RANGE SEARCHING ON A GRID

(extended abstract)

Mark H. Overmars

RUU-CS-85-17a
june 1985

Rijksuniversiteit Utrecht

Vakgroep informatica

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

RANGE SEARCHING ON A GRID

(extended abstract)

Mark H. Overmars

Technical Report RUU-CS-85-17a
june 1985

Department of Computer Science
University of Utrecht
P.O. Box 80.012
3508 TA Utrecht
the Netherlands

RANGE SEARCHING ON A GRID

(extended abstract)

Mark H. Overmars

1. Introduction.

Solutions to searching problems normally are designed to deal with arbitrary precision reals. Hence, the universe from which the objects in the set and the query objects can be taken is infinite. On the other hand, in many applications the universe is bounded and small. For example, in a database we store the age of people or their telephone number. Clearly, both are integers between 0 and some known fixed upperbound.

Assuming the universe is fixed, many searching problems can be solved in a more efficient way using perfect hashing techniques or trie structures. For example, Fredman e.a.[2] have shown that the member searching problem (that asks whether a key is member of a set of keys) can be solved in $O(1)$ query time using $O(n)$ storage (where n denotes the number of keys in the set) when the size of the universe is bounded and known. Willard[8] uses this perfect hashing technique to solve the 1-dimensional nearest neighbor searching problem (that asks for the key in the set nearest to the query object) in $O(\log\log m)$ query time using $O(n)$ storage, where m is the size of the universe. We will shortly recall this solution in section 2 because the rest of the paper is based on it.

Searching problems in a fixed 2-dimensional universe, i.e., a m*m grid have not been studied until recently. Karlsson in his thesis[3] addresses the nearest neighbor searching problem on a 2-dimensional grid. He shows that in the case of a L_1 or L_∞ metric the problem can be solved in O(loglog m) query time using O(n) storage. (See also [4].) See Overmars[7] for an overview of results in computational geometry on a grid.

In this paper we consider the <u>range searching problem</u> on a m*m grid: We are given a set of n points in the plane with integer coordinates between 0 and m-1. These have to be stored such that given a range (i.e., an axis-parallel rectangle) we can efficiently determine which points in the set lie inside (or on the boundary of) the range. We will describe a data structure to store the n points using O(n.log n) storage such that range queries can be performed in O(k+loglog m) time, where k is the number of reported answers. (Compare with the best structures for arbitrary points in the plane that have a query time of O(k+log n) using O(n.log n) storage [1,10].)

The problem is solved in three steps. First we consider the <u>dominance searching problem</u> (also called the ECDF-searching problem) on a grid: Given a set of n grid points, store them such that for a given query point (a,b) those points $p=(p_1,p_2)$ with $p_1 \geq a$ and $p_2 \geq b$ can be determined efficiently. It is the special instance of the range searching problem in which the ranges have the form $([a..\infty],[b..\infty])$. In the non-grid case the best known solution yields a query time of O(k+log n) using O(n) storage (e.g. McCreight[5]). We will give a solution (on a grid) using O(n) storage with a query time of O(k+loglog m).

Next, we consider the range searching problem with ranges of the form $([a_1..a_2],[b..\infty])$. This is called a <u>half-infinite range query</u>. Again, the best known structure yields O(k+log n) query time using O(n) storage. We will reduce the query time to O(k+loglog m) without altering the storage bound. Finally, we adapt a technique of Edelsbrunner[1] to solve the general range searching problem.

One of the disadvantages of the method is that the preprocessing time is very high. In section 4 we will briefly describe a completely different way of solving the problem (based on a structure of Willard[9]) that does not use perfect hashing techniques. It yields a query time of $O(\sqrt{\log m})$ but has much lower preprocessing cost. For more details see the final paper.

