An investigation of multi-level k-ranges

by

Sean M. Falconer and Bradford G. Nickerson

TR04-163, June 30, 2004

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
www: http://www.cs.unb.ca

Abstract

Multi-level k-ranges are an efficient theoretical data structure for range searching introduced by J. L. Bentley and H. A. Maurer. Bentley and Maurer showed that a 1-level k-range offers $O(k \log N + A)$ query time, where k is the number of dimensions, N is the number of data points and A is the number of points matching the query at the expense of $O(N^{2k-1})$ storage. They also introduced the multi-level k-range, which offers slightly slower query time but with $O(N^{1+\varepsilon})$ storage for any fixed values of k and $\varepsilon > 0$.

In this paper, we investigate an implementation of the multi-level k-range data structure. The ℓ-level k-range is compared to naive and R*tree search over N randomly generated k-d points. We show that the R*tree search is significantly faster and that even naive search is faster for most test cases. Results indicate that multi-level k-ranges are not competitive due to their (previously unreported) complexity. Our results indicate that ℓ-level k-ranges require $Q(N, k, \ell) = O((2\ell)^{(k-1)}(\log N + A))$ time for range search. We show that $S(N, k, \ell) = O(N^{1+2(k-1)/\ell})$ and when $N \neq a^{\ell}$ where a and ℓ are positive integers, $S(N, k) = \Theta(N^{1+2(k-1)/\log_2 N})$.

Contents

List of Figures

List of Tables

1 Introduction

The study of data structures that support efficient searching has been one of the fundamental research areas of computer science for many years. Donald Knuth [8] dedicated much of his third edition of *The Art of Computer Programming* to the subject of single key searching. However, only 20 pages are dedicated to the section on secondary key searching, and Knuth points out that very little was known at the time of his writing. Since then, there has been a large amount of research done in the area of "multikey searching" and we now have a large variety of efficient multi-dimensional data structures for addressing this problem.

The specific problem of searching we are concerned with is called "range searching". Range searching is the process of retrieving appropriate records given a query. An *orthogonal range query* is a query that asks for all data with key values within a specified range, that is, data between an upper an lower bound [3]. This can also be phrased in geometric terms, where we are given set F of N points in k-space to preprocess into some data structure. After these points are preprocessed into our given data structure, we answer queries which ask for all points $x \in F$ such that the first coordinate, x_1, is in some range $[L_1, H_1]$, the second coordinate $x_2 \in [L_2, H_2], \ldots$, and $x_k \in [L_k, H_k]$ [4].

Range searching is necessary for a wide range of applications, including databases, geographic data, and computer graphics. In this paper, we will be investigating implementations of a theoretical data structure introduced by Bentley and Maurer [4]. In their paper, they proposed three data structures: k-ranges, multi-level k-ranges, and non-overlapping k-ranges. The first had optimal retrieval time but at the cost of exponential storage. The second, which we implemented, has slightly increased retrieval time but reduced storage. The third has even slower query time, but optimal storage space. They introduced these data structures in order to explore the worst-case complexity of range searching, therefore, the emphasis of their paper was "theoretical", and they ignored the actual performance of the algorithms. We implemented the multi-level k-range in an effort to explicitly verify the asymptotic query time and also explore whether this data structure can be used for any "practical" applications.

2 Data Structures

As mentioned, Bentley and Maurer introduced three new data structures in their paper. In our paper, we will only be concerned with two of these, the 1-level k-range, and the multi-level or ℓ-level k-range. Both of these data structures are static in the sense that they do not support insertion or deletion. In this section, $P(N, k)$, $S(N, k)$, and $Q(N, k)$ denote the preprocessing time, storage space, and range search time respectively.

2.1 One-level k-range

The first step in constructing a k-range[1] is to convert our N real points of set F into the integer space. This is done by sorting each point $x \in F$ by its corresponding k

[1] In this section, we will denote all 1-level k-ranges simply as k-ranges.

dimensions. Let $x = (x_1, x_2, \ldots, x_k)$, then $\overline{x} = (\overline{x_1}, \overline{x_2}, \ldots, \overline{x_k})$ where $\overline{x_i}$ corresponds to the rank of x_i with respect to all other points sorted on the i-th coordinate. Therefore, all points are "normalized" by sorting all dimensions in ascending order and removing duplicates, this can be done in $O(kN \log N)$ with $O(N)$ space.

Furthermore, any query of the form $[L_1, H_1] \ldots [L_k, H_k]$ is normalized as $[\overline{L_1}, \overline{H_1}] \ldots [\overline{L_k}, \overline{H_k}]$ over the points 1 to N. This can be done with $2k \log N$ comparisons [4]. This normalization time was taken into account by Bentley and Maurer in their analysis of the preprocessing and query times of each data structure.

Let us first consider a 1-range. A 1-range is a linear array G of N elements where each element consists of a set of points G_i and a pointer p_i $(1 \leq i \leq N)$. G_i is the set of all points in $\overline{F}$ with first coordinate equal to i, and p_i points to the next nonempty element G_j. For instance, consider the set $\{(1,6),(3,3),(5,1),(5,5),(6,2)\}$, then Figure 1 corresponds to the 1-range storing these points. As we can see, pointer p_1 for element M_1 points to M_3 since M_2 is empty.

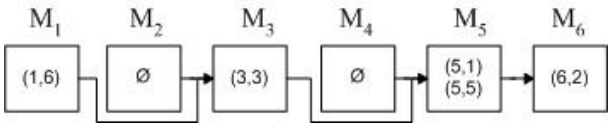

Figure 1: Visual representation of a 1-range [4].

