimensional version of the range searching problem were obtained. Applying a normalization technique, these multi-dimensional solutions also work for sets of points in

arbitrary space, rather than on a grid. In this way new solutions to the multi-dimensional range searching problem were obtained that are in some respects faster than any known method when the dimension is at least 3.

Although the methods are optimal (with respect to query time) in some models of computation on a grid (see Karlsson[11]) a number of open problems do remain. First of all, the models of computation in [11] are not very general. Hence, using some special techniques, more storage, etc., improvement might be possible. It is also not clear whether $\Omega(n \log n)$ storage is necessary for obtaining efficient solutions on a grid. Moreover, the structures presented are static. Inserting or deleting a point results in reconstructing most of the structure. An interesting question is whether on a grid dynamic data structures for range searching do exist that are more efficient than known structures in arbitrary space.

The methods presented highly depend on the fact that answers have to be reported. They are clear examples of *filtering search* as introduced by Chazelle[4]. Hence, when we are only interested in e.g. the number of points that lie in a given query range, the methods do not work well. Solutions to such counting problems on a grid are an interesting topic of research.

We have shown that the methods do not only apply to range searching. We succesfully solved the line segment intersection problem as well. There are many more rectangle problems that, in arbitrary space, are solved in a similar way as range searching. A prime example is the question whether a given query point lies in a set of rectangles. No results for these problems on a grid are known.

We have shown that multi-dimensional search problems can be solved more efficiently by first normalizing them to a grid and next solving them on this grid. Similar techniques have succesfully been applied to other geometric problems (see [2,10,14]). This method is worth being studied further.

References

[1] Bentley, J.L., and J.H. Friedman, Data structures for range searching, *ACM Comp. Surveys* 11 (1979), 397-409.

[2] Bentley, J.L., and H.A. Maurer, Efficient worst-case data structures for range searching, *Acta Inform.* 13 (1980), 155-168.

[3] Bentley, J.L., and D. Wood, An optimal worst-case algorithm for reporting intersections of rectangles, *IEEE Trans. on Computers* C-29 (1980), 571-577.

[4] Chazelle, B., Filtering search: A new approach to query-answering, *Proc. 24th Annual Symp. on Foundations of Computer Science,* 1983, 122-132. (to appear in SIAM J. Comp.)

[5] Chazelle, B., Lowerbounds on the complexity of multidimensional searching, *Proc. 27th Annual Symp. on Foundations of Computer Science,* 1986, 87-96.

[6] Edelsbrunner, H., A note on dynamic range searching, *Bull. of the EATCS* 13 (1981), 34-40.

[7] Edelsbrunner, H., *Intersection problems in computational geometry,* Techn. Rep. F93, Inst. f. Information Processing, TU Graz, 1982.

[8] Fredman, M.L., J. Komlos and E. Szemeredi, Storing a sparse table with O(1) worst case access time, *J. ACM* 31 (1984), 538-544.

[9] Fries, O., K. Mehlhorn, S. Näher and A. Tsakalidis, A loglog n data structure for three sided range queries, *Inform. Proc. Lett.* (to appear).

[10] Gabow, H.N., J.L. Bentley and R.E. Tarjan, Scaling and related techniques for geometry problems, *Proc. 16th Annual ACM Symp. on Theory of Computing,* 1984, 135-143.

[11] Karlsson, R.G., *Algorithms in a restricted universe,* PhD thesis, Techn. Rep. CS-84-50, Dept. of Comp. Science, University of Waterloo, 1984.

[12] Karlsson, R.G., and J.I. Munro, Proximity on a grid, *Proc. 2nd Symp. on Theoretical Aspects of Computer Science,* Lect. Notes in Computer Science 182, Springer-Verlag, 1985.

[13] Karlsson, R.G., and M.H. Overmars, *Scanline algorithms on a grid,* Techn. Rep. RUU-CS-86-18, Dept of Computer Science, University of Utrecht, 1986.

[14] Karlsson, R.G., and M.H. Overmars, *Normalized divide and conquer: A scaling technique for solving multi-dimensional problems,* Techn. Rep. RUU-CS-86-19, Dept of Computer Science, University of Utrecht, 1986. (to appear in *Inform. Proc. Lett.*)

[15] Keil, J.M., and D.G. Kirkpatrick, Computational geometry on integer grids, *Proc 19th Annual Allerton Conference,* 1981, 41-50.

[16] McCreight, E.M., Priority search trees, *SIAM J. Computing* 14 (1985), 257-276.

[17] Muller, H., Rastered point location, *Proc. Workshop on Graphtheoretic Concepts in Computer Science (WG 85),* Trauner Verlag, 1985, 281-293.

[18] Overmars, M.H., Range searching on a grid (ext. abstract), *Proc. Workshop on Graphtheoretic Concepts in Computer Science (WG 85),* Trauner Verlag, 1985, 295-305.

[19] Preparata, F.P., and M.I. Shamos, *Computational geometry,* Springer-Verlag, New York, 1985.

[20] Scholten, H.W., and M.H. Overmars, *General methods for adding range restrictions to decomposable searching problems,* Techn. Rep. RUU-CS-85-21, Dept. of Comp. Science, University of Utrecht, 1985. (to appear in *J. Symbolic Comp.*)

[21] van Emde Boas, P., Preserving order in a forest in less than logarithmic time and linear space, *Inform. Proc. Lett.* 6 (1977), 80-82.

[22] Willard, D.E., Log-logarithmic worst-case range queries are possible in space $\Theta(n)$, *Inform. Proc. Lett.* 17 (1983), 81-84.

[23] Willard, D.E., New trie data structures which support very fast search operations, *J. Comput. Syst. Sci.* 28 (1984), 379-394.

[24] Willard, D.E., New data structures for orthogonal range queries, *SIAM J. Computing* 14 (1985), 232-253.

[25] Willard, D.E., and G.S. Lueker, Adding range restriction capability to dynamic data structures, *J. ACM* 32 (1985), 597-617.