2. A very fast trie structure.

In this section we recall some results of Willard[8] that are used in this paper. Given a 1-dimensional set of keys V in a fixed universe of size m we want to store V in such a way that given a key K we can efficiently determine the largest element in V $\leq$K (or the smallest $\geq$K). To this end we store the points in the leaves of a trie. A trie is a binary search tree in which each node corresponds to a part of the universe. The root corresponds to the whole universe and the two sons of a node β correspond to equal halves of the part of β. Clearly, the depth of such a trie is bounded by O(log m) and the structure uses O(m) storage. To reduce the amount of storage required we continue the splitting of the tree until the part of a node β contains 0 or 1 keys. If the part below β contains 1 point we make β into a leaf. The leaves we link in a list. See figure 2.1. for an example. With each level of the trie we store a structure of Fredman e.a.[2] containing all nodes and leaves on this level. Hence, we can determine in O(1) time whether a given node on a given level of the trie is present. The structure clearly takes O(n.log m) storage. Willard[8] calls this structure a x-fast trie.

To perform a query with a key K we will look for the leaf K comes in when searching in the trie, but, rather than searching we use the extra structures to find the leaf faster. To this end we compute which node on the middle level of the trie should contain K in its subtrie. This can easily be done in O(1) time. In another O(1) time we determine whether this node β is present. If it is, the leaf we are looking for lies in the subtrie below β and we continue the search on the bottom half of the trie. Otherwise, the leaf lies in the top

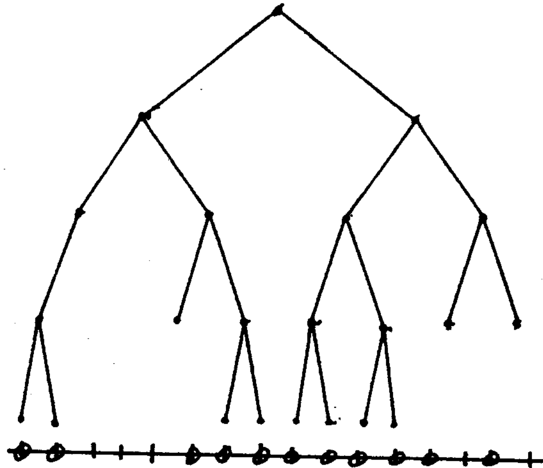

Figure 2.1.

half of the trie and we continue the search there. In this way we locate the leaf in O(loglog m) time. The answer we look for is either stored in this leaf or in its neighbor which we can easily find using the list.

The storage requirement can be reduced to O(n) by pruning the trie. Rather can continuing the splitting until a node contains 1 key we stop when the node contain $\leq$log m keys. These $\leq$log m keys we store in an AVL-tree. To perform a query we first find the leaf (of the trie) in the way described above and, next, search in the AVL-tree at this leaf which takes another O(loglog m) time. This structure is called an y-fast trie. (See [8] for details.)

Theorem 2.1. ([8]) Given a set of n keys V in a fixed universe of size m, we can store them using O(n) storage, such that, given a key K, we can determine in O(loglog m) time the largest key in V $\leq$K.

3. Range queries in the plane.

We will now solve the range searching problem on a 2-dimensional grid. Hence, we are given a set of n point on a m*m grid. We will assume that no two points lie on the same grid line. This restriction

makes the description of the method somewhat easier. The general case can be handled in a similar way. The method we will describe is a combination of techniques of Willard[8], McCreight[5] and Edelsbrunner[1]. For details see the final paper.

3.1. Dominance searching.

To solve the dominance searching problem we store the points in the set V, sorted with respect to their x-coordinates, in a x-fast trie T. Starting at the root of T, we associate with each node β the point p in the subtree rooted at β with highest y-coordinate that has not been stored at a node above β. Figure 3.1.1. gives a running example of a configuration of points on a 16*16 grid. Figure 3.1.2. shows the trie T we obtain with the associated points.

With each leaf p_i we associate two extra structures P_i and R_i. P_i is a priority search trie ([5]) of the (at most) log m points associated with nodes on the search path to p_i. P_i takes O(log m) storage and accommodates dominance queries in O(loglog m) time. R_i is a list containing pointers to the nodes in the tree that are rightson of a node on the search path to p_i but do not lie on the search path themselves. These nodes are stored in order of the y-coordinate of their

