数理解析研究所講究録
522巻 1984年 131-145

A HASHING METHOD OF FINDING THE MAXIMA OF A SET OF VECTORS

Masami Machii and Yoshihide Igarashi

町井 優美　　五十嵐 善英

Deparment of Computer Science

Gunma University, Kiryu 376, Japan

1. Introduction

One of the fundamental problems in computational geometry is that of finding the maxima of a set of vectors in d-dimensional space. This problem is closely related to that of finding the convex hull of a polygon which is also fundamental in computational geometry [2][3]. The solutions to these problems can be applied to some problems in mathematical programming or in multi-attribute database systems.

The vector distribution model which we consider in this paper is as follows: Let U_1, U_2, . . . and U_d be totally ordered sets, respectively. Let V be a set of d-dimensional vectors (d-vectors for short) in the Cartesian product $U_1 \times U_2 \times \ldots \times U_d$. For any vector v in V let $x_i(v)$ denote the i-th component of v. A partial ordering $\leqq$ is defined on V in a way that for u, v in V, $u \leqq v$ if and only if $x_i(u) \leqq_i x_i(v)$ for all i = 1,. . ., d, where $\leqq_i$ is the total ordering on U_i. A relation < is defined on V in a way that for u, v in V, $u < v$ if and only if $x_i(u) <_i x_i(v)$ for all i = 1, . . . ,d, where $a <_i b$ means that $a \leqq_i b$ and $a \neq b$ on total ordered set U_i. As in the early literature on finding maxima in a vector set we use a model with some restrictions from a mathematically tractable reason [2][4]. One restriction is that for each vector the magnitude of one component is independent of the magnitude of other components. The second restriction

implies that for the given set of n d-vectors the magnitude of each vector is an integer in the range 1 through n. In order to estimate the expected number of maxima in a vector set or to estimate the expected running time of an algorithm on this kind of problems, a precise specification of vector distribution is required. Bentley et al assumed a further restriction in [2] that each set of n d-vectors corresponds to one of $(n!)^d$ assignments. They estimated the expected number of maxima in a set of n d-vectors that is randomly chosen as one of $(n!)^d$ assignments [2]. Using this result they show an algorithm in linear expected running time of n for finding the set of maxima of a set of the same restricted model [2]. However, we do not assume the last restriction in this paper. That is, we do not assume that for any u and v in the vector set $x_i(u) \neq x_i(v)$ for all i. When we say a set of n d-vectors in this paper, it implies a set of n d-vectors randomly chosen from $\{1, \ldots, n\}^d$.

For v in d-space V, v is defined to be a maximal element (or maximum for short) of V if there does not exist u in V such that $u \geq v$ and $u \neq v$. For v in d-space V, v is defined to be a weak-maximal element (or weak-maximum for short) of V if there does not exists u in V such that $u > v$.

For the d-space we shall prepare m^d linear lists (or appropriate data structures such as AVL trees or B-trees) called hashing cells representing sets of d-vectors, where m is an appropriate integer. Each hashing cell is denoted by $C(i_1, \ldots, i_d)$, where for each j $(1 \leq j \leq d)$ i_j is an integer from $\{1, \ldots, m\}$. For a set of hashing cells we can define maxima and weak-maxima in the same way as for maxima and weak-maxima of a set of vectors. That is, $C(i_1, \ldots, i_d)$ is a maximum (or weak-maximum) of S if and only if $(i_1, \ldots, i_d)$ is a maximum (or weak-maximum) of $\{(k_1, \ldots, k_d) \mid C(k_1, \ldots, k_d)$ is in S$\}$.

We design a hashing algorithm for finding all the maxima in a given

set of n d-vectors chosen from $\{1, \ldots, n\}^d$. The expected running time of the algorithm is $O(n)$. The worst case running time of the algorithm is $O(d\,n\,(\log_2 n)^{d-2})$ which is better than the worst case running time $O(n\,(\log_2 n)^{d-1})$ of the linear expected time algorithm by Bentley et al [2]. That is, our fast expected running time algorithm has also a respectable worst case performance.

