Discovering Large Empty Maximal Hyper-Rectangle in Multi-Dimensional Space

Liang-Ping Ku, Bing Liu, Wynne Hsu
Department of Information Systems and Computer Science
National University of Singapore
Republic of Singapore, 119260
Email: [kulp,liub,whsu]@iscs.nus.edu.sg

Abstract

Given a collection of points in a multi-dimensional space, we consider the problem of finding the set of all possible *Maximal Hyper-Rectangle* (MHR), defined to be hyper-rectangles that are empty and have at least a point bounding each of its surfaces. It is easy to see that there are enormous number of such MHRs in a given instance, and most of the time, applications require only to find the "largest" MHR or"sufficiently large" MHRs. Our proposed algorithm solved all the above problems by setting a criterion to measure *sufficiently large* MHRs so that only those large MHRs will be reported. The algorithm runs much faster when the criterion set is "reasonably tight" as pruning is done naturally in the algorithm.

1 Introduction

Problems occurring in multi-dimensional space have not been studied as extensively as those for one- or two-dimensional (1-D and 2-D) counterparts. One such example is the problem of finding the largest empty rectangle in a 2-D space containing points obstacles [CDL86] [Orl90]. To the best of our knowledge, no equivalent has been found to solve multi-dimensional version of the problem, namely, finding the largest empty hyper-rectangle in a multi-dimensional space.

This problem of finding largest empty hyper-rectangle is applicable in the area of data mining as database of k attributes naturally form a k-dimensional space, and each data tuple forms a point in that space. A detailed description of this application and their related works are presented [LKH97]. By finding the "largest" (or "sufficiently large") empty hyper-rectangle means that there is no data tuple whose attributes fall in the ranges of the empty hyper-rectangle. This has different implications depending on the type of database of interest. For example, in a medical database of patients contracting a specific disease, an empty region within the database would mean that no patient suffering from the disease has attributes within these ranges, and hence it may trigger the discovery of a prevention or cure to the disease.

In another example, in a multi-processor machine, where processors are connected in a hyper-cube mesh, some processors may be busy while some are free. Hence this collection of processors is like a multi-dimensional space with points denoting the busy processors. In the problem of processors assignment, when a job is submitted to the system, the system needs to find a mesh of connected processors of certain size that satisfy the request of the job. Hence, this problem is simply transformed into the problem of finding an empty "hyper-rectangle" in the space of the mesh of processors that satisfy the request.

Many other applications could be listed, for example in material manufacturing such as crystal, plastic, steel, etc. where points could be the faulty points (bubbles, impurities, etc.), and the problem is to salvage as large a block of the material as possible. In floor space planning, points could be pillars or any permanent fixtures, and the problem is to find the largest space for holding certain events or housing certain equipments etc.

To solve this problem, a possible approach is to find the set of all possible empty hyper-rectangles in the space. However, this is an infinite set, and hence infeasible to find. Instead, we propose to find only the set of all possible *Maximal Hyper-Rectangles* (MHR) defined as follows. A *Maximal Hyper-Rectangle* R is a hyper-rectangle such that it is:

Rectilinear : All the sides of R are parallel to the respective axis, and orthogonal to the rest.

Empty : R does not have any points in its interior (We say that R does not *contain* any point).

Maximal : There does not exist another empty hyper-rectangle R' such that the entire hyper-rectangle R falls within the interior of R' (We say that R' does not *contain* R).

It is easy to see that the set of all possible MHRs (which is finite) is representative of all possible empty hyper-rectangle, since any empty hyper-rectangle must be contained within one of the MHR in the set. Furthermore, if we are only interested in the largest or the sufficiently large empty hyper-rectangles, then naturally we can just consider the MHRs defined above.

Now, let us give a formal definition of the problem: Given a k-dimensional space S, where each dimension i is bounded by a minimum and maximum value (denoted by $Smin_i$ and $Smax_i$ respectively), S contains n number of points. The problem is to find all possible MHRs, or the "largest" MHR, or the "sufficiently large" (to be defined later) MHRs in the space S.

