Multidimensional Searching Trees with Minimum Attribute

Haiyan Zhao Zhenjiang Hu Masato Takeichi

The study of data structures for rapid searching is a fascinating subject of both practical and theoretical interest. This paper proposes a new data structure, called k-d-m tree, to address an efficient algorithm for solving the minima searching problem in multidimensional space, which has many applications in areas such as data mining and computational geometry. We will show that with k-d-m tree, not only minima searching can be done very efficiently, but also the existing operations for searching and updating on multi-dimensional tree can be performed as usual. In addition, we demonstrate how a multi-dimensional minima-searching problem can be solved recursively by decreasing the dimensionality.

1 Introduction

The *minima searching problem*, an important computational geometry problem [7] [6], is to determine whether a k-dimensional point is minimal among a set of points. Let S be a set of points in a k-dimensional space. A point p is said to be a minima if it does not dominate any other point in the set S, where a point $(a_1, \ldots, a_k)$ is said to *dominate* a point $(b_1, \ldots, b_k)$, denoted by $(a_1, \ldots, a_k) \geq (b_1, \ldots, b_k)$, if and only if $\wedge_{i=1}^{k}(a_i \geq b_i)$.

This problem arises in many applications. Suppose we have a set of programs for performing the same task rated on the two dimensions of space complexity and time complexity. If we plot these measures as points in the plane, then a point (i.e., program) dominates another only if it is more space efficient and time efficient. We may want to dismiss those programs that do not dominate any other one, and we can use the minima searching to achieve this.

The study of efficient data structure for facilitating rapid minima searching has been a fascinating subject, from the classical structure of heap for one dimensional space [13] to the attempt in the plane [8] [10] [12], and through the research for multidimensional space [2] [3] [4]. However, as far as we are aware, there actually lacks a concrete but efficient solution to the k-dimensional minima searching, although we find it very important for mining optimized association rules from huge databases [14]. Many researchers [8] [10] [12] concentrated their attention only on the maxima (minima) searching in the plane (two dimensional space) and gave their solutions with O$(\log N)$ searching

最小属性値を持つ多次元探索木の提案.

趙 海燕, 東京大学大学院 工学系研究科 情報工学専攻, Department of Information Engineering, University of Tokyo.

胡 振江, 武市 正人, 東京大学大学院 情報理工学系研究科 数理情報学専攻, Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo.

コンピュータソフトウェア, Vol.19, No.1(2002), pp.22–28.
[小論文] 2001 年 8 月 10 日受付.

time, but their methods are difficult to extend to a k-dimensional space for $k \geq 3$. On the other hand, Bentley [5] *conjectured* that k-dimensional maxima (minima) searching could be solved in $\mathrm{O}(\log^{k-1} N)$ time and $\mathrm{O}(N \log^{k-2} N)$ space by using the divide-and-conquer approach, but he provided neither a corresponding data structure nor a detailed algorithm, which has been criticized in [9] [1].

In this paper, we shall present a novel data structure, called *k-d-m tree*, which is an extension of *multidimensional binary searching tree* (*k-d tree* for short) [2] with *minimum attribute*, for efficient minima searching in multidimensional space. We will show that with k-d-m tree, not only minima searching can be done very efficiently, but also the existing operations for searching and updating on multi-dimensional tree can be performed as usual. In addition, we propose a concrete algorithm showing how a multi-dimensional minima-searching problem can be solved recursively by decreasing its dimensionality.

Throughout the paper, we use N to denote the number of the points being considered (in the tree), and k to the dimensionality of the searching space. In addition, we use the notation of Haskell [11], a purely functional language, to describe data structures and algorithms.

The rest of this paper is organized as follows. After reviewing the k-d tree [2] in Section 2, we present k-d-m tree in Section 3. We show how minima searching can be efficiently performed on k-d-m tree in Section 4, and conclude the paper in Section 5.