2. Finding Weak-Maximal Cells

We prepare m^d hashing cells for implementing our algorithm. Each vector of a given vector set is ditributed to its corresponding hashing cell. Then we consider the set of hashing cells that are "not nill" (i.e., hashing cells that contain at least one vector). We first design an algorithm for finding all the weak-maximal hashing cells (weak-maximal cells for short) of the set of "not nill" hashing cells.

Algorithm 1 (Finding all weak-maximal cells for d = 2).

We prepare m^2 hashing cells $C(i, j)$ $(1 \leq i \leq m, 1 \leq j \leq m)$.

```
begin
1.  distribute all vectors to their corresponding hashing cells (i.e.,
    vector (i, j) is put into C(⌊im/n⌋, ⌊jm/n⌋), where ⌊a⌋ is the smallest
    positive integer not less than a);
    for each (i, j) in {1, . . ., m}^2  if C(i, j) is empty,then the cell
    is marked "nill" and otherwise, the cell is marked "not nill";
2.  SMAX:= ∅;
3.  for j:= 1 step 1 until m do
4.    if C(m,j) is "not nill" then SMAX:= SMAX ∪{C(m,j)};
5.  M:= max{ j| C(m,j) is "not nill"};
6.  for i:= m-1 step -1 until 1 do
      begin
7.      P(i):= max{ k | C(i,k) is "not nill"};
8.      for j:= M step 1 until P(i) do
          begin
9.          if C(i,j) is "not nill" then SMAX:= SMAX ∪ {C(i,j)};
```

```
            end;
10.       M:= max{M, P(i)}
        end;
11.   return SMAX
    end.
```

Theorem 1. The time complexity of Algorithm 1 is $O(\max\{n, m^2\})$, where n is the number of vectors in a given 2-space and m^2 is the number of hashing cells.

Proof. The running time to initialize m^2 hashing cells is $O(m^2)$. We may suppose that all the hashing cells are marked "nill" at the initialized stage. Then line 1 of the algorithm takes O(n) time. The computing time from line 2 to line 11 is O(m). The time complexity of the algorithm is, therefore, $O(\max\{n, m^2\})$. □

For a given set of n 2-vectors, if we can initialize and mark "nill" the m^2 hashing cell in time O(m), then the above algorithm can be implemented in time $O(\max\{n, m\})$. For example, if the m^2 hashing cells are initialized in time O(m) by a parallel machine with m processors in some fashion, then the above algorithm can be implemented in time $O(\max\{m, n\})$. We, therefore, have the next corollary.

Corollary 1. If we do not count the time to initialize m^2 hashing cells, including to mark "nill", then the time complexity of Algorithm 1 is $O(\max\{n, m\})$.

The technique that we used in Algorithm 1 can be extended to solve the general problem of finding weak-maximal cells of a set of d-dimensional hashing cells. To solve this general case we use an algorithm of the maxima searching problem. Bentley devised an efficient algorithm for the maxima searching problem by using the devide-and-conquer method [4]. The maxima searching is the problem how we determine if a new vector is a maximum of a vector set. Although the result which we need here is on weak-maxima, the modification

required by this change is straightforward.