When $k = 1$, the search is simple and takes $O(A)$ time where A is the number of points found. We also have to take into account the normalization time, which gives us the following complexities:

$$P(N,1) = O(N \log N),$$
$$S(N,1) = O(N), \text{ and}$$
$$Q(N,1) = O(\log N + A).$$

Before we can consider $k = 2$ we must first introduce some new notation. For all i, j, t with $1 \leq i \leq j \leq N$ and $1 \leq t \leq k$ let $F_{i,j}^{(t)}$ be the subset of F containing all points whose t-th coordinate is between i and j.

$$F_{i,j}^{(t)} = \{x | x \in F,\ i \leq \overline{x_t} \leq j\}$$

So for $k = 2$, we must store a 2-range of F obtained by storing each of the sets $F_{i,j}^{(2)}$ for $1 \leq i \leq j \leq N$ as a 1-range $R_{i,j}$ and by a 2-d array P of points, with each element $P_{i,j}$ $(1 \leq i \leq j \leq N)$ pointing to $R_{i,j}$. To carry out a range search $[\overline{L_1}, \overline{H_1}], [\overline{L_2}, \overline{H_2}]$, we search in the 1-range $F_{\overline{L_2},\overline{H_2}}^{(2)}$ for points between $\overline{L_1}$ and $\overline{H_1}$. This structure yields the following complexities:

$$P(N,2) = O(N^3),$$
$$S(N,2) = O(N^3), \text{ and}$$
$$Q(N,2) = O(\log N + A).$$

Now let us consider the general case, $k \geq 2$. We store F as follows. First store $F^{(k)}_{i,j}$ as (k-1)-ranges, $R_{i,j}$, with a 2-d array P of pointers, $P_{i,j}$, pointing to $R_{i,j}$. To carry out a range search $[\overline{L_1}, \overline{H_1}], [\overline{L_2}, \overline{H_2}] \ldots [\overline{L_k}, \overline{H_k}]$ in F, we carry out a range search $[\overline{L_1}, \overline{H_1}], [\overline{L_2}, \overline{H_2}] \ldots [\overline{L_{k-1}}, \overline{H_{k-1}}]$ in $F^{(k)}_{\overline{L_k}, \overline{H_k}}$. We continue this process until we search for $[\overline{L_1}, \overline{H_1}]$ in a 1-range. The generalized k-range has the following complexities:

$$P(N,k) = O(N^{2k-1}),$$

$$S(N,k) = O(N^{2k-1}), \text{ and}$$

$$Q(N,k) = O(k \log N + A).$$

As we can see, the k-range's preprocessing and storage is exponential in k. Also, the query time is extremely fast, the $O(k \log N)$ factor is from normalizing the query, and $O(A)$ is the number of points reported. The query time makes this data structure attractive, however, the preprocessing and storage is too large to realistically consider this for practical applications.

2.2 Multi-level k-range

In the previous section we discussed k-ranges, which have extremely efficient range searching times with the expense of exponential storage and preprocessing. The k-ranges explicitly stored all possible queries, or a complete covering over each coordinate interval, this lead to the exponential storage. In this section we will discuss the multi-level k-range, which is a modification of the k-ranges designed to reduce storage and preprocessing time.

Let us first consider the 2-level structure for a 2-level 2-range. On the first level, we consider one "block" which contains $N^{1/2}$ "units", each containing $N^{1/2}$ points each. In Bentley and Maurer's paper, they assumed N is a perfect square, in our actual implementation we take the ceiling of this value, but for now, assume N is a perfect square. In the first level we store all $C(N^{1/2}+1, 2) = O(N)$ consecutive intervals of units, that is $O(N)$ 1-ranges, where $C(n,p)$ is the number of ways to choose p element subsets from a set of n elements. Rather than storing 1-ranges as a linked-list of N points as in the k-ranges, we store the points in an array or vector sorted on the x-value, this way we have storage proportional to the number of points stored in the range. In the second level, we have $N^{1/2}$ blocks, each containing $N^{1/2}$ units. Within each block, we store all possible intervals of units as 1-ranges. This structure can be seen in Figure 2 for $N = 9$. In the figure, the bold vertical lines represent block boundaries and the regular vertical lines represent unit boundaries. Each horizontal line represents 1-ranges; these 1-ranges do not extend across block boundaries.

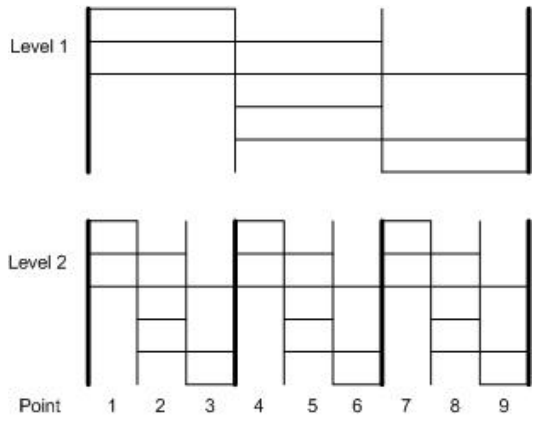

Figure 2: Visual representation of a 2-level 2-range [4].

To perform a range search on a 2-level 2-range we must choose a covering of the y-range from both the first and second level. This can be done by selecting at most one covering from level 1 and two from level 2. We can see this in Figure 3, where the crossed-hatched sections represent the coverings from each level, these coverings represent the ranges we must search.

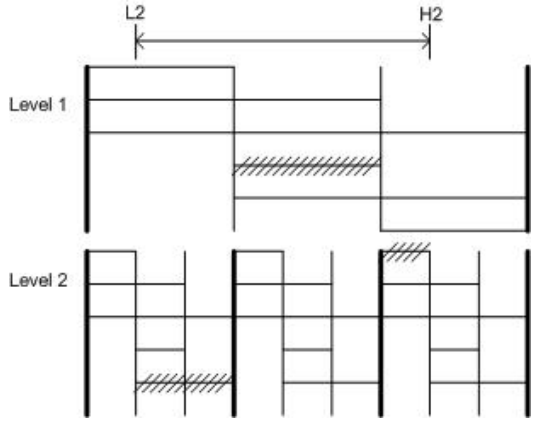

Figure 3: Example of searching a 2-level 2-range [4].

Example 2.2.1: 2-level 2-range
Consider the following example of constructing and searching a 2-level 2-range.

Let $F = \{(8,6),(18,28),(2,1),(21,86),(32,34),(3,18),(7,13),(22,146),(51,62)\}$.

Sort these points on both the x and y coordinate values:

Rank:	1	2	3	4	5	6	7	8	9
Sorted on x:	2	3	7	8	18	21	22	32	51
Sorted on y:	1	6	13	18	28	34	62	86	146

Now $\overline{F} = \{(4,2),(5,5),(2,1),(6,8),(8,6),(2,4),(3,3),(7,9),(9,7)\}$. This allows us to build the following 2-level 2-range.