2 K-d Tree

Before addressing our k-d-m tree, we briefly review the definition of k-d tree firstly proposed by Bentley [2].

k-dimensional binary search tree, or k-d tree, is a binary search tree in which each node contains k keys, left and right pointers to its subtrees, and the discriminator between 1 and k that indicates which key in the node is used for splitting; for a node u having discriminator $i \in \{1, 2, \ldots, k\}$, all nodes in its left subtree are those that their ith key is less than or equal to the ith key in u, and all nodes in its right subtree are those that their ith key is greater than the ith key in u.

Figure 1 gives an example of a 2-d tree where 2 keys are stored in each node, and all nodes with the same distance to the root have the same discriminator as indicated in the discriminator column. The discriminator of a node at distance ℓ from root is $1 + \ell \bmod k$. Notice that 1-d tree is exactly the standard binary search tree.

It has been shown in [2] that insertion, deletion and searching can be efficiently implemented as for the standard binary search tree algorithms, and that k-d tree can be used for a variety of other operations, including orthogonal range searching (to report all points within a given rectangle), partial match queries (to report all points whose values match a given k-dimensional vector with possibly a number of wild cards), and nearest neighbor searching.

Now we can define k-d tree each of whose nodes contains a discriminator and a list of k keys by

```
    type KdTree a = BTree (Dsc,[a])
```

where the binary tree type `BTree a`, whose nodes are of type `a`, and the discriminator type `Dsc` are defined by

```
    data BTree a = Empty
                 | Node a (BTree a)
                          (Btree a)
    type Dsc = Int.
```

Note that for brevity, we assume that all keys have the same type a. Generally, they could have different types, but a simple boxing technique can make them the same.

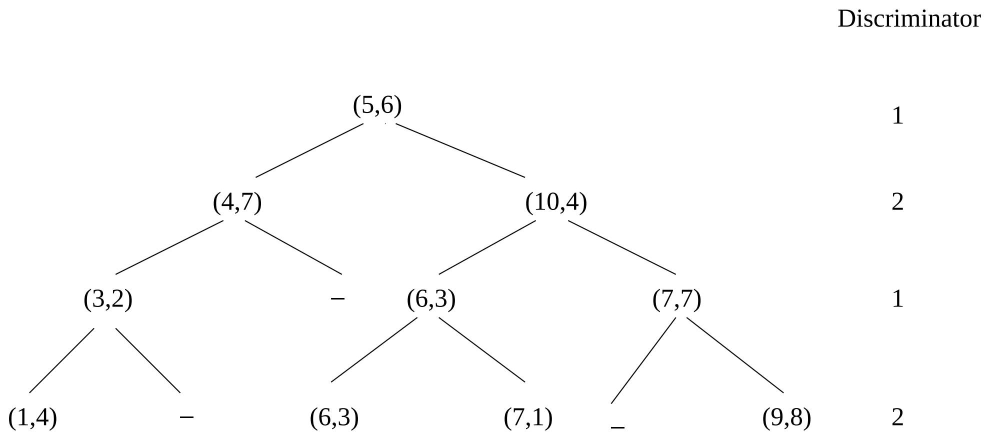

Fig.1 A 2-d Tree

3 K-d-m Tree

As argued in Introduction, although much research has been devoted to minima searching on a plane (2-dimensional space) [8] [10] [12], no concrete data structure and detailed algorithm have been proposed for k-dimensional minima searching using $O(\log^{k-1} N)$ query time, which was conjectured by Bentley in [5]. Since the minima searching on k-d tree is a special case of range searching, one may expect to use range searching for minima searching, but this would have time complexity of $O(\log^k N)$, which is not as efficient as we would expect.

As a matter of fact, minima searching enjoys some important property which may serve for building a more efficient data structure. For the minima searching, rather than searching for all the points falling in a range, we are only interested in deciding whether a point $(a_1, \ldots, a_k)$ is minimal, that is, whether there exists any other point $(b_1, \ldots, b_k)$ in the range where $a_i \geq b_i$ for each $i \in \{1, 2, \ldots, k\}$.