Algorithm 2 (Finding all weak-maximal cells of a set of d-hashing cells).

```
    procedure WMAXI(S, d);
    begin
1.    T1:= {C(i_1, . . . ,i_{d-1}, m)| C(i_1, . . . ,i_{d-1}, m) is in S};
2.    T(m):={ C(i_1, . . . ,i_{d-1}) |C(i_1, . . . ,i_{d-1}, m) is in S};
3     if d = 3 then T2:= the set of weak-maximal cells of T(m) (we can
      compute it by calling Algorithm 1) else T2:=WMAXI(T(m), d-1);
4.    for j:= m-1 step -1 until 1 do
        begin
5.        T(j) := {C(i_1, . . . ,i_{d-1}) | C(i_1, . . . ,i_{d-1}, j) is in S};
6.        for each C(i_1, . . . , i_{d-1}) in T(j) do
            begin
7.            if C(i_1, . . . ,i_{d-1}) is a weak-maximum of T2 ∪ {C(i_1,. . .,
              i_{d-1})} then do
                begin
7.                T1 := T1 ∪ {C(i_1, . . . ,i_{d-1}, j)}
8.              end
            end;
          T2:= {C(i_1,. . .,i_{d-1}) |C(i_1, . . .,i_{d-1},j) is in T1}
        end;
9.    WMAXI:= T1
    end;
    begin (main program)
10.   distribute all vectors of a given set to their corresponding hashing
      cells (i.e., vector (i_1, . . . ,i_d) is put into C(⌊i_1 m/n⌋, . . . ,
      ⌊i_d m/n⌋);
11.   for each (k_1, . . . ,k_d) in {1, . . . ,m}^d, if C(k_1, . . . ,k_d) is
      empty, mark "nill", and otherwise "not nill" the cell;
12.   S:= the set of hashing cells that are marked "not nill";
13.   WMAXI(S, d)
    end.
```

In general the set of hashing cells that are "not nill" does not satisfy the restrictions of vactor distribution described in Section 1. However, Bentley's multidimensional divide-and-conquer technique can be applied to a

wide class of vector distribution models [4]. Bentley's algorithm for the maxima searching problem can be applied with a straightforward modification to line 7 of Algorithm.

Lemma 1. (Bentley [4]) Let n be the number of vectors in a given set on d-space. Then there is an algorithm to determine if a new vector is a maximum (or weak-maximum) of the set in time $O((\log_2 n)^{d-1})$. The preprocessing running time cost for this algorithm is $O(n\ (\log_2 n)^{d-2})$.

The next lemma is not difficult to be proven. The proof is left for the reader as an exercise.

Lemma 2. The number of weak-maximal cells in any subset of the set of m^d d-hashing cells is not greater than $m^d - (m-1)^d$ (i.e., $O(d\, m^{d-1})$), and this upper bound is optimal.

Concerning the weak-maxima searching problem of hashing cells we can show that the running time cost is rather independent of the size of the given hashing cells. The technique to show this fact is essentially the same as the divide-and-conquer method in [4]. We divide a set of hashing cells into two subsets according to values of a certain component that are compared with the middle value of the considered range instead of dividing into two subsets of the same size.

Lemma 3. Let m be the number of hashing ranges of each coordinate of d-space (i.e., there are m^d hashing cells). Then there is an algorithm to determine if a new hashing cell is a weak-mamimum of a given set of hashing cells in time $O((\log_2 m)^{d-1})$. The preprocessing running time cost for this algorithm is $O(m^d (\log_2 m^d)^{d-2})$.

Proof. Our divide-and-conquer method is to choose a hyper-plane at the middle of the range of the considered coordinate dividing the vector set into two subsets A and B, where vectors in A locate on the left of the plane and vectors in B locate on the right of the plane. The other parts of the

algorithm is the same as the Bentley's one [4]. The preprocessing work implies the construction of a multidimensional search tree described in [4]. Since the number of hashing cells in a d-space is at most m^d, the upper bound of the preprocessing work may be denoted by $P(m^d, d)$. Then we have the following recurrence of preprocessing running time $P(m^t, k)$:

$$P(m^t, k) = 2P(m^t/2, k) + P(\min\{m^t/2, m^{k-1}\}, k-1) + O(m^t),$$

$$P(m^t, 2) = O(m^t) \quad \text{and}$$

$$P(q, k) = O(1) \quad \text{for } q \leq 1.$$

The solution to this recurrence is $P(m^t, k) = O(m^t(\log_2 m^t)^{t-2})$ for $t \geq 2$. Therefore, the preprocessing running time is $O(m^d(\log_2 m^d)^{d-2})$.