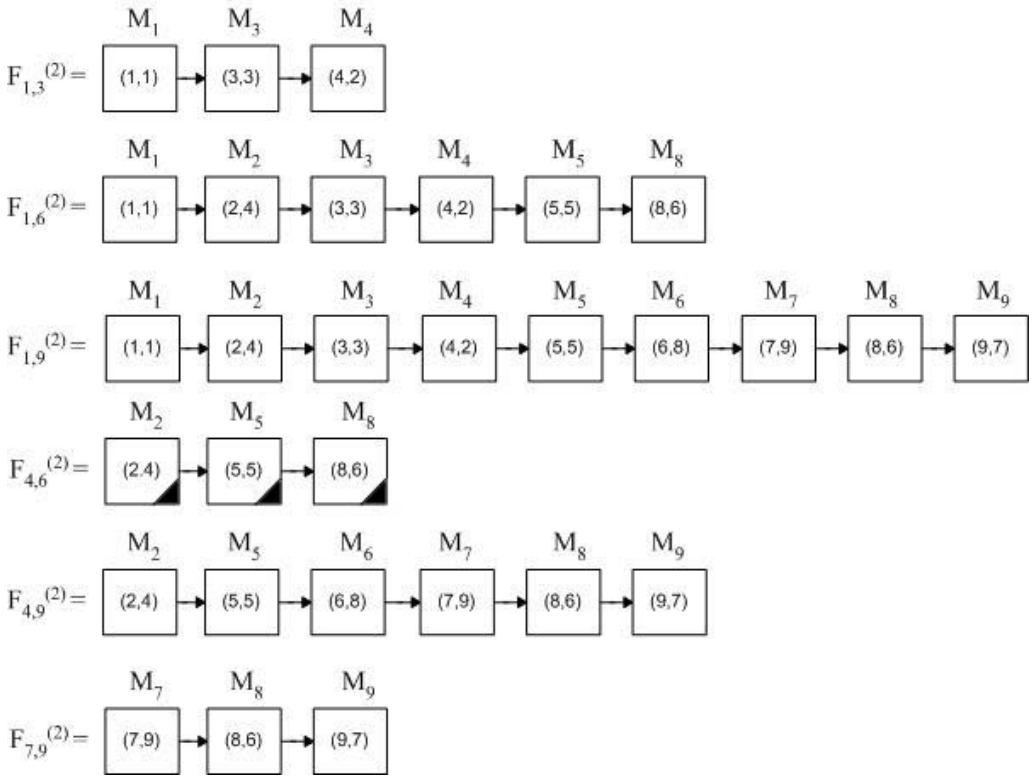

Figure 4: Level 1 of example 2-level 2-range.

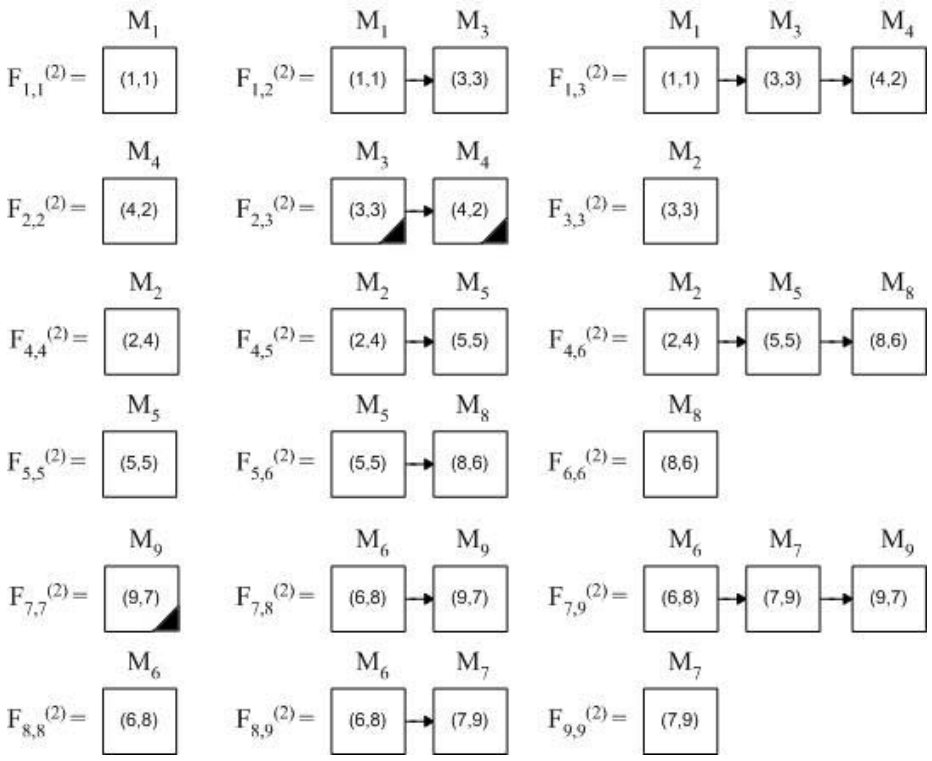

Figure 5: Level 2 of example 2-level 2-range.

Consider the range query $[3, 25], [4, 65]$. When we normalize this query it becomes $[2, 8], [2, 7]$. This is normalized by converting the L_i's to the rank of the first value greater than or equal to it in F, and the H_i's to the largest rank that is less than or equal to it. To answer this query, we must search at most 3 coverings of the y-range. For this particular query, we find that $F^{(2)}_{4,6}$ corresponds to a complete covering in level 1. This leaves at most two coverings in level 2 to search. In this case, we must search $F^{(2)}_{2,3}$ and $F^{(2)}_{7,7}$. The $F^{(2)}$ ranges searched for this example are shaded in Figure 3. ♦

2.3 Analysis

For a 2-level 2-range, we search at most 3 1-ranges, and each search can be conducted in logarithmic time, so we have

$$Q(N, 2) = O(\log N + A).$$

Furthermore, the storage and preprocessing have now been reduced to

$$S(N, 2) = O(N^2), \text{ and}$$

$$P(N, 2) = O(N^2).$$

Now we must consider the case where $\ell > 2$ and $k > 2$. Firstly, consider an ℓ-level 2-range. In this case, on the first level we have one block containing $N^{1/\ell}$ units with a maximum of N points each. The second level has $N^{1/\ell}$ blocks each containing $N^{1/\ell}$

units with a maximum of $N^{1-1/\ell}$ points each. On level i we store $N^{(i-1)/\ell}$ blocks, each containing $C(N^{1/\ell}, 2) = O(N^{2/\ell})$ intervals representing at most $N^{1-(i-1)/\ell}$ points [4]. (Bentley and Maurer's paper generalizes the maximum number of points per unit as $N^{1-(i+1)/\ell}$, but this is in fact a typo, it should have been $N^{1-(i-1)/\ell}$ points.) To answer a query, we select an appropriate covering of the query's y-range for each level and then search the corresponding 1-ranges (see Figure 7 for pseudocode). Furthermore, the ℓ level structure gives us $S(N) = P(N) = O(N^{1+2/\ell})$, so as we increase levels, we decrease storage and preprocessing, but increase search time by having to search at most an additional 2ℓ ranges.