Hinted by the use of discriminator, say i in a k-d tree, for indicating that the ith dimensional key in the node is used for splitting the points, we extend it by associating with each node an additional attribute called *minimum*. The minimum attribute of a node whose discriminator is i in the tree represents the minimum value among the ith key values of all the nodes of the subtree rooted at this node.

Let's look at an example. The k-d tree in Figure 1 is represented by a k-d-m tree in Figure 2, where each node has an additional value indicating its minimum attribute. Since the discriminator of the root node $(5, 6)$ is 1, its corresponding minimum attribute 1 denotes the minimum value of the first keys among all the nodes of this tree, whereas the minimum attribute 1 for the node $(10, 4)$ denotes the minimum value of the second keys among all the nodes of the subtree rooted at $(10, 4)$, because its discriminator is 2.

The definition of the type for k-d-m tree is

```
type KdmTree a = BTree (Dsc, [a], a)
```

where the last component of the node is added to denote the minimum attribute.

We shall be more precise about the invariants of k-d-m tree. Let t be a k-d-m tree whose root node n is $(\mathtt{dsc},\mathtt{keys},\mathtt{m})$, and let $(\mathtt{dsc_l},\mathtt{keys_l},\mathtt{m_l})$ and $(\mathtt{dsc_r},\mathtt{keys_r},\mathtt{m_r})$ be a node in the left and right subtree of the node n respectively. The following summarize the invariants that the k-d-m tree t should satisfy.

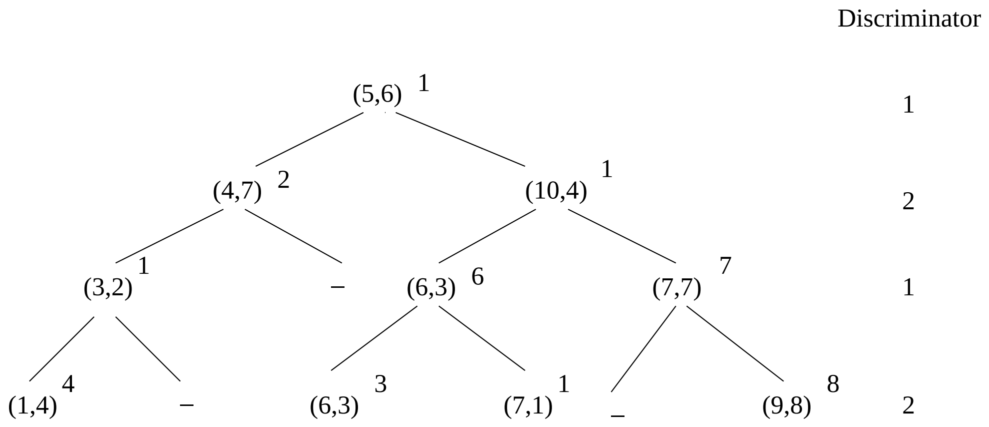

Fig.2 A 2-d-m Tree

- *Splitting Invariants*:

$$\begin{array}{lcl} \texttt{keys}_{\texttt{l}} \texttt{ !! dsc} & \leq & \texttt{keys !! dsc} \\ \texttt{keys !! dsc} & < & \texttt{keys}_{\texttt{r}} \texttt{ !! dsc} \end{array}$$

- *Minimum Invariants*:

$$\begin{array}{lcl} \texttt{keys}_{\texttt{l}} \texttt{ !! dsc} & \geq & \texttt{m} \\ \texttt{keys}_{\texttt{r}} \texttt{ !! dsc} & \geq & \texttt{m} \\ \texttt{keys !! dsc} & \geq & \texttt{m} \end{array}$$

Here we use `keys !! dsc` to return the `dsc`-th key from the key list `keys`. It is worth noting that the splitting invariants are from the requirements of k-d trees while the minimum invariants are related to our minimum attribute. In addition, we always assume that the tree is *balanced* for the sake of efficient implementation of operations like insertion.