The searching algorithm of a set of hashing cells is the same as the Bentley's method [4]. We suppose that the search tree is provided by the preprocessing work. To test if a new hashing cell is a weak-maximum we first determine if it lies in the left-half cube A or the right-half cube B on a specific coordinate. If it is in B, then we visit only the right-half cube B. If it is in A, we first test if it is dominated by any hashing cell in A, and if not then we check to see if the projection of the new cell to the plane dividing the cube into A and B is dominated by the projection of any hashing cell in B to the same plane. When we denote the running time cost by Q(m, d) we have the following recurrence:

$$Q(t, k) = Q(t/2, k) + Q(m, k-1) + O(1),$$

$$Q(t, 1) = O(1) \quad \text{and} \quad Q(1, k) = 0.$$

The solution to the recurrence is $Q(m, d) = O((\log_2 m)^{d-1})$. We should notice that the running time cost can be expressed as a function of only m and d. □

Theorem 2. The running time cost of Algorithm 2 is $O(\max\{n, m^{d-1}(\log_2(m^{d-1})^{d-3}, m^d(\log_2 m)^{d-2}\}$.

Proof. The running time cost at line 10, line 11 and line 12 of the

main program is $O(n) + O(m^d)$. The multidimensional search tree for the test at line 7 of procedure WMAXI is not completely constructed as pre-processing work. It is gradually constructed when new elements are inserted into T2. That is, the multidimensional search tree which is used for the test at line 7 is a dynamic datastructure in a sense. However, the method of constructing gradually the multidimensional search tree is essentially the same as the method of constructing the search tree in [4], and the total running time cost for constructing the search tree is the same as the pre-processing time described in Lemma 3. Thus it is $O(m^{d-1}(\log_2(m^{d-1}))^{d-3})$.

Let $T(m^d, d)$ be the upper bound of the running time cost of procedure WMAXI(S, d) excluding the total running time cost of constructing the multi-dimensional search tree of T2, where the number of elements of S is at most m^d. The running time at line 1 and line 2 is $O(m^{d-1})$. The running time of calling procedure WMAXI(T(m), d-1) at line 3 is expressed as $T(m^{d-1}, d-1)$. From Lemma 3 for each hashing cell in T(j) the search test at line 7 takes $O((\log_2 m)^{d-2})$. Therefore, the running time from line 4 to line 8 excluding the running time updating the multidimensional search tree of T2 is $O(m^d(\log_2 m)^{d-2})$. From Lemma 2 the running time at line 9 is $O(d\,m^{d-1})$ (if we use an appropriate datastructure, this bound can be reduced). Thus we have the following recurrence:

$T(m^d, d) = T(m^{d-1}, d-1) + O(m^d(\log_2 m)^{d-2}) + O(m^{d-1}(\log_2 m^{d-1})^{d-3})$, and

$T(m^2, 2) = O(m^2)$ from the proof of Theorem 1. Solving this recurrence we have $T(m^d, d) = O(\max\{m^d(\log_2 m)^{d-2}, m^{d-1}(\log_2 m^{d-1})^{d-3}\})$. Hence we complete the proof. □

Corollary 2. If $\log_2 m \geq d^{d-3}$, then the running time cost of Algorithm 2 is $O(\max\{n, m^d(\log_2 m)^{d-2}\})$.

Corollary 3. If for some constant $k > 0$ and $s > 0$ $k\,m^{d+s} \leq n$, then the time complexity of Algorithm 1 is linear in the number of input vectors.

From the last corollary if we choose, for example, m such that $n = m^{d+1}$, the time complexity of Algorithm 2 is O(n). The last corollary will be used in the next section to show that a hashing algorithm of finding the maxima of a given vector set runs in linear expected time of the number of vectors in the given set. We should note that the fact described in Corollary 3 can be simply shown by a direct application of the Bentley's result in [4]. That is, from the Bentley's result we can simply show that there is an algorithm for finding the set of weak-maximal cells in time $O(\max\{n, m^d(\log_2 m^d)^{d-2}\})$. Therefore, if d^{d-2} is not small, our result of Corollary 2 is stronger than the direct result from the Bentley's result. However, this difference is minor. Algorithm 2 and its time complexity analysis are motivated mainly from theoretical interests.