In addition, we define a lower (or upper) surface of a MHR R in the i^{th} dimension to be the $k-1$ dimensional space obtained by fixing dimension i of R with the lower (or upper) value of R. A surface is said to be bounded if it contains at least a point or part of the boundary of S. (Boundaries are treated as special points which will be discussed later in Section 2.2.) We call these the bounding points of the surface. Then, we note that an equivalent definition of R being *maximal* is that all its surfaces ($2k$ of them) must be bounded. This is because any possible hyper-rectangle R' that

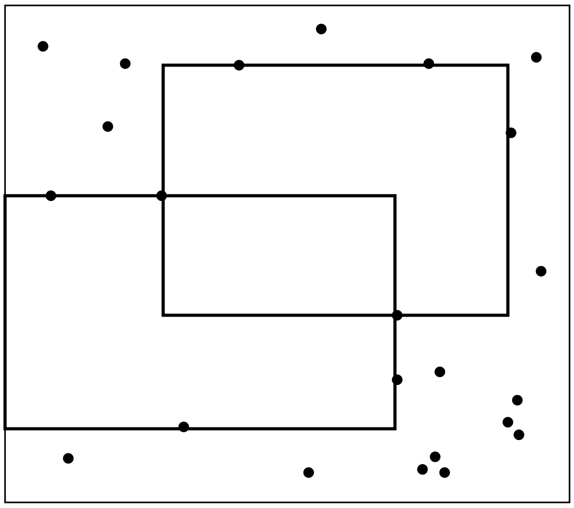


Figure 1: Example of 2 maximal hyper-rectangle (MHR) in a 2-D space

contains R will also contain some of its bounding points, therefore there will not exist any empty hyper-rectangle R' that contains R.

For example, Figure 1 shows two examples of MHR in the 2-D space. Note that each surface (side for this case) of the MHR are bounded by at least one point or a boundary of the space.

The rest of the paper is organized as follows: Section 2.1 presents an overview of the proposed algorithm, followed by a detailed algorithm in Section 2.2. Section 2.3 provides an example which runs through one iteration of the algorithm, while Section 2.4 presents a detailed proof of the correctness of the proposed algorithm. The run time analysis of the algorithm is given in Section 2.5. Experimental results on both random data and real data are presented in Section 3. Directions for further research are given in Section 4 and finally Section 5 concludes the paper.

2 The Proposed Algorithm

2.1 Overview of the Algorithm

The main idea of the algorithm is as follows. Given a k-dimensional space S, we first start with no points in S, hence we have only one MHR, which occupies the entire space of S. Then the n points are incrementally added to S, one point at a time. At each insertion, we update the set of MHRs that have been found this far. The update is done as follows. When a new point is added, we identify all the existing MHRs that contain this point (including those boundary inclusion at the surfaces). Among these identified MHRs, those that contain this new point at their surfaces are updated with the new bounding point, whereas those hyper-rectangles that have the new point in their interior are delete, since they are no longer empty. For each of these deleted hyper-rectangle, using the newly added point as reference, two new empty hyper-rectangle are formed by splitting it at the reference point along each dimension. This splitting is done such that the reference point will be the upper and lower bounding point of the newly created hyper-rectangles in that dimension. If these new empty hyper-rectangles are found to be maximal and sufficiently large, they are inserted into the set of MHRs, otherwise they are discarded. However, if the problem is to find all possible MHRs, then any MHR will be considered sufficiently large by the algorithm.

2.2 The Detailed Algorithm

Given a point X, we let $X(i)$ denote the value of X in the i^{th} dimension. A MHR R is denoted as $R = ((L_1, U_1), (L_2, U_2), \ldots, (L_i, U_i), \ldots, (L_k, U_k))$ where L_i and U_i are respectively the sets of bounding points of the lower and upper surfaces of R along the i^{th} dimension. (A MHR can be uniquely defined by its bounding points.) Hence, the bounding values of R in the i^{th} dimension is $l(i)$ for any $l \in L_i$ and $u(i)$ for any $u \in U_i$ respectively.

Let T denote a data structure that stores a collection of MHRs, and it supports the following functions:

1. $Insert(T, R)$: inserts the MHR R into T.
2. $Delete(T, R)$: deletes the MHR R from T.
3. $ContainmentSearch(T, X)$: returns a list of MHR from T that contains the point X, or bounded by the point X.

The data structure T can be implemented using fairly standard method; first transforming the MHRs into $2k$-dimensional points [PS85] and then storing them in a *Pseudo 2k-d tree* [OL82].

We also defined a function $BigEnough(R)$ which returns TRUE if the MHR R is considered "sufficiently large". Note that the function $BigEnough()$ must satisfy the following: if $BigEnough(R')$ is true, then $BigEnough(R)$ must be true for all R that contains R'.

For simplicity of notation in the algorithm, we let Sl_i and Su_i (for each dimension i) to be points, where $Sl_i(i) = Smin_i$, $Su_i(i) = Smax_i$, and $Sl_i(j)$ and $Su_i(j)$ are undefined for $i \neq j$. Any comparison with $Sl_i(j)$ or $Su_i(j)$ (where $i \neq j$) is always true. Hence, these special points Sl_i and Su_i serve as the *bounding surfaces* of the space S along the i^{th} dimension. The detailed algorithm is best described using pseudo code, and it is given in Figure 2.