Finally, lets consider the ℓ-level k-range structure. An ℓ-level k-range is inductively constructed out of ℓ-level (k-1)-ranges. This is done recursively by building ℓ-level (k-1)-ranges over the k-th coordinate, then by building ℓ-level (k-2)-ranges over the (k-1)-th coordinate, and so on (see Figure 6 for pseudocode). Bentley and Maurer stated that by choosing ℓ as a function of k and ε, for any fixed k and $\varepsilon > 0$, the generalized ℓ-level k-range has the following complexities:

$$P(N) = S(N) = O(N^{1+\varepsilon}), \text{ and}$$

$$Q(N) = O(\log N + A).$$

in order to understand the relationship between ℓ, k, and N. Since Bentley and Maurer did not show the exact calculation for ε, and this factor is important to know in order to understand the relationship between ℓ, k, and N, we further analyze the storage complexity. A recursion relation for the storage complexity can be defined as follows:

$$S(N, k, \ell) = \sum_{i=1}^{\ell} N^{(i-1)/\ell} C(N^{1/\ell}, 2) S(N^{1-(i-1)/\ell}, k-1),$$

where we sum the total storage from $i = 1$ to $i = \ell$, multiplying the storage of the blocks by the storage for the units and recursing on the maximum number of points and $k - 1$. Solving the recursion reveals the following value for $S(N, k)$:

$$\begin{aligned}
&S(N, 1, \ell) = O(N) \\
&S(N, 2, \ell) = O(N^{1+2/\ell}) \\
&S(N, 3, \ell) = O(N^{2/\ell}) O(N^{1+2/\ell}) \\
&S(N, 4, \ell) = O(N^{2/\ell}) O(N^{2/\ell}) O(N^{1+2/\ell}) \\
&\vdots \\
&S(N, k, \ell) = O(N^{1+2(k-1)/\ell}).
\end{aligned}$$

The asymptotic analysis indicates that by choosing $\ell = 2(k-1)$, we should be able to obtain $O(N^2)$ storage. In section 4.2.5 we verify this result empirically.

2.4 Range Indexing

In Bentley and Maurer's complexity analysis, they assume that the implementation of the multi-level k-range has $O(1)$ lookup time for each range structure, and that each level has at a fixed size with respect to the number of blocks, units, and points. Although they did not discuss in their paper how to do this, it can be done by indexing

the constructed ranges in a canonical form.

Consider the 2-level 2-range with $N = 9$. For each unit, we assign to ranges starting from the left most unit boundary to be indexed with the lowest possible values. So, for level 1 of our 2-level 2-range, the ranges $F^{(2)}_{1,3}$, $F^{(2)}_{1,6}$, and $F^{(2)}_{1,9}$ would be indexed with 0, 1, and 2 respectively. For the next unit, we perform the same operation, but account for the previously constructed ranges, therefore, the ranges $F^{(2)}_{3,6}$ and $F^{(2)}_{3,9}$ have indices of 0 and 1, but since there has been 3 previously constructed ranges, their indices are offset to be 3 and 4. Finally, $F^{(2)}_{6,3}$ would be indexed as 5. Indexing the ranges for level 2 is slightly more complicated because we now have to consider multiple blocks. Since each block has the same number of ranges (ie. $C(N^{1/\ell}, 2)$), we simply need to calculate the current block that we are in. This can be done by taking $\lfloor i/BW \rfloor$, where i corresponds to left range boundary of $F^{(2)}_{i,j}$ and BW is the block width. Then, to calculate the index, we add the number of ranges per block multiplied by the current block number plus the index calculated based on the unit. All of these calculations can be done in constant time (see Figure 8 for pseudocode).

3 Algorithms

In this section we introduce the algorithms used to construct, search, and perform the constant time range lookups for the multi-level k-range data structure. The algorithms accuracy was validated by comparing the search results to the results obtained through naive searching.

Assume that the construction and search algorithms are part of a *KRange* object, and that this object has local values for k and ℓ. Also, this object has a local array called *level*, where $level[i]$ stores the k-ranges for the ith level. Finally, assume that ranges are indexed starting at zero, and a range $F^{(t)}_{i,j}$ holds values with t-th coordinate ranks from i to $j - 1$ inclusive.

3.1 Construction Algorithm

```
ConstructRange(I: array of point data, ISize: size of I, L: left value, R: right value)
 1  if k = 1
 2    then ConstructOneRange(I, ISize, L, R)
 3         return
 4
 5  /* get the length of the largest range */
 6  maxrank ← I[ISize].Ranks[k], levelRootOfN ← ⌈maxrank^{1/l}⌉, M ← levelRootOfN^l
 7
 8  /* initialize range values */
 9  numOfBlocks ← 1, blockWidth ← M
10  expansion ← ⌈N^{1/l}⌉, unitWidth ← ⌈blockWidth/levelRootOfN⌉
11  for i ← 0 to ℓ − 1 /* construct each level */
12  do
13      startOfBlock ← 0, endOfBlock ← blockWidth
14      for cb ← 0 to numOfBlocks − 1 /* construct each block */
15      do
16          for cu ← startOfBlock to endOfBlock − 1, cu += unitWidth /* construct each unit */
17          do
18              rside ← cu + unitWidth − 1
19              rangeData ← data from I that falls between cu and right
20              if rangeData is not empty
21                then level[i].F^(k)_{cu,rside+1} ← new KRange object with k-1 dimensions and ℓ levels
22                     level[i].F^(k)_{cu,rside+1}.ConstructRange(rangeData, cu, right)
23
24              /* successively build units */
25              for rs ← rside + unitWidth to rs < endOfBlock, rs += unitWidth
26              do rangeData ← data from I that falls between cu and rs
27                  if rangeData is not empty
28                    then level[i].F^(k)_{cu,rs} ← new KRange object with k-1 dimensions and ℓ levels
29                         level[i].F^(k)_{cu,rs}.ConstructRange(rangeData, cu, rs)
30
31
32          startOfBlock ← endOfBlock, endOfBlock += blockWidth
33
34      numOfBlocks *= levelRootOfN
35      blockwidth /= levelRootOfN, unitWidth /= levelRootOfN
```

Figure 6: Construction algorithm for a multi-level k-range.