Notice that k-d-m tree is a simple extension of k-d tree, and thus many existing operations on k-d tree can be done in a similar way on k-d-m tree. For example, the insertion and deletion can be done very similarly, except for the need to maintain the minimum invariants. It is actually not difficult to keep these invariants by a constant factor in the process of deleting and inserting.

Consider, for example, to insert a new point to a k-d-m tree. To keep the minimum invariants in the process of insertion, we only need to compare the minimum attribute of the current visiting node with the corresponding key of the new point. If the former is greater than the latter, then the former will be updated to the latter. Taking a concrete example, let us insert point $(6, 5)$ to the 2-d-m tree in Figure 2. We first compare the new point $(6, 5)$ with the root node $(5, 6)$. Since the minimum attribute 1 for the root node is less than the counterpart 6, the first key of the new point, we don't need to do anything for this minimum attribute. And according to the splitting invariants for k-d-m tree, the new point should be inserted in the right subtree, which, as we know, can be achieved by recursively calling the subtree. As a result we get the new tree shown in Figure 3, in which the minimum attribute for the node $(7, 7)$ is updated to 6 after the new point is inserted as its left son, since its former minimum attribute 7 is greater than the counterpart 6 of point $(6, 5)$.

In the rest of this paper, we focus on how to efficiently perform minima searching on k-d-m tree.

4 Minima Searching on K-d-m Trees

To get a clear explanation of how minima searching can be done efficiently with k-d-m tree, we start with the 1-dimensional case, and then examine higher dimensional cases.

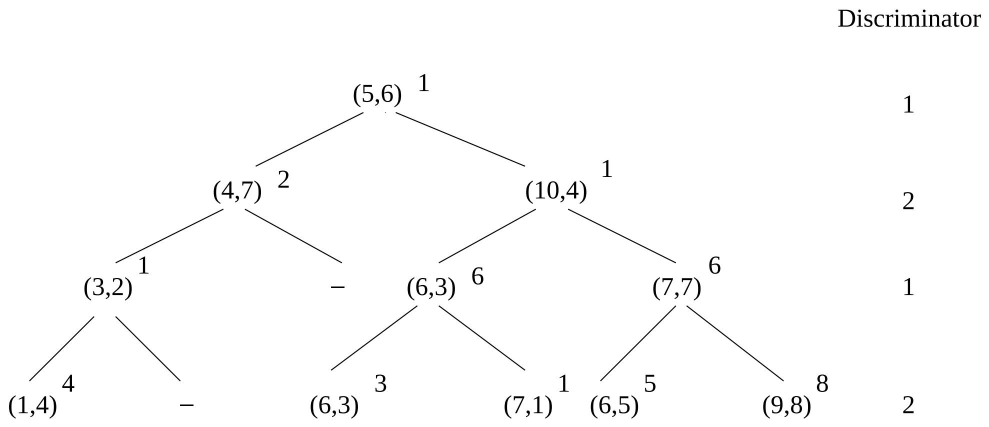

Fig.3 2-d-m Tree after insertion

4.1 Minima Searching in 1-D Space

For the N points in 1-dimensional space, 1-d-m tree is basically the standard binary search tree, except that each node is associated with an additional minimum value. We can determine if a new point p is a minima among the N points just by comparing it with the minimum attribute of the root node. So the time complexity $T(N,k)$ for this case is

$$T(N,1) = \mathrm{O}(1).$$

4.2 Minima Searching in the Plane

For the N points in the plane, 2-d-m tree can help to quickly determine whether a new point is a minima. Since it is a plane where each point is represented by two values [x-value, y-value], the discriminator is only 1 or 2. The following function `minima2 [x,y] t` returns `True` if the point `[x,y]` is a minima among the points stored in the k-d-m tree `t`, and returns `False` otherwise.

```
minima2 :: [a] -> KdmTree a -> Bool
minima2 p Empty = True
minima2 p (Node (d,ks,m) l r)
 = if lessEq ks p then False
   else
     if p!!d <= ks!!d
     then minima2 p l
     else
       if p!!((d+1) ‘mod‘ 2) >= minT l
       then False
       else minima2 p r
```