2.3 An Example

For an example of one iteration through the algorithm, assume that the points $p1, p2, \ldots, p9$ are already inserted and we are inserting the point $p10$ (refer to Figure 3). Among all the MHRs returned by $ContainmentSearch()$, we run through the iteration for the MHR ACEG for this example. Since the point $p10$ is inside this rectangle, it is deleted from T. Next, we consider splitting ACEG along dimension

Algorithm FindMHR

$Insert(T, R = ((\{Sl_1\}, \{Su_1\}), (\{Sl_2\}, \{Su_2\}), \ldots, (\{Sl_i\}, \{Su_i\}), \ldots, (\{Sl_k\}, \{Su_k\})))$

```
for each point X
  RL = ContainmentSearch(T, X)
  for each R = ((L_1, U_1), (L_2, U_2), ..., (L_i, U_i), ..., (L_k, U_k)) in RL
    if X is on a surface of R
      insert X into the set of bounding points in that surface
    else
      Delete(T, R)
      for each dimension i
        for each j ≠ i
          L'_j = {l ∈ L_j | l(i) < X(i)}
          U'_j = {u ∈ U_j | u(i) < X(i)}
          L''_j = {l ∈ L_j | l(i) > X(i)}
          U''_j = {u ∈ U_j | u(i) > X(i)}
        endfor
        if L'_j and U'_j not empty for all j ≠ i
          and BigEnough(R' = ((L'_1, U'_1), (L'_2, U'_2), ..., (L_i, {X}), ..., (L'_k, U'_k)))
            Insert(T, R')
        endif
        if L''_j and U''_j not empty for all j ≠ i
          and BigEnough(R'' = ((L''_1, U''_1), (L''_2, U''_2), ..., ({X}, U_i), ..., (L''_k, U''_k)))
            Insert(T, R'')
        endif
      endfor
    endif
  endfor
endfor

report all MHRs or the largest MHR in T
```

Figure 2: The proposed algorithm FindMHR in pseudo code.

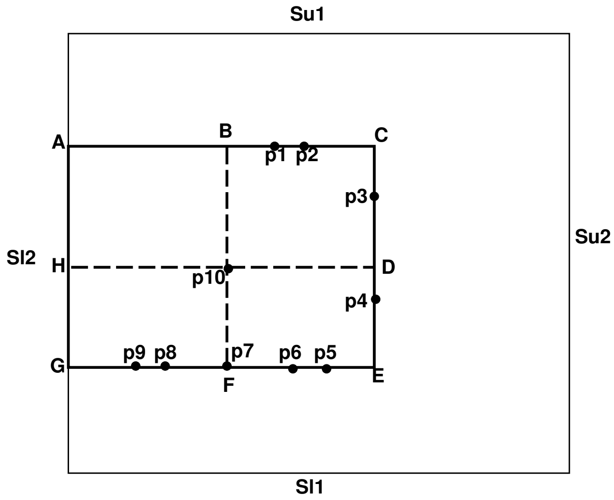


Figure 3: Example to illustrate the algorithm FindMHR on a 2-D space

1. In this case, ACEG (R in the algorithm) is split into ABFG and BCEF (R' and R'' in the algorithm respectively). $L_1' = \{p8, p9\}, U_1' = \{\}, L_1'' = \{p5, p6\}$ and $U_1'' = \{p1, p2\}$ are the sets of bounding points on the line FG, AB, EF and BC respectively. Since U_1' is empty, R' is not inserted into T (i.e. ABFG is not maximal.). On the other hand, L_1'' and U_1'' are not empty i.e. $R'' = ((\{p10\}, \{p3, p4\}), (\{p5, p6\}, \{p1, p2\}))$ (or BCEF) is maximal, and hence it is inserted into T (assuming that it is big enough).

Then, we consider splitting along dimension 2. ACEG is split into HDEG and ACDH (R' and R'' in the algorithm respectively). $L_2' = \{Sl_2\}, U_2' = \{p4\}, L_2'' = \{Sl_2\}$ and $U_2'' = \{p3\}$ are the sets of bounding points on the line GH, DE, AH and CD respectively. Since all are non-empty, $R' = ((\{Sl2\}, \{p4\}), (\{p5, p6, p7, p8, p9\}, \{p10\}))$ and $R'' = ((\{Sl2\}, \{p3\}), (\{p10\}, \{p1, p2\}))$ are maximal and hence inserted into T, (again assuming they are big enough).

This completes one iteration of the algorithm which has processed the MHR ACEG. Similar procedure are done for each R in RL, and for each inserted point X.