3. Finding Maximal Vectors

The main algorithm of this paper is to find all maxima of a given d-vector set. The algorithm consists of four parts.

Algorithm 3 (Finding all maximal vectors).

(1) Prepare m^d hashing cells, distribute n vectors into their corresponding cells, and mark each cell "nill" or "not nill" according to the contents of the cells.
(2) Find all weak-maximal cells that are "not nill".
(3) For each weak-maximal cell, find all maximal vectors in the weak-maximal cell.
(4) Merge the maximal vectors of all weak-maximal cells, and construct the set of all maximal vectors of the given set.

The computation for part 1 and part 2 of Algorithm 3 has been already described in the previous section. If the number of maximal vectors in a weak-maximal cell is not greater than some fixed number, we solve part 3 of the above algorithm by a conventional method, and otherwise, we solve it by

calling Algorithm 3 recursively. We now describe how we compute part 4 of Algorithm 3.

We extend the definition of the hashing cell $C(a_1,\ldots,a_d)$ as follows:

(1) If for each j ($1 \leq j \leq d$) a_j is an integer from $\{1, \ldots, m\}$, then $C(a_1, \ldots, a_d)$ is a hashing cell which has been already defined.

(2) Let π be a special symbol which is not an integer. Suppose that for each q ($1 \leq q \leq m$) $C(a_1, \ldots, a_{j-1}, q, a_{j+1}, \ldots, a_d)$ has been already defined. Then $C(a_1,\ldots,a_{j-1},\pi,a_{j+1},\ldots,a_d)$ is defined to be the composite of $C(a_1,\ldots,a_{j-1},1,a_{j+1},\ldots,a_d)$, $C(a_1, \ldots, a_{j-1}, 2, a_{j+1},\ldots,a_d)$,$\ldots$,$C(a_1,\ldots,a_{j-1},m,a_{j+1},\ldots,a_d)$, where a_t ($1 \leq t \leq j-1$ or $j+1 \leq t \leq d$) is an integer from $\{1, \ldots, m\}$ or π.

A composite of hashing cells is called a hashing space. Hereafter, $C(a_1, \ldots, a_d)$ denotes a hashing space. We define $\overline{C}(a_1, \ldots, a_d)$ to be the set of maximal vectors of $\{ v \mid v$ is a vector distributed to any hashing cell of hashing space $C(a_1, \ldots, a_d)\}$. Let $PRO(j, (i_1, \ldots, i_t))$ be a (t-1)-vector obtained from $(i_1, \ldots, i_t)$ by omitting the j-th component. That is, $PRO(j, (i_1, \ldots, i_t)) = (i_1, \ldots, i_{j-1}, i_{j+1}, \ldots, i_t)$ which is a projection of the vector to a hyper-plane. If S is a set of vectors, PRO(j, S) is defined to be $\{PRO(j, v) \mid v$ is in S$\}$. We now are ready to describe the main part of our algorithm.

Algorithm 4 (When $\overline{C}(a_1, \ldots, a_d)$ has been already computed for each $(a_1, \ldots, a_d)$ in $\{1, \ldots, m\}^d$, compute the set of maximal vectors of the given set of vectors). From the definition of a composite of hashing cells $\overline{C}(\pi, \pi, \ldots, \pi)$ is the set of maximal vectors of the given set of vectors.