3.2 Search Algorithm

```
SearchRange(L: left point, R: right point, k)
 1  yi ← L.Ranks[k], yj ← R.Ranks[k]
 2  if k = 1
 3    then SearchOneRange(yi, yj), return
 4  numOfBlocks ← 1, blockWidth ← M
 5  unitWidth ← ⌈blockWidth/levelRootOfN⌉
 6  for i ← 0 to ℓ − 1
 7  do /* Calculate the complete covering for level i*/
 8     l ← ⌈yi/unitWidth⌉ ∗ unitWidth
 9     r ← ⌊(yj + 1)/unitWidth⌋ ∗ unitWidth
10     if blocks ← 1 OR rprev < lprev
11       then if l < r
12              then F_{l,r}^{(k)}.SearchRange(L, R, k-1)
13       else  if l < lprev
14              then F_{l,lprev}^{(k)}.SearchRange(L, R, k-1)
15            if rprev < r
16              then F_{rprev,r}^{(k)}.SearchRange(L, R, k-1)
17     lprev ← l, rprev ← r
18     numOfBlocks *= levelRootOfN
19     blockwidth /= levelRootOfN
20     unitWidth /= levelRootOfN
```

Figure 7: Search algorithm for a multi-level k-range.

3.3 Indexing Algorithm

The following algorithm calculates the array index, in constant time, that corresponds to the given left and right boundary values based on the block width and unit width.

```
GetRangeIndex(I: left boundary, J: right boundary, BW: block width, UW: unit width)
 1  /* set the current block value */
 2  CB ← ⌊i/BW⌋
 3
 4  /* set t to be the i value based on the current block, that is,
 5  if i is the left value of the first unit in any block, t would be 0 */
 6  t ← levelRootOfN − (i mod BW)/UW
 7
 8  /* set the index; first offset it based on the current block */
 9  index ← CB ∗ levelRootOfN ∗ (levelRootOfN + 1)/2
10
11  /* now offset based on the current unit; this calculates the number of units
12  per block and subtracts off the number of units we've seen so far (ie. how many indices have
13  been assigned), then it adds the order in which j gets assigned. */
14  index ← index + (levelRootOfN ∗ (levelRootOfN + 1) − t ∗ (t + 1))/2 + (j − i)/UW
15
16  return index − 1
```

Figure 8: Range indexing algorithm

4 Experimental Results

We randomly generate N points which satisfy the uniform distribution and store them as multi-level k-ranges. Then we randomly generate range queries of varying size and search the k-ranges with these queries. The following sections contain the algorithms used for random number generation as well as the main test algorithm.

4.1 Test Algorithms

The main test algorithm is shown below in Figure 9.

```
MainTest()
1  for k ← 2 to MAX_K
2  do
3      for i ← 0 to |NPOINTS|
4      do  /* create NPOINTS[i] points and store them in an l-level k-range */
5          call generatePoints
6          for q ← 0 to NQUERY
7          do call generateQuery to generate and search a query
```

Figure 9: Main test process.

4.1.1 Random Number Generation

The random number generator is based on Knuth's algorithm in Section 3.6 of [8]. It implements the best linear congruential random number generator proposed by D. H. Lehmer. The algorithm can be seen in Fig. 10. It returns a random number between min and max. The routine *RAN_START* and *RAN_ARR_NEXT* are from Knuth's algorithm [7], where *RAN_START* sets the seed for the random number generator based on the current time and *RAN_ARR_NEXT* contains the continuous sequence of random numbers.

```
GENERATERAND(min, max)
1  if first time call
2    then SRANDOM(time(NIL)) /* set a random seed based on current time */
3         RAN_START(random()) /* call Knuth's function to start */
4
5  /* get the next random number */
6  randomNumber ← RAN_ARR_NEXT() mod (max − min + 1)
7  return randomNumber + min
```

Figure 10: Pseudocode of random number generator.

4.1.2 Random point generation

The algorithm shown below in Fig. 11, generates N uniform randomly distributed points, then computes the rankings for each dimension, and finally inserts the points into the multi-level k-range.

```
GENERATEPOINTS(k, N)
 1  for i ← 0 to N
 2  do
 3      /* create a k dimensional point */
 4      Point p(k)
 5      for ki ← 0 to k
 6      do /* store random value in ki^th dimension */
 7          p.coordinate[ki] = generateRand(MIN, MAX)
 8          /* add coordinate value to an array for ranking input */
 9          rankArray[ki][i] ← p.coordinate[ki]
10
11      /* add point p to the list of points */
12      pointArray.Add(p)
13
14
15  /* call function to rank the coordinate values */
16  rankInput(pointArray, rankArray)
17
18  /* create KRange object, and construct the range */
19  KRange krange(k, l)
20  krange.constructRange(pointArray)
```

Figure 11: Pseudocode to generate N k-d uniformly distributed random points.

4.1.3 Random orthogonal query generation

We generate $NQUERY = 300$ random k-d queries for each set of N points. For each query, we calculate the time taken to find the points in range, and then store that value if the percentage falls within the following query window sizes: $(0, \log N^{1/2})$, $(\log N^{1/2}, \log N)$, and $(\log N, \log N^2)$. After the queries are run, we know the total time it took to find points within each boundary and how many times we found that many points; this allows us to calculate the average time to find points in range for each query window size. Figure 12 below shows the algorithm used to perform this operation. The function *getRank* retrieves the rank of the kth coordinate value in $O(\log N)$ time. The function $QWindowSize$ determines the index in the array $queryInfo$ in which to store the time based on how many points were found in range.

```
GENERATEQUERY(k)
 1  /* create points that represent the query */
 2  Point pLeft, pRight
 3  for ki ← 0 to k
 4  do
 5      pLeft.coordinate[ki] ← generateRand(MIN, MAX)
 6      pRight.coordinate[ki] ← generateRand(MIN, MAX)
 7
 8      if pRight.coordinate[ki] < pLeft.coordinate[ki]
 9        then swap(pRight.coordinate[ki], pLeft.coordinate[ki])
10
11      pLeft.rank[ki] ← getRank(pLeft.coordinate[ki])
12      pRight.rank[ki] ← getRank(pRight.coordinate[ki])
13
14  /* create a new Timer object */
15  Timer t
16  /* search the range and report the points to the array inRange */
17  krange.search(pLeft, pRight, inRange)
18
19  /* get the query window index and store information about the time to search */
20  queryInfo[QWindowSize(|inRange|)].Add(t.getTime())
```

Figure 12: Pseudocode for generating random search query.