In the case of an empty tree `Empty`, `minima2 p Empty` returns `True` for any point `p`. In the case of an nonempty tree `Node (d,ks,m) l r`, we check if the keys `ks` stored in the root node `(d,ks,m)` is less than or equal to (i.e., dominated by) the point `p` using the function `lessEq` defined by

```
lessEq :: [a] -> [a] -> Bool
lessEq xs ys
  = and (map (<=) (zip xs ys))
```

If this is true, we know that `p` is not a minima so we return `False` as the result. Otherwise, we do as follows.

- If `p!!d <= ks!!d`, we know from the splitting invariants of k-d-m tree that there does not exist a node with keys of `ks` in `r` such that `lessEq ks p`, so we only need to check `l` to see if `p` is a minima;
- Otherwise, we may need to check both `l` and `r`. The advantage of introduction of the minimum attribute allows us only to check `r`. This can be explained as follows. We check “`p!!((d+1) ‘mod‘ 2) >= minT l`” where `minT` is function to extract the minimum attribute of the root node of a tree.

```
minT :: KdmTree a -> a
minT Empty = infinity
minT (Node (_,_,m) l r) = m
```

If this is true, we know from the minimum invariants of k-d-m tree that there must exist a node (in `l`) whose keys are less than or equal to `p` so we return `False` as the result, otherwise, we are sure that there does not exist a node (in `l`) whose keys are less than or equal to `p`, so we just turn to check `r`.

From the above scheme, we can see that by using the minimum attribute stored in the node, this minima searching is recursively done by going down to its left subtree or right subtree, and the depth of the tree is always decreased by one. Under the assumption that the tree is balanced, the time complexity for this minima searching is

$$T(N, 2) = T(N/2, 2) + \mathrm{O}(1)$$

which is $\mathrm{O}(\log N)$.

It is worth comparing our data structure with those in [8] [10] [12]. For a 2-dimensional tree, different from our associating each node with a minimum x-value or y-value of the tree rooted at the node according to its discriminator, the existing approach associates each node with the same minimum y-value, which makes it difficult to generalize to k-dimensional space. As will be seen later, our k-d-m tree can deal with minima searching of k-dimensional space easily.

4.3 Minima Searching in K-D Space

Rather than showing the concrete algorithm for minima searching in arbitrary dimensional space, we begin by demonstrating how to deal with minima searching in 3-D space, and then highlight how to generalize it to k-D space.

First of all, we would like to show that we can efficiently perform minima searching over a k-d-m tree for a point $(\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_{k-1}, \_)$, where _ denotes "don't care" key. For example, for a 3-dimensional point $(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$, we may be only interested in the first $\mathtt{a}_1$ and $\mathtt{a}_2$ and do not care about $\mathtt{a}_3$. So we may want to see if $(\mathtt{a}_1, \mathtt{a}_2, \_)$ is a minima with respect to the first two keys in a 3-d-m tree. And this minima searching can be done using $\mathrm{O}(\log N)$ time over a 3-d-m tree, in a similar way with that for 2-D minima searching. The idea is to change the algorithm for 2-D minima-searching by extending the scope of nodes for checking: the nodes include not only their children but may also their grandchildren.

With this in mind, now we can solve the minima searching in 3-D space by the following pseudo Haskell program.

```
minima3 p Empty  = True
minima3 p (Node (d,ks,m) l r)
 = if lessEq ks p then False
   else
     if p!!d <= ks!!d
     then minima3 p l
     else
       if not (minima2' p l)
       then False
       else minima3 p r
```

where `minima2' p l` is almost the same as `minima2 p l` except that we ignore the `d`-th key of nodes in the tree.

The time cost for `minima3` is

$$T(N, 3) = T(N/2, 3) + T(N/2, 2) + \mathrm{O}(1).$$

With the result

$$T(N, 2) = \mathrm{O}(\log N)$$

we soon get $\mathrm{O}(\log^2 N)$ as the time cost for minima searching in 3-D space.