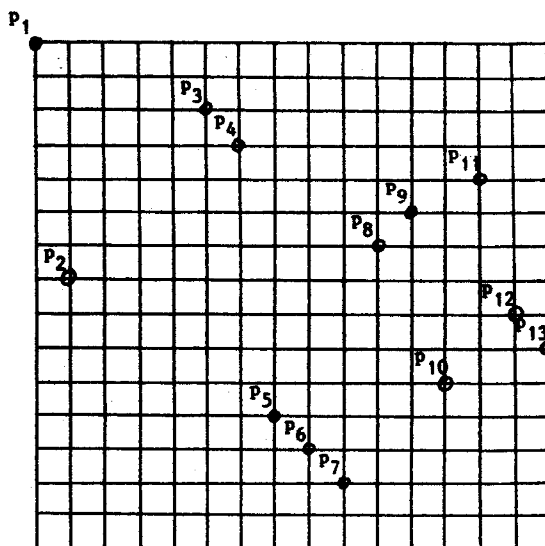

Figure 3.1.1.

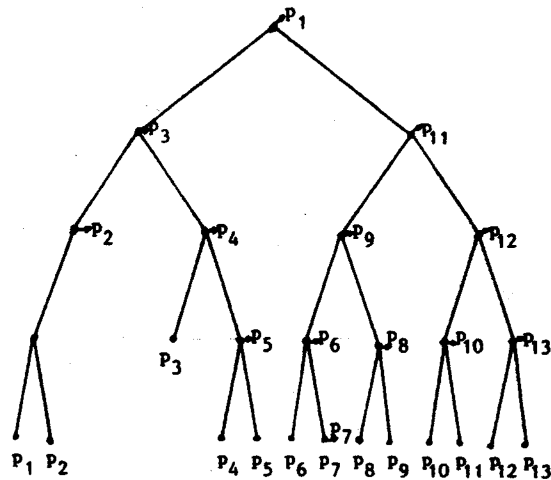

Figure 3.1.2.

associated points. E.g., in our example, P_6 contains the points p_1, p_{11}, p_9 and p_6. R_6 contains the nodes with the points p_8, p_{12} and p_7 in this order. Clearly a list R_i takes at most O(log m) storage.

To perform a query with a point (a,b), we first locate in O(loglog m) time the largest leaf p_i with $p_i \leq a$. All answers to the query are either associated with a node on the search path to p_i or to a node that is part of a subtree to the right of the search path. Using P_i we can determine the answers on the search path. To determine the other answers we have to search in the right subtrees of the nodes on the search path. Like in [5], as soon as we have started looking in such a subtree we can locate all answers in this subtree in time the number of answers found. Unfortunately, we cannot start looking in each of these log m subtrees because this would take to much time when we do not find any answers. To avoid this we use the list R_i. We start looking in the first node β stored in this list. When it does not contain an answer we don't have to look in its subtree because no point in the subtree can lie in the range. And because β was the node with the highest y-coordinate we do not have to look in any other subtree. Otherwise, we look in the subtree below β and continue with the next node in R_i. One can easily see that in this way all answers are found in time O(#answ).

Let us, for example, perform a query with the point (8,8). In this case we find the leaf p_6. Searching in P_6 we find the answer p_9. Next we look at the list R_6. The first node in this list contains p_8. Because this point is an answer we report it and look in its subtree where we find p_9. The next node in R_6 contains p_{12}. This is no answer and we are done.

The structure as described uses O(n.log m) storage. To reduce the storage requirements we use pruning like in section 2 but, rather than using AVL-trees we use priority search trees to store the log m points in a leaf. This leads to the following result.

<u>Theorem 3.1.1</u>. Given a set of n point on a m*m grid, we can store them using O(n) storage such that dominance queries can be answered in O(k+loglog m) time, where k is the number of reported answers.

3.2. Half-infinite ranges.

We will now consider ranges of the form $([a_1..a_2],[b..\infty])$. We use the same structure as described in the previous section except that we associate different lists to the leaves. With each leaf p_i we associate lists $R_i^0..R_i^{\log m}$ and $L_i^0..L_i^{\log m}$. R_i^z is a list of all nodes that are rightson of a node β on the search path where the level of β is $>z$. Similar for L_i^z but with the leftsons. In the lists we store the nodes in order of y-coordinate of the associated points.