4.2 Experimental Results

Experiments were run on a Sun Microsystem 880 with four 1.2 GHz UltraSPARC III processors, 16 GB's of main memory, running Solaris 8. Times were obtained using the timeval struct, which reports seconds and microseconds.

4.2.1 Range Search Test

Table 1 lists our results from $k = 2$ up to and including $k = 8$ for searching using a naive approach (linear bruteforce search), multi-level k-ranges, and a R*tree [2, 6]. The first column for each query window size represents the average search time, T_{NAIVE}, for the naive approach. The next two columns are T_{Kr}/T_{NAIVE} and T_{R*}/T_{NAIVE} for T_{Kr} equal to the average search time for the multi-level k-range and T_{R*} equal to the average search time for the R*tree. The average for each query window size, [0, $\log N^{1/2}$), [$\log N^{1/2}$, $\log N$), and [$\log N$, $\log N^2$), is calculated over a minimum of 30 queries and a maximum of $NQUERY = 300$. That is, we process 300 queries for each row of Table 1, and for each average search time, atleast 30 of those 300 queries fall within the window size.

The data points and queries were originally generated using the Test Algorithms listed in section 4.1. The points for each k and N were saved to a file along with the queries, and then these files were used by the naive and R*tree search. The query window sizes

taken into account in the experiment were kept small, in order to avoid the search time being dominated by reporting. For the multi-level k-range, we used $\ell = \lceil \log_2 N \rceil$ in order to minimize $S(N, k)$ (see Lemma 4.2).

Table 1: Data structure comparison for range searching in milliseconds

	N	$[0, \log N^{1/2})$			$[\log N^{1/2}, \log N)$			$[\log N, \log N^2)$		
k = 2	100	0.005	0.83	0.48	0.009	1.09	0.47	0.023	1.07	0.49
	1000	0.072	0.59	0.24	0.130	0.82	0.32	0.420	0.98	0.32
	10000	0.776	0.35	0.20	1.606	0.75	0.29	5.302	1.03	0.29
	100000	15.400	f	0.24	29.277	f	0.35	75.152	f	0.39
k = 3	100	0.005	2.83	0.54	0.011	2.23	0.52	0.024	1.77	0.45
	1000	0.059	1.32	0.42	0.137	1.27	0.48	0.335	1.36	0.41
	10000	0.745	f	0.31	1.608	f	0.42	4.047	f	0.39
	100000	17.035	f	0.35	31.752	f	0.51	65.657	f	0.54
k = 4	100	0.005	6.84	0.52	0.011	6.69	0.63	0.022	4.61	0.43
	1000	0.064	5497.56	0.36	0.158	4039.68	0.43	0.341	1572.38	0.43
	10000	0.796	f	0.35	1.875	f	0.48	4.507	f	0.59
	100000	17.819	f	0.38	38.166	f	0.70	84.830	f	0.57
k = 5	100	0.005	13.43	0.54	0.014	10.83	0.71	0.019	10.80	0.60
	1000	0.056	f	0.46	0.187	f	0.51	0.308	f	0.56
	10000	0.715	f	0.42	2.077	f	0.53	5.340	f	0.42
	100000	17.475	f	0.51	37.654	f	0.90	131.848	f	0.62
k = 6	100	0.005	f	0.62	0.015	f	0.71	0.018	f	0.45
	1000	0.048	f	<0.01	*	*	*	*	*	*
	10000	0.604	f	<0.01	*	*	*	*	*	*
	100000	21.350	f	<0.01	*	*	*	*	*	*
k = 7	100	0.005	f	0.65	*	*	*	0.023	f	0.57
	1000	0.04	f	0.01	*	*	*	*	*	*
	10000	0.586	f	<0.01	*	*	*	*	*	*
	100000	35.039	f	<0.01	*	*	*	*	*	*
k = 8	100	0.005	f	0.79	*	*	*	*	*	*
	1000	0.036	f	0.01	*	*	*	*	*	*
	10000	0.587	f	<0.01	*	*	*	*	*	*
	100000	36.721	f	0.03	*	*	*	*	*	*

* - no queries in this range f - failed during construction

As we can see from the above table, the multi-level k-range could not be constructed for most of the data points. This is because of the memory requirements of the data structure (see Theorem 4.3). We can also see that the R*tree outperformed the multi-level k-range in every test and even the naive search was faster for 17 of the 24 test cases that the multi-level k-range participated in. Section 4.2.2 and 4.2.3 explains why this poor performance occurs. We discovered that naive search always outperforms ℓ-level k-ranges for relatively low values of k.

4.2.2 Search Time Analysis

We consider ℓ and k as factors in the query analysis and arrive at the following theorem.

Theorem 4.1. $Q(N,k,\ell) = O((2\ell)^{(k-1)}(\log N + A))$.

Proof:
For each k we recursively search a maximum of 2ℓ ℓ-level (k-1)-ranges until $k = 1$; when $k = 1$, the ℓ-level 1-range is searched in $O(\log N + A)$ time. The following recursion can be defined:

$$Q(N,k,\ell) \leq 2\ell Q(N,k-1,\ell).$$

We solve the recursion as follows:

$$\begin{aligned}
&Q(N,k,\ell) \leq 2\ell Q(N,k-1,\ell)\\
&Q(N,k-1,\ell) \leq 2\ell Q(N,k-2,\ell)\\
&Q(N,k-2,\ell) \leq 2\ell (Q(N,k-3,\ell)\\
&\vdots\\
&Q(N,2,\ell) = O(\log N + A).
\end{aligned}$$

This gives $Q(N,k,\ell) \leq (2\ell)^{(k-1)}O(\log N + A)$, and $Q(N,k,\ell) = O((2\ell)^{(k-1)}(\log N + A))$.◀

By considering ℓ and k as factors we see that the search time is significantly worse than the previously reported search time of $Q(N) = O(\log N + A)$ for fixed k and $\varepsilon > 0$. With $Q(N,k,\ell) = O((2\ell)^{(k-1)}(\log N + A))$, the multi-level k-range search becomes worse than naive search for most values of k.

4.2.3 Level Analysis

Bentley and Maurer proved that increasing ℓ decreases the storage and preprocessing time for the multi-level k-range by a factor of $2/\ell$. This can be seen by considering $S(N,k) = O(N^{1+\varepsilon})$, where $\varepsilon = 2(k-1)/\ell$ as was calculated in section 2.3. As ℓ grows, ε gets smaller, and decreases storage.