2.4 Proof of Correctness

In this section, we prove that the algorithm FindMHR produces *all possible* MHRs that has sizes accepted by the $BigEnough()$ function. (When $BigEnough()$ is set to true for all cases, then the algorithm reports all possible MHRs.) To do this, we have to show that any hyper-rectangle reported by FindMHR *must be maximal* and accepted by $BigEnough()$, **and** any given MHR that is accepted by $BigEnough()$ in the k-dimensional space is reported by the algorithm. The former is easy to show as our algorithm checks each hyper-rectangle before it is inserted into T. Hence we will focus our attention to prove the later.

We first show that our algorithm will return all possible MHRs by assuming that all MHR are "sufficiently large" (i.e. $BigEnough()$ returns true for all MHRs). We then extend the proof to show that the algorithm will return all "sufficiently large" MHR defined by the function $BigEnough()$.

Let us now present some general properties of MHRs.

Lemma 1 *A MHR R in T does not contain the point X in its interior, if and only if R remains as a MHR after X is inserted, or equivalently, a MHR R in T contains the point X in its interior, if and only if R ceased to be a MHR after X is inserted.*

> Proof: If R does not contain X, it remains empty after X is inserted. Hence, it remains as a MHR. Conversely, if R remain as MHR after the insertion, implies that it is empty. Hence it cannot contain X. $\Box$

Lemma 2 *All new MHRs that are formed after the point X is inserted must touch X.*

> Proof: Assume a new MHR R does not touch X, i.e. X is not a bounding point of R. Then before X is inserted, R is already a MHR, hence it contradicts the fact that R is a new MHR. $\Box$

Lemma 3 *All new MHRs that are formed after point X is inserted must be inside the space formed by the union of the MHRs that contain the point X (i.e. MHRs found by $ContainmentSearch()$).*

> Proof: Assume that there exist a new MHR R that is not in the union of the MHRs found by $ContainmentSearch()$. Before X is inserted, R is empty, but not MHR (else it is not new). This implies that there exist another MHR R' that contains R before X is inserted (by definition). However, R' cannot remain as MHR after X is inserted since R is now a new MHR. This implies that R' contains X (by Lemma 1) and hence it must be found by $ContainmentSearch()$. Hence this contradicts the assumption that R is not in the union. $\Box$

Next, we present below the proof of the algorithm, in a sequence of lemmas.

Lemma 4 *All possible new MHRs that are formed after point X is inserted will be inserted into T by the algorithm.*

> Proof: Combining Lemma 2 and Lemma 3, any new MHR must be in the union of the hyper-rectangles found by $ContainmentSearch()$ and must touch X. In the algorithm FindMHR, the MHR R that contains the point X is split along each dimension. This gives all possible MHRs that are inside R which touch X. This is repeated for all MHRs that were found by $ContainmentSearch()$. Hence, all possible MHRs that are inside the union of the hyper-rectangles found by $ContainmentSearch()$ and touch X are constructed by the algorithm, and if they are maximal, they are inserted into T. $\Box$

Lemma 5 *All hyper-rectangles in T that are not MHRs after the insertion of X are deleted from T by the algorithm.*

Proof: By Lemma 1, all hyper-rectangles found to contain X are not MHRs, and they are found by the function $ContainmentSearch()$ and thereafter they are deleted by the algorithm. $\square$

Lemma 6 *All MHRs that are not affected by the insertion of point X are left untouch by the algorithm.*

Proof: By Lemma 1, all MHR that does not contain X remains as MHR. These MHR are not returned by $ContainmentSearch()$, and hence remain untouched by the algorithm. $\square$

Lemma 7 *No identical MHRs can be generated by the algorithm FindMHR.*

Proof: Let R be a newly generated MHR split off from the MHR R' that was found to contain the point X. Assume that there exist an identical R in T before X is inserted. i.e. R and R' are both MHRs before X is inserted. However this is impossible as R' must contains R (since R is split from R'). Hence, there does not exist an identical R in T.
Next, we assume that another instance of R is split off from another MHR R'' that also contains the point X. This means that R' and R'' must have the same boundary for all the sides except for the side cut off by the point X. This in turn implies that R' contains R'' or vice versa. This contradicts that both R' and R'' are MHRs before X is inserted.
Hence, the algorithm will not generate the same instance of MHR more than once. $\square$

Lemma 8 *If T stores all the MHR in the space S before X is inserted, T will be updated by the algorithm to store all the MHR in the space S after X is inserted.*

Proof: By Lemma 4, Lemma 5, Lemma 6 and Lemma 7, the lemma is proved. $\square$

Lemma 9 *The algorithm, upon termination, will store all possible MHR in T.*

Proof: We will prove by induction on the number of points inserted into S. At the base case when there is no points inserted, there is only 1 MHR which occupies the whole space S. Assuming that at a given iteration, T store all possible MHR in S up the last point that was inserted. By inserting the next point, T will stores all possible MHR in S including the newly added point (by Lemma 8). By induction, the lemma is proved. $\square$

We will now extend the proof to show that the algorithm returns all MHRs that are considered "sufficiently large" by the function $BigEnough()$.