To perform a query we locate the leaves p_i and p_j where a_1 and a_2 come into. We search in both P_i and P_j to determine the answers on one of the two search paths. Next we determine the level z where the search paths split. All other answers lie in between the two search paths. We can find these points using the lists R_i^z and L_j^z. Finding the answers now works in exactly the same way as in section 3.1. (For details see the final paper.) The query time is again bounded by O(k+loglog m).

Theorem 3.3.1. Given a set of n points on a m*m grid, we can store them using O(n.log n) storage such that range queries can be answered in O(k+loglog m) time, where k is the number of reported answers.

4. A different approach.

The methods described in the preceding sections have one disadvantage: the preprocessing cost is very high. The reason for this is that the perfect hashing method of Fredman e.a.[2] takes $O(n^3 \log m)$ preprocessing time. As a result there is no hope of making the structures dynamic (not even using techniques like in [6]). In this section we will shortly indicate a completely different approach with a better preprocessing time at the cost of an increase in query time. The method is based on another structure of Willard[9]. We will only describe the structure for dominance searching. The generalization to half-infinite ranges and arbitrary ranges goes in a way related to the methods described in section 3. See the final paper for more details.

Rather than using a binary trie we use a trie in which each node has F(m) sons. Hence, the depth of the trie is bounded by

$$\frac{\log m}{\log F(m)}.$$

With the nodes we associate points in the same way as described in section 3.1. To perform a query with a point (a,b) we search with a down the tree, checking all points we pass on the search path. Because the structure is a trie, for each node on the path we can determine in O(1) time in which son the search must continue. Hence, the search takes time in the order of the depth of the trie.

Next, we have to look into all subtrees that lie right of the search path to search for more answers. Again we cannot afford the time to look into all these trees. To avoid this we will associate lists with internal nodes. With each node β we associate two lists A_β and R_β. A_β contains all sons of β in order of y-coordinate of the

points. R_β contains all right brothers of β in order of y-coordinate. For each node β on the search path we use R_β to determine in which subtrees to the right of β we have to look for answers. Once we are looking in a subtree for answers and have come to a node β we use A_β to determine in which sons we have to continue the search. One easily shows that in this way the searching for answers takes time O(k) plus the depth of the trie.

Each internal node uses O(F(m)) storage. Hence, the total amount of storage is bounded by

$$O(n.F(m).\frac{\log m}{\log F(m)}).$$

To reduce the amount of storage we again use pruning. We continue splitting until a node contains F(m).log m/log F(m) points. These points we store in a priority search tree. In this way the query time becomes

$$O(k+\frac{\log m}{\log F(m)}+\log(F(m).\frac{\log m}{\log F(m)})).$$

Choosing $F(m)=2^{\sqrt{\log m}}$ we obtain a query time of $O(k+\sqrt{\log m})$.

The structure can be built in O(n.log m/log n) using a kind of radix sorting.

> <u>Theorem 4.1</u>. Given a set of n point on a m*m grid, there exists a structure using O(n) storage that can be built in O(n.log m/log n) time such that dominance queries can be answered in $O(k+\sqrt{\log m})$ time, where k is the number of answers.

The same result can be obtained for half-infinite ranges. See the final paper for details. Using the same technique as described in section 3.3. we obtain:

Theorem 4.2. Given a set of n point on a m*m grid, there exists a structure using O(n.log n) storage that can be built in O(n.log n) time such that range queries can be answered in $O(k+\sqrt{\log m})$ time, where k is the number of answers.

5. References.

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

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

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

[4] Karlsson, R.G. and J.I. Munro, Proximity on a grid, Proc. 2nd Symp. on Theoretical Aspects of Computer Science, Lect. Notes in Comp. Science 182, Springer-Verlag, 1985, pp. 187-196.

[5] McCreight, E.M., Priority search trees, SIAM J. Comput. 14 (1985), pp. 257-276.

[6] Overmars, M.H., The design of dynamic data structures, Lect. Notes in Comp. Science 156, Springer-Verlag, 1983.

[7] Overmars, M.H., Computational geometry on a grid: an overview, to appear.

[8] Willard, D.E., Log-logarithmic worst-case range queries are possible in space θ(N), Inform. Proc. Lett. 17 (1983), pp. 81-84.

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

[10] Willard, D.E., New data structures for orthogonal range queries, SIAM J. Comput. 14 (1985), pp. 232-253.