The obvious generalization of this algorithm carries through to k-dimensional space without difficulty. We can recursively implement minima searching over k-d-m tree on the basis of that over (k-1)-d-m tree. As a result, the time cost for minima searching over k-d-m tree is

$$T(N, k) = T(N/2, k) + T(N/2, k-1) + \mathrm{O}(1),$$

and we thus get

$$T(N, k) = \mathrm{O}(\log^{k-1} N).$$

We have implemented the minima searching on k-d-m tree in Haskell, which is available at site `http://www.ipl.t.u-tokyo.ac.jp/~zhhy/kdmtree/tstkdmtree.lhs`.

5 Conclusion

In this paper we propose a new data structure k-d-m tree, an extension of k-d tree, on which minima searching can be done efficiently while keeping other operations like insertion and deletion to be almost the same as that on the k-d tree. As far as we are aware, we give the first concrete algorithm for k-dimensional minima searching in $\mathrm{O}(\log^{k-1} N)$ time, which is as efficient as the conjecture given by [5]. And moreover, the space our solution demands is linear, which is much better than $\mathrm{O}(N \log^{k-2} N)$ conjectured in [5].

References

[1] Bentley, J. L., Clarkson, K. L. and Levine, D. B. : Fast Linear Expected-Time Algorithms for Computing Maxima and Convex Hulls, *Annual ACM-SIAM Symposium on Discrete Algorithm*, pp. 509–517, Jan. 1990.

[2] Bentley, J. L. : Multidimensional Binary Search Trees Used for Associative Searching, *Communication of ACM*, Vol. 18, No. 9 (1975), pp. 509–517.

[3] Bentley, J. L. : Multidimensional Binary Search Trees in Database Applications, *IEEE Trans. on SE*, Vol. 5, No. 4 (1979), pp. 333–340.

[4] Bentley, J. L. and Friedman, J. H. : Data Structures for Range Searching, *Computing Surveys, ACM*, Vol. 11, No. 4 (1979), pp. 397–409.

[5] Bentley, J. L. : Multidimensional Divide-and-Conquer, *Communication of ACM*, Vol. 23, No.4 (1980), pp. 214–229.

[6] Chazelle, B. : Computational Geometry: A Retrospective, *Proc. ACM STOC '94*, pp. 75–94, Montreal, Quebec, Canada, May 1994.

[7] Chiang, Y. J. and Tamassia, T. : Dynamic Algorithms in Computational Geometry, *Proceedings of the IEEE, Special Issue on Computational Geometry*, Vol. 80, No. 9 (1992), pp. 1412–1434.

[8] Frederickson, G. N. and Rodger, S. : A New Approach to the Dynamic Maintenance of Maximal Points in Plane, *Discrete and Computational Geometry*, Vol. 5 (1990), pp. 365–374.

[9] Iyengar, S. S., Kashyap, R. L., Vaishnavi, V. K. and Rao, N. S. V. : Multidimensional Data Structures: Review and Outlook, *Advances in Computers*, Vol. 27 (1988), pp. 69–94.

[10] Janardan, R. : On the Dynamic Maintenance of Maximal Points in the Plane, *Information Processing Letters*, Vol. 40 (1991), pp. 59–64.

[11] Jones, S. P. and Hughes, J. (editors) : *Haskell 98: A Non-strict, Purely Functional Language*, Available online: `http://www.haskell.org`, February 1999.

[12] Kapoor, S. : Dynamic Maintenance of Maximas of 2-D Point Sets, *ACM Symposium on Computational Geometry*, pp. 140–148, Stony Brook, New York, June 1994.

[13] Knuth, D. E. : *The Art of Computer Programming, Vol. 3 Sorting and Searching*, Addison-Wesley, 1973.

[14] Zhao, H., Hu, Z. and Takeichi, M. : Mining Optimized Ranges, *The 1st Asian Workshop on Programming Languages and Systems*, pp. 112–122, Singapore, December 18–20, 2000.