Lemma 10 *If a MHR R is not accepted by the function $BigEnough()$, then any MHR R' that is derived from R through a sequence of point insertion (i.e. R' is obtained by a sequence of splitting up of R) will not be accepted by $BigEnough()$.*

> Proof: Since R contain R', and by definition of $BigEnough()$, the lemma is proved. □

Lemma 11 *The algorithm, upon termination, will store all possible sufficiently large MHRs defined by $BigEnough()$ in T.*

> Proof: Since new MHRs generated are split off from existing MHRs, and by Lemma 10, it is safe to discard any MHR that fails the $BigEnough()$ test. Together with Lemma 9, the lemma is proved. □

2.5 Run Time Complexity

T is implemented using a pseudo 2k-d tree [OL82], hence each insert and delete operation on T can be implemented in $O((\log m)^2)$ amortized time, where there are $O(m)$ number of MHR stored in T. Containment search in T is done in $O(\log m + C)$ where there are $O(C)$ number of MHR returned.

A lose bound on the number of MHR (ie m) is $O(n^{2(k-1)})$. This can be shown by construction as follows. Any possible MHR can be constructed by choosing $2(k-1)$ points from the n points, to bound the $k-1$ dimensions, and the bounding points for the last dimension is simply decided from the geometry. Note that this is a very lose bound as many hyper-rectangle constructed this way are not empty, and hence not MHRs.

We also assume that the amount of bounding points per MHR is small, hence the operation to compute L'_j, U'_j, L''_j, U''_j is relatively constant.

Hence the algorithm runs in $O(n(\log m + C) + nC(\log m)^2 + nCk(\log m)^2 + nCk^2)$ which simplifies to $O(nCk(\log m)^2 + nCk^2)$. Since we know that C is of $O(m)$ and m is of $O(n^{2k-2})$, the run time is simplified further to $O(n^{2k-1}k^3(\log n)^2)$.

We note that the complexity is high, but in practice, we expect the complexity to be much lower (see Section 3 for experimental results). We also conjectured that, due to our pruning mechanism using the $BigEnough()$ function, the actual execution time is also related to the density of the space, that is, the algorithm runs faster on a denser dataset. However, density is an increasing function of n, the number of data points. Therefore, the analytical run time will be a complex formula relating n, the $BigEnough()$ function definition, and the density of the dataset.

3 Experimental Results

The algorithm is implemented in C++ and some of our experiments are executed both on a Digital Alpha 8400 workstation under normal load conditions, while others are executed on a Pentium Pro 200 system under light load conditions.

n	Execution Time (in seconds)				Number of MHR m			
	2-D	3-D	4-D	5-D	2-D	3-D	4-D	5-D
50	0	1	4	54	312	2317	12782	110821
100	0	1	19	246	555	5351	48713	412861
150	0	3	63	733	773	10053	110639	1045326
200	0	5	94	1467	962	12991	158871	1883495
250	0	6	160	19504	1155	17226	244223	3115776
300	0	8	244	NA	1168	20937	330759	NA

Table 1: Execution time and number of all possible MHRs for different dataset sizes and dimensions.

3.1 Random Data

In this section, we experimented with randomly generated data. For simplicity of notation, we will refer to the dataset with n points in the k-dimensional space as k-D:n. For example, 4-D:250 is the dataset for 250 points in the 4-dimensional space.

First, we would like to measure the actual running time of the implemented algorithm, as well as to obtain an empirical bound for m, the number of MHRs. For this, we ran our program on 2-D, 3-D, 4-D and 5-D datasets with 50, 100, 150, 200, 250 and 300 data points. All datasets are executed with *BigEnough*() set to true for all MHRs.