In their analysis they assumed that N is always equal to a^ℓ where a and ℓ are positive integers. In general, this will rarely be the case. In order to construct the multi-level k-range when $N \neq a^\ell$, the value of N that is used is $M = \lceil N^{1/\ell} \rceil^\ell$, rather than N. If $N \neq a^\ell$, i.e. $N < a^\ell$, then M could be very different than N. Therefore, we must answer two important questions: what value of ℓ will minimize storage based on N? and does the storage complexity change if $N \neq a^\ell$?

Lemma 4.2. $S(N,k,\ell)$ is minimal when $\ell = \lceil \log_2 N \rceil$.

Proof:
When $N \neq a^\ell$, where a and ℓ are positive integers, the value of $M = \lceil N^{1/\ell} \rceil^\ell$ is used by the ℓ-level k-range in order construct the block and unit boundaries.

For $\ell \geq \log_2 N$, and $\ell < \infty$, $\lceil N^{1/\ell} \rceil = 2$. This can be rewritten as $N^{1/\ell} \leq 2$, which reduces to

$$\frac{1}{\ell} \log N \leq \log 2,$$

by taking the logarithm of both sides. We see that increasing ℓ past $\log_2 N$ will not decrease storage because $\lceil N^{1/\ell} \rceil^\ell$ becomes equivalent to 2^ℓ and $S(N, k, \ell)$ becomes $O(2^{\ell^{1+2(k-1)/\ell}})$. Therefore, we can only reduce storage by increasing ℓ up to $\lceil \log_2 N \rceil$. ◀

Theorem 4.3. $S(N, k) = \Theta(N^{1+2(k-1)/\log_2 N})$ for fixed $\ell = \lceil \log_2 N \rceil$, where $N \neq a^\ell$ (i.e. $N < a^\ell$), and a and ℓ are positive integers.

Proof:
From Lemma 4.2 we know that $S(N, k, \ell)$ is minimal when $\ell = \lceil \log_2 N \rceil$. We can substitute this value of ℓ into $S(N, k, \ell)$. This gives

$$S(N, k, \ell) = O(N^{1+2(k-1)/\log_2 N})$$

and since ℓ is no longer a factor, we can write the storage as

$$S(N, k) = O(N^{1+2(k-1)/\log_2 N}).$$

We know this is the minimum possible storage from Lemma 4.2, therefore, $S(N, k) = \Theta(N^{1+2(k-1)/\log_2 N})$.◀

Bentley and Maurer only considered the ideal case, ie. $N = a^\ell$, when they proved that $S(N) = O(N^{1+\varepsilon})$. The ideal case will rarely occur; in general the minimal storage is exponential in k. This result helps explain why our multi-level k-range implementation could not be constructed successfully for most tests in section 4.2.1.

4.2.4 Level Experiment

Table 2 lists our results for constructing multi-level k-ranges with gradually increasing ℓ values over two 3-dimensional data sets of $N = 100$ and $N = 1000$. The *Predicted* column lists the calculation of $S(N, k, \ell)$ for the given value of N, k, and ℓ based on $S(N, k, \ell) = O(N^{1+2(k-1)/\ell})$. The *Implementation* column lists the total allocated pointers by our algorithm when constructing the multi-level k-range for the given parameter values. The two bold rows correspond to when $\ell = \lceil \log_2 N \rceil$ for $N = 100$ and $N = 1000$. We could not verify the storage for large N because of the ℓ-level k-range memory requirements. As we can see, the value of both the predicted storage and our actual storage is minimal for this value of ℓ. Furthermore, once $\ell > \lceil \log_2 N \rceil$, the storage continually increases. This is as predicted by our analysis in the previous section.

Table 2: Storage allocation as ℓ increases

	ℓ	Predicted	Implementation
N = 100, k = 3	2	1000000	486536
	4	65536	221899
	6	59049	482341
	7	**2047**	**135043**
	8	4096	258925
	10	16384	988321
	12	65535	3902805
N = 1000, k = 3	2	1073741824	2618500012
	4	1679616	19992814
	6	1048575	29274490
	8	531441	54174922
	10	**16384**	**9610069**
	12	65535	37492425
	14	262143	148673341

Figure 13 displays a graph representing Table 2. The y-axis is the logarithmic scale for the storage capacity for each test case. The x-axis is the number of levels used by the multi-level k-range.

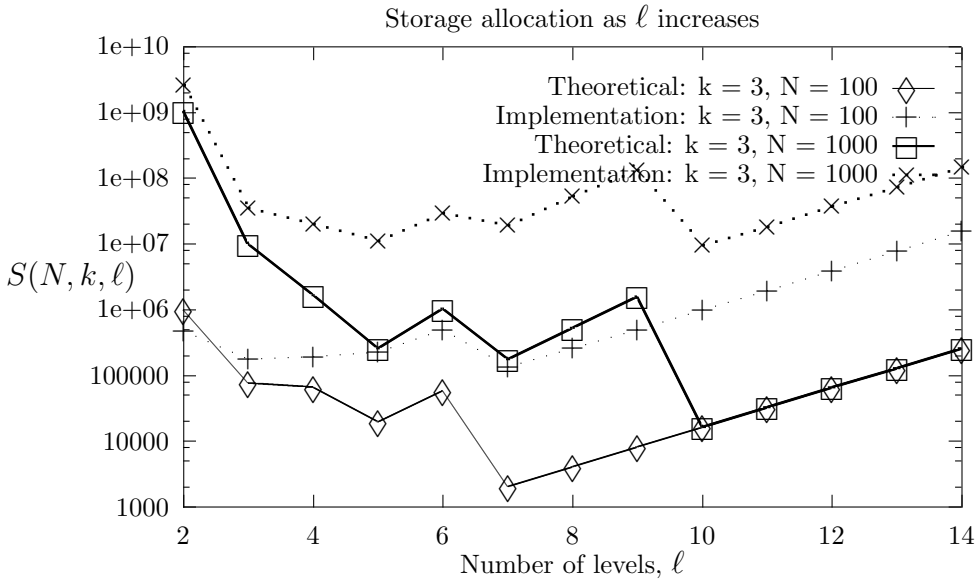

Figure 13: Experimental and theoretical storage allocation.

Figure 14 displays a graph representing the bytes allocated by the ℓ-level k-range while constructing data sets $N = 100$ and $N = 1000$ with $k = 3$. ℓ is varied from 2 to 14 (the x-axis), and the y-axis represents the bytes allocated.