procedure $\overline{C}(a_1, \ldots, a_j, \pi, \ldots, \pi)$ (each a_t ($1 \leq t \leq j$) is an integer from $\{1, \ldots, m\}$);

```
    begin
1.    if  j = d  then  return  C̄(a_1, . . . ,a_d);
2.    for  k:= m  step  -1 until 1  do
3.      C̄(a_1, . . .,a_j, k, π, . . . , π);
4.    M:= C̄(a_1, . . .,a_j,m, π, . . ., π);
5.    M1:= PRO(j+1, M);
6.    M2:= ∅;
7.    for k:= m-1 step -1 until 1 do
        begin
8.        for each v in C̄(a_1, . . .,a_j, k, π, . . . , π) do
9.          if PRO(j+1, v) is a maximum of M1 ∪ {PRO(j+1,v)} then  do
              begin
10.             M:= M ∪ {v};
11.             M2:= M2 ∪ {PRO(j+1,v)}
              end
12.       M1:= M1 ∪ M2
        end;
13    return  M
    end
    begin (main program)
14.   C̄(π, π, . . . , π)
    end.
```

For the search test at line 9 of Algorithm 4 we must provide an appropriate multidimensional search tree. We do not here describe the details of the search tree. We leave it as an exercise for the reader. We roughly estimate the preprocessing running time for constructing such a search tree to show that Algorithm 3 runs in expected linear time. We first should note that the projection of any vector not in any weak-maximal cell is not inserted into M1. Therefore, the vectors that we should consider in the construction of the search tree is in a weak-maximal cell. From Lemma 2 the number of weak maximal cells is $O(d\,m^{d-1})$. The expected number of vectors in M1 is, therefore, upper bounded by $O(n\,d\,m^{d-1}/m^{d}) = O(d\,n/m)$. However, in the worst case the number of vectors in M1 is $O(n)$. If we first construct an appropri-

ate multidimensional search tree of vectors in the maximal cells, vectors in M1 can be expressed on the whole tree by making some marks on appropriate nodes of the tree. From this observation we have the following lemma.

Lemma 4. Let $n= O(m^{d+s})$ for some constant $s > 0$. Then the preprocessing running time cost to construct an appropriate multidimensional serach tree for the test at line 9 of Algorithm 4 is $O(n^{1-\varepsilon})$ for some $\varepsilon > 0$ in the expexted case, and $O(n(\log_2 n)^{d-2})$ in the worst case.

Proof. From Lemma 1 and the observation descrived above the lemma, the expected running time is $O(d\,n/m\ (\log_2\ (d\,n/m))^{d-2}) < O(n^{1-\varepsilon})$ for some $\varepsilon > 0$, and the worst case running time is $O(n(\log_2 n)^{d-2})$. □

Lemma 5. Suppose that for each $(a_1, \ldots, a_d)$ in $\{1, \ldots, m\}^d$ $\overline{C}(a_1, \ldots, a_d)$ has been already computed and that the multidimensional search tree of vectors in the weak maximal cells has been already prepared. Then the expected and the worst case running time costs of Algorithm 4 is $O(d^2 n (\log_2 n)^{d-2}/m)$ and $O(d\,n\,(\log_2 n)^{d-2})$, respectively.

Proof. All operations in Algorithm 4 are concerned with vectors in weak-maximal cells. Since the algorithm is recursively called at line 2 and line 3, the actual operations on these lines are the same as the operations from line 4 to line 13. Therefore, the running time cost of the algorithm can be evaluated by counting the number of operations for each vector of weak-maximal cells.

For each vectors v in any weak-maximal cell the number of operations concerning v is O(d). Among these operations the search test at line 9 is the most time consuming one. Since the dimension of PRO(j+1, v) at line 9 is d-1, from Lemma 1 each search test for v in a weak-maximal cell takes $O((\log_2 n)^{d-2})$. From Lemma 2 the expected number of vectors in the union of weak-maximal cells is $O(d\,n\,m^{d-1}/m^d) = O(d\,n/m)$. In the worst case this number is n. Therefore, the expected running time cost and the worst case

running time cost of Algorithm 4 are $O(d^2 n (\log_2 n)^{d-2}/m)$ and $O(d n (\log_2 n)^{d-2})$, respectively. □

Theorem 3. Let $n = O(m^{d+s})$ for some constant $s > 0$. Then the expected running time cost and the worst case running time cost of finding the maxima of a set of n d-vectors from $\{1, \ldots, n\}^d$ by Algorithm 3 are $O(n)$ and $O(d^2 n (\log_2 n)^{d-2})$, respectively.