Table 1 tabulates the results, namely the actual execution time (in seconds) and the number of MHRs reported for each dataset. (These are executed on a Alpha system.) Looking at the execution time, we notice that 2-D datasets are processed extremely fast (less than 1 second), whereas 5-D datasets may run for hours. Note that the execution of 5-D:300 terminated prematurely as the system runs out of resources. We also observed that, especially for higher dimensions (4-D and 5-D), execution time increase at a faster rate as n increases. For example, the column for execution time for 5-D datasets increases from 54 secs, to 246 secs, to 733 secs, to 1467 secs, and to 19504 secs! This effect is increasingly worse as the number of dimensions increases. (Compare column for 4-D and 5-D). This conforms to our analytical run time bound which is dependent on the number of MHRs, m, which in turn is losely bounded by $n^{2(k-1)}$, a polynomial of linear power to the dimension k. We will not use the actual run time to estimate its bound, as the measured execution time depend heavily on the system load and various parameters of the system such as priority of the job. Instead, we analyze the bound of m. The logarithm graph of m against n is plotted as shown in Figure 4. As expected, they are quite linear. The gradient of each graph is an approximation of the actual value of the power of the polynomial (i.e. $2(k-1)$, or 2, 4, 6, 8 for 2-D, 3-D, 4-D and 5-D respectively). From the graph, the gradient are approximated to be 0.74, 0.96, 1.82 and 2.07 respectively. These values are very much lower than our theoretical bound. Hence, we conclude that in practice with relatively random data, the bound for m of $n^{2(k-1)}$ is never attained. From the experimental results, we also conjectured that there might be a

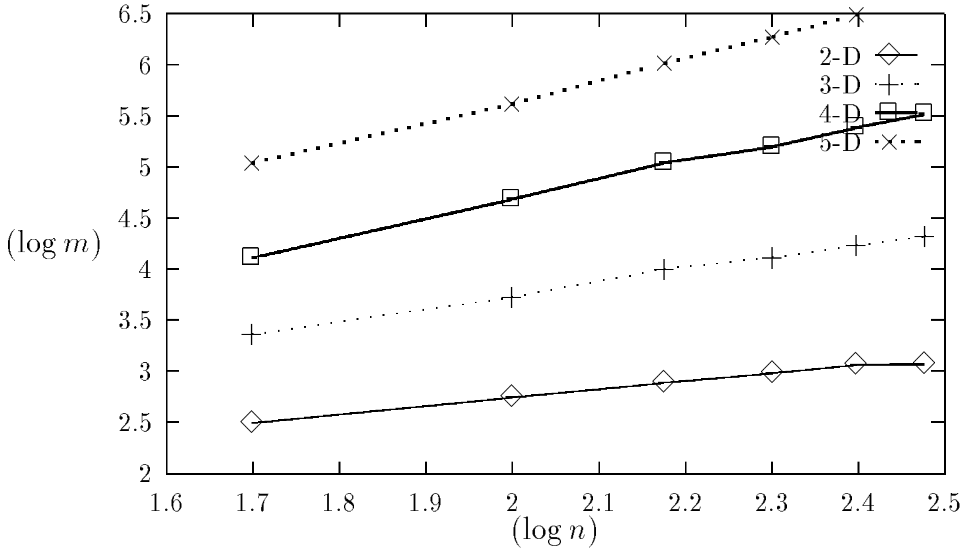


Figure 4: Graph of $(\log m)$ against $(\log n)$.

tighter theoretical worst case bound for m.

Next, we conducted experiments to see the effects of the execution time with respect to the tightness of the function $BigEnough()$. In our experiments, we define $BigEnough()$ to be a function as follows; the function returns TRUE if the MHR has sizes of all its dimensions exceeding the respective given threshold values. These threshold values for each dimension is defined to be a fixed percentage of the full range of values in that dimension (i.e. $Smax_i$ - $Smin_i$ for each dimension i). Note that $BigEnough()$ that uses higher percentage is considered tighter than one that uses lower percentage, since it will reject more MHRs than the later.

We ran our experiments using percentage of 10%, 15%, 20%, 25% and 30%. The dataset used for this experiment is a 4-D:1000.

Table 2 summarizes the results of the experiment. As expected, less MHRs are reported using a tighter $BigEnough()$, and more importantly, the execution time is greatly reduced. This is because the algorithm discard MHRs as and when they fail the $BigEnough()$ test. Hence, a tight function reduces memory requirement for the structure T as well as execution time to process the small MHRs. For example, it takes almost 3 hours to complete for 10% threshold, while 30% threshold requires only 38 seconds. (These are executed on a Alpha 8400 system.) Moreover, about 700 thousands MHRs are reported in the first case. This "overwhelm of information" is a result of setting $BigEnough()$ too lose. These 700 thousands MHRs is useless to one who tries to do data mining to discover holes in databases. On the other hand, setting $BigEnough()$ to 30% returns only 32 MHRs, which is reasonably manageable by humans, and in most application, this is sufficient. Hence, a good approach in using our proposed algorithm is to try with tight $BigEnough()$ functions, and if the

BigEnough() settings	10%	15%	20%	25%	30%
Execution time (Secs.)	10171	726	286	88	38
Number of MHR	701703	241050	48874	14148	32

Table 2: Execution time and number of all MHRs in a 4-D space with 1000 data points, and varying *BigEnough*() settings.

number of MHRs reported is zero, or less than expected, then the function could be incrementally loosened.