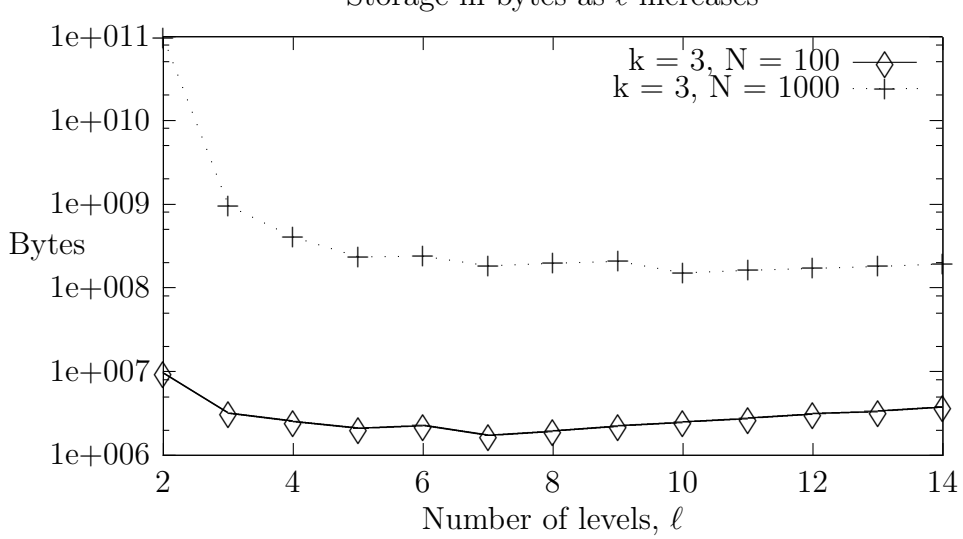

Figure 14: Multi-level k-range storage allocation in bytes.

4.2.5 Storage Complexity Test

Table 3 lists for different values of N, the total number of pointers allocated by our implementation of the multi-level k-range data structure during the process of construction. The Ratio column represents the value of p_i/p_{i-1}, where p_{i-1} and p_i are the number of pointers indicated in row $i-1$ and i respectively.

Table 3: Multi-level k-range storage complexity results

	N	Pointers	Ratio
k = 2,	100	5005	
$\ell = 2$	200	20640	4.1239
	400	66010	3.1982
	800	266062	4.0306
	1600	952020	3.5782
	3200	3768482	3.9584
			Avg. = 3.7779
k = 3,	100	210453	
$\ell = 4$	200	572364	2.7120
	400	3045166	5.3203
	800	14247151	4.6786
	1600	63319429	4.4444
	3200	279432608	4.4106
			Avg. = 4.3132

In section 2.3 we calculated the storage complexity as $S(N,k,\ell) = O(N^{1+2(k-1)/\ell})$. This result can be used to approximate $S(N,2,2)$ and $S(N,3,4)$ for the data points listed in the above table. Since, for each value of k, we chose $\ell = 2(k-1)$, we obtain $S(N,k,\ell) = O(N^2)$ for both $k = 2$ and $k = 3$. Therefore, since N grows by a factor of 2 in the above table, the storage grows by a factor of 4. This factor of 4 correlates with the empirical results shown in Table 3.

We can also see from Table 3 why the multi-level k-range cannot be constructed for large values of N and k. The total number of allocated pointers becomes very large as N and k grow. Increasing ℓ will decrease storage and preprocessing, but due to the nature of the overlapping ranges, there will be a lot of duplicate pointer references. Also, since the multi-level k-range works with $\lceil N^{1/\ell} \rceil^\ell$ when $N \neq a^\ell$, a lot of empty units may be created. This adds additional storage without any gain in information or speed.

5 Conclusions

We've fully implemented and tested the multi-level k-range data structure. To our knowledge, this data structure has never been implemented. We developed and further analyzed Bentley and Maurer's storage complexity result by finding the actual value of ε. This allowed us to empirically verify the storage complexity and demonstrate why this data structure is not practical for range searching.

We compared multi-level k-ranges to naive and R*tree searching. The testing demonstrated that the multi-level k-range cannot be constructed for even mid-range values of N and k. This is due to the design of the data structure where ℓ must be very large in order to reduce the storage complexity to a reasonable size. However, we proved that there is a limit to how large we can make ℓ and still have it reduce the storage, therefore, we cannot reduce storage enough to make practical use of this data structure. Results also demonstrated that the R*tree and naive search out perform the multi-level k-range search. We showed that when ℓ and k are considered as factors, $Q(N,k,\ell) = O((2\ell)^{(k-1)}(\log N + A))$. This explained the poor performance of the multi-level k-range.

We proved that $S(N,k) = \Theta(N^{1+2(k-1)/\log_2 N})$ for $N \neq a^\ell$, where a and ℓ are positive integers. This demonstrated why the data structure had problems with storage during our empirical tests. Due to these new complexity results, the multi-level k-range could never be competitive for searching or be a "practical" data structure for an application.

References

[1] P. K. Agarwar, "Range Searching", *Handbook of discrete and computational geometry*, CRC Press Inc., Boca Raton, FL, 1997, pp. 575-581.

[2] N. Beckmann, H. P. Kriegal, R. Schneider, and B. Seeger, The R*Tree: An Efficient and Robust Access Method for Points and Rectangles, *Proc. ACM SIGMOD Intl. Symp. on the Management of Data*, 1990, pp. 322-331.

[3] J. L. Bentley, J. H. Friedman, "Data Structures for Range Searching", *Computing Surveys*, Vol. 11, No. 4, 1979.

[4] J. L. Bentley, H. A. Maurer, "Efficient Worst-Case Data Structures for Range Searching", *Acta Informatica*, Vol. 13, No. 2, 1980.

[5] B. Chazelle, "Filtering Search: A New Approach To Query-Answering", *SIAM J. COMPUT.*, Vol. 15, No. 3, August 1986.

[6] K. K. Chu, Database Research Group: R*tree sourcecode, http://www.cse.cuhk.edu.hk/ kdd/program.html, Last Visit: April, 2004.

[7] D. E. Knuth, *The Art of Computer Programming*, Vol. 2, *Seminumerical Algorithms*, 2^{nd} edition, Addison Wesley, Mass., USA, 1973.

[8] D. E. Knuth, *The Art of Computer Programming*, Vol. 3, *Sorting and Searching*, 2^{nd} edition, Addison Wesley, Mass., USA, 1973.

[9] F. F. Yao, "Computational Geometry", *Handbook of Theoretical Computer Science: Volume A: Algorithms and Complexity*, Elsevier, Amsterdam, 1990, pp. 368-374.