Proof. The computing time of part (1) and part (2) of Algorithm 3 is $O(n)$ from Theorem 2 since we suppose that n is $O(m^{d+s})$ for some $s > 0$. For each weak-maximal cell $C(a_1, \ldots, a_d)$, if the number of vectors in $C(a_1, \ldots, a_d)$ is smaller than a fixed value, say h, then we compute $\overline{C}(a_1, \ldots, a_d)$ by a conventional method, and otherwise, it is computed by calling Algorithm 3 recursively. However, the depth of recursion is limited by t such that $n(d/m)^t \leq h$. This limitation is adopted to avoid the poor worst case performance of the algorithm. We may choose $t = O(d)$ to satisfy the inequality $n(d/m)^t \leq h$. In average case the recursion depth t or less than t is sufficient to compute the maxima of each weak-maximal cell. From Lemma 5 the expected running time cost of Algorithm 4 is less than linear. Therefore, the expected running time cost of Algorithm 4 with the running time cost of finding the maxima of each weak-maximal cell is still less than linear. Hence the expected running time cost of Algorithm 3 is $O(n)$.

If after the recursive call of depth t there are still some weak-maximal cells at this level that have more than h vectors, we apply the Bentley's divide-and-conquer method of finding maxima [4] to these weak-maximal cells. Since the worst case running time cost of the divide-and-coquer method is $O(n(\log_2 n)^{d-2})$ [4], by this switch from the hashing method to the divide-and-conquer method the worst case running time cost of Algorithm 3 becomes $O(d^2 n(\log_2 n)^{d-2})$. □

The hashing method which we have discussed in this paper is particularly efficient when vectors are uniformly distributed in a fixed range. It is a very natural question what m should be chosen for our hashing algorithm. A small s may be recommended in many cases, where s is a positive constant specified in Lemma 4 or Theorem 3. For example, $n = O(m^{d+1})$ may be a good choice. In general d is very small compared with n or m. We therefore may considered that d is a constant. Our hashing method has a very fast average running time as well as a respectable worst case performance.

Refernces

[1] A.V.Aho, J.E.Hopcroft and J.D.Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

[2] J.L.Bentley, H.T.Kung, M. Schkolnick and C.D.Thompson, "On the average number of maxima in a set of vectors and applications", J.ACM, 25, 4, pp.536-543 (1978).

[3] J.L.Bentley and M.I.Shamas, "Divide and conquer for linear expected time ", Information Processing Processing Lett. 7, 2, pp.87-91 (1978).

[4] J.L.Bentley, "Multidimensional divide-and-conquer", C.ACM, 23, 4, pp. 214-229 (1980).

[5] H.T.Kung, "On the computational complexity of finding the maxima of a set of vectors, Proc. 15th Annual IEEE Symp. on Switching and Automata Theory, pp.117-121, 1974.

[6] H.T.Kung, F.Luccio and F.P.Preparata, "On finding the maxima of a set of vectors", J.ACM, 22, 4, pp.469-476 (1975).

[7] F.Luccio and F.P.Preparata, On finding the maxima of a set of vectors, Istituto di Science dell'Informazione, Universita di Pisa, Italy, 1973.

[8] M.H.Overmars, The Design of Dynamic Data Structures, Ph.D Thesis, Department of Computer Science, University of Utrecht, Utrecht, Netherlands, 1983.

[9] L.A.Santalo, Encycropedia of Mathematics and Its Applications, Vol.1: Integral Geometry and Geometric Probability, Addison-Wesley, Reading, Mass., 1976.

[10] F.F.Yao, On finding the maximal elements in a set of plane vectors, Rep. UIUCDCS-R-74-667, Department of Computer Science, University of Illinois, Urbana, Illinois, 1974.