In the third experiment, we aim to see the changes of execution time with respect to the density of datasets. As conjectured in Section 2.5, the execution time has a complex relation with the density and number of points.

We ran the experiments by fixing the *BigEnough*() function at 25% (as defined in the previous experiment) and vary the number of points (i.e. density) between 0 and 10000, in a 4-D space. Figure 5 summarized the results in 2 line graphs.

Figure 5(a) shows the number of MHRs. As expected, the number of MHRs increases initially with the number of points, but when the space becomes dense, the number of MHRs decreases as less MHR will satisfy the *BigEnough*() function. The number of MHRs becomes zero when n (number of points) exceeded 4000.

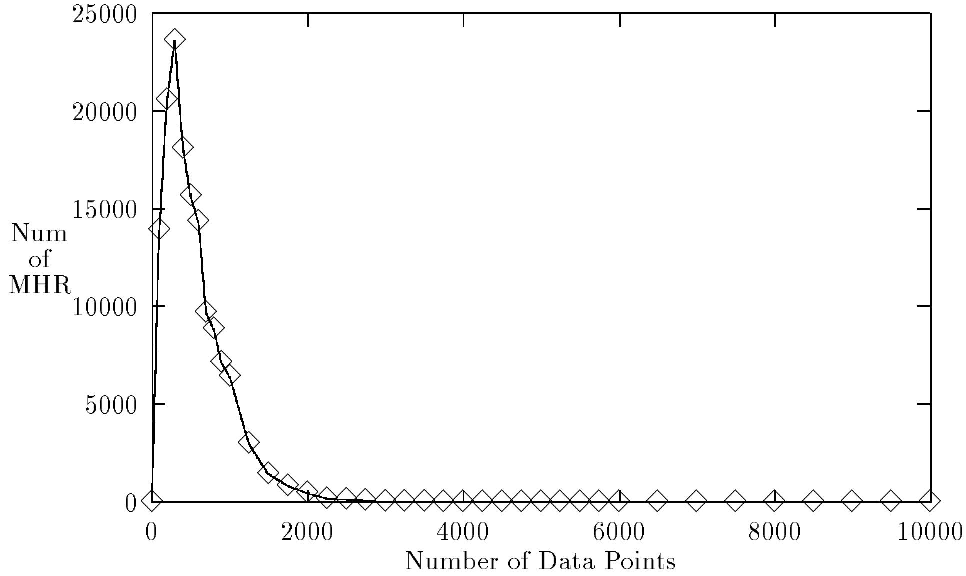
Figure 5(b) shows the execution time. (these runs are executed on a Pentium Pro 200 system). We note that the execution time increases initially and gradually slowing down. When the number of MHRs reaches zero at $n > 4000$ (compare with graph in Figure 5(a)), the execution time plateaus off as n increases, decreasing occasionally (by a few seconds), and then increase again very slowly at the tail of the curve ($n > 6000$). This confirms our suspicion that when the density reaches certain threshold ($n > 4000$), there is some gain in time because the number of MHRs reduces to a small number very quickly. Hence less time is needed to process the subsequent points. However, the gradual increase at the tail ($n > 6000$) shows that when the space is already very dense, any additional point added does not gain much. This is because they do not reduce the number of MHRs in T, (usually already zero), and hence does not help in reducing the processing time of subsequent added points.

This relationship between execution time and the density and the definition of *BigEnough*() is very complex, as it is highly dependent on how *BigEnough*() is defined. Hence, we could not provide an exact formula for such bound.

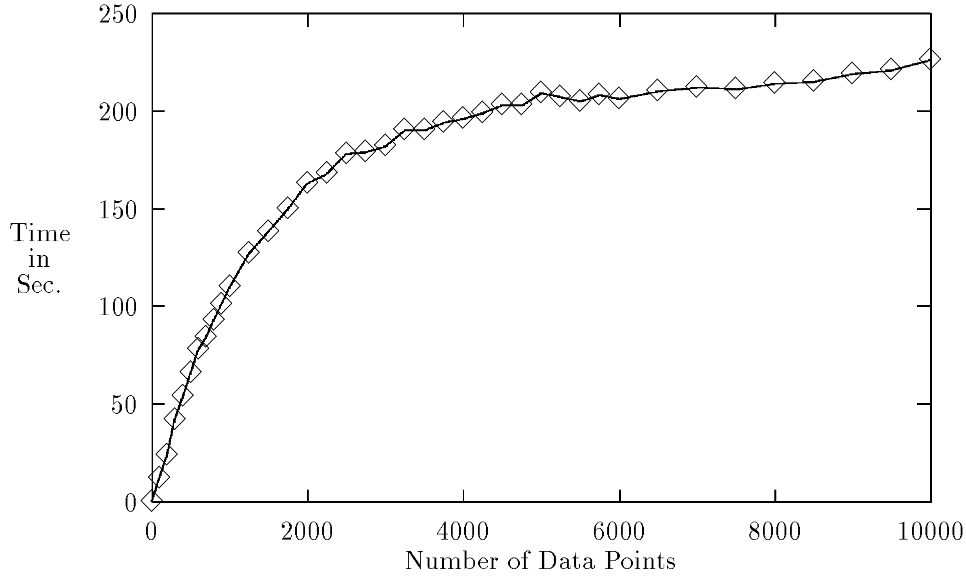
3.2 Real Disease Database

In the fourth set of experiments, we use a real-life disease dataset obtained from a medical doctor. This dataset has 7 continuous attributes, and 713 data points. The algorithm is run 8 times using different attribute combinations, i.e., combining 2, or 3 or 4 attributes. The minimal size of the MHRs in each experiment is specified by the doctor. The running times of all the tests are within 1 or 2 seconds.

In these experiments, many interesting holes are discovered. For example, it is suspected that if the systolic blood pressure (SBP) of a subject is high, then his/her



(a) Number of MHR (m) vs Number of Points (n)



(b) Execution time vs Number of Points (n)

Figure 5: Results of experiments on density

disystolic blood pressure (DBP) is also high. A hole is discovered in the region of high SBP and low DBP. This hole confirms the suspected fact. Some holes are quite unexpected. For instance, the doctor has a strong belief that the higher the SBP, the more likely the subject will get the disease. However, it is found that between the age of 18-50 (the upper bound of age is 69), there is no subject whose SBP is higher than 145 (the upper bound is 231) and is diagnosed to have the disease. There are also other discovered holes that are believed to be due to insufficient data.

4 Further Research

We note that when the dimension is above 5, the program utilizes enormous amount of dynamic memory to store T. As the system resources are being depleted by the program, it resulted in the premature termination of the program. A plausible solution is to consider divide and conquer techniques such that some part of T could be store on disk instead. Hence, further research could be done to modify the algorithm to cater for such large memory requirements.

In another direction, we could look at the problem of "information overwhelm" by the output of our algorithm. For example, in the application of data mining, the users would like to find out the regions (not necessary rectangles) that are empty. These regions are of course overlapped with many of our MHRs, which represent the same general regions. Further work could be done to "summarized" the output of the algorithm (i.e. the set of MHRs found) and present them in some useful and meaningful way to users. [1]

5 Conclusion

The problem of finding all possible MHRs in a multi-dimensional space was studied. We propose an $O(n^{2k-1}k^3(\log n)^2)$ incremental algorithm to solve the problem, as well as a detailed proof of its correctness. The algorithm is implemented in C++ running on Digital Alpha 8400 workstation and Pentium Pro 200 system. Experiments are conducted on random data to measure the run time performance, as well as a feel on the size of m, the number of all possible MHRs. It is also shown that in practice, the execution time is much lower than the analytical worst case bound. Furthermore, if a good $BigEnough()$ is used, the algorithm converges faster to the required results. Lastly, a real disease database is analyzed and some interesting MHR are reported.

[1] Note that in another piece of work, a useful "summary" of the collection of MHR is proposed for the purpose of of decision support in 2-D bin-packing system. Interested readers are referred to [Ku96].

6 Acknowledgements

We will like to thank Dr Tan Tiow Seng of Department of Information Systems and Computer Science, National University of Singapore, for his comments in the early phase of this work. We also wish to thank Dr Lam Kwok Yan of the same department, for running part of our experiments in the machine in his research lab. Special thanks also goes to the system administrators of the department for their support during our experiments.

References

[CDL86] B. Chazelle, R. L. Drydale, and D. T. Lee. Computing the Largest Empty Rectangle. *Siam J. Computing*, 15(1):300–315, 1986.

[Ku96] L.P. Ku. Algorithms for an Incremental Bin-Packing System. *MSc. Thesis, Dept. of Information Systems and Computer Science, National University of Singapore*, 1996.

[LKH97] B. Liu, L.-P. Ku, and W. Hsu. Discovering Interesting Holes in Data. *Proceedings of 15th International Joint Conferernce on Artificial Intelligence*, 1997.

[OL82] M. H. Overmars and J. V. Leeuwen. Dynamic Multi-Dimensional Data Structures Based on Quad- and K-D Trees. *Acta Informatica*, 17:267–285, 1982.

[Orl90] M. Orlowski. A New Algorithm for the Largest Empty Rectangle Problem. *Algorithmica*, 5:65–73, 1990.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry – An Introduction.* Springer Verlag, 1985.