An Optimal Divide-Conquer Algorithm for 2D Skyline Queries

Hai-Xin Lu, Yi Luo, and Xuemin Lin

School of Computer Science & Engineering
University of New South Wales
Sydney, NSW 2052, Australia
{cshlu, luoyi, lxue}@cse.unsw.edu.au
http://www.cse.unsw.edu.au

Abstract. Skyline query processing is fundamental to many applications including multi-criteria decision making. In this paper, we will present an optimal algorithm for computing skyline in the two dimensional space. The algorithm has the progressive nature and adopts the divide-conquer paradigm. It can be shown that our algorithm achieves the minimum I/O costs, and is more efficient and scalable than the existing techniques. The experiment results demonstrated that our algorithm greatly improves the performance of the existing techniques.

1 Introduction

Given a set of points, "skyline query" returns the points in the set, which are not "dominated" by another point (see Fig. 1 for example). Skyline query processing is fundamental to many applications, including multi-criteria decision making [12] where the optimal criteria are sometimes conflicting. For instance, consider the query "finding a cheap car with a recent model". This is a typical example involving the two conflicting goals, cheap price and recent model, since cars recently made are likely expensive. Instead of identifying an optimal answer to the query, it may only be possible to provide a set of candidate answers that likely meet the user requirements. Clearly, some users may be able to afford a car with the latest model, while others may only be able to choose cars as new as possible within their budgets. In this application, it is unlikely that one will consider cars both older and more expensive than another; thus, the set of candidate cars form skyline (see Fig. 1 for example).

Efficient main-memory algorithms for computing skyline were first investigated in [6]. To manage large datasets which may not necessarily fit in main-memory, two skyline algorithms were developed in [1]. The algorithm *Divide-Conquer* [1] proposed to divide a data space into partitions so that main memory algorithms can be applied to each partition, while *Block-Nested-Loop* proposed to maintain a candidate list for skyline. Both Divide-Conquer and Block-Nested-Loop require multiple reads of a dataset. Further, they are not suitable for online processing of skyline queries since the first skyline point cannot be generated until the entire dataset is scanned. To resolve these problems, two new algorithms *Bitmap* and *Index* were developed in [13].

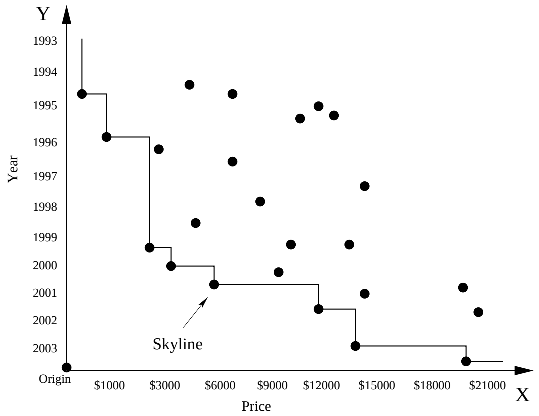


Fig. 1. An example of Skyline of cars

Note that the above algorithms take the same assumption - there is no spatial index. The algorithm (NN) [7] is the first algorithm for computing skyline based on R-trees [2, 4], which uses the nearest neighbour search techniques [5, 11]. Observe [10] the heavy I/O overheads in NN especially in a high dimensional space. The algorithm (BBS) [10] was developed to minimise the I/O costs, which extended the best-first search technique [5] for the nearest neighbour problem. As reported in [10], BBS guarantees the minimum I/O overheads but requires a larger memory space than that in NN in the two dimensional space. Note that less the memory space requires, more scalable the algorithm is.

In this paper, we will present a novel algorithm for computing skyline in the two dimensional space. The algorithm guarantees the minimum I/O costs, and requires a smaller memory space than the existing techniques (thus is more scalable than the existing techniques). Further, our new algorithm does not require a global validation of those progressively generated local skyline points. Our experiment results demonstrated that the new algorithm not only shares the different advantages from the existing techniques but also is much more efficient than the existing techniques.

The rest of the paper is organised as follows. Next section is dedicated to some background knowledge and related work. In section 3, we present our new skyline algorithm and its performance analysis. Section 4 reports the experimental results. This will be followed by conclusion and remarks.

2 Preliminary

This section provides a brief overview of NN and BBS. To the best of our knowledge, NN and BBS are the only R-tree based algorithms for computing skyline.

Given a set of points S in the 2 dimensional space, a point p in S is a *skyline point* of S if p is not *dominated* by another point in S. A point (x', y') *dominates* another point (x, y) if $x' \leq x$ and $y' \leq y$. The *skyline* of S is the set of skyline points of S.

An *entry* in R-tree refers to a node's MBR or a data point. As depicted in Fig. 3, each e_i (for $0 \leq i \leq 11$) is an entry, as well as the data points from a to t. An entry is dominated by a point p if the *lower-left corner* of the entry is dominated by p.

Both NN and BBS apply R-trees based nearest neighbour search techniques [5, 11]. Note that a nearest neighbour search returns the points closest to a given query point. NN and BBS need to deal with the *distance* from a rectangle (entry) to a point, that is, the minimum distance between p and a point on the rectangle. In this paper, we denote the distance from an entry to the query point by $minDist$.

2.1 Nearest Neighbours (NN)

Kossmann et al. [7] proposed a divide-conquer algorithm NN based on multiple calls of the nearest neighbour search algorithms [5, 11]. The algorithm iteratively generates a nearest neighbour point to the origin in a given region, and is applicable to any *monotonic* distance function (for example, Euclidean distance). The basic idea is to enforce the nearest neighbour points generated iteratively to be part of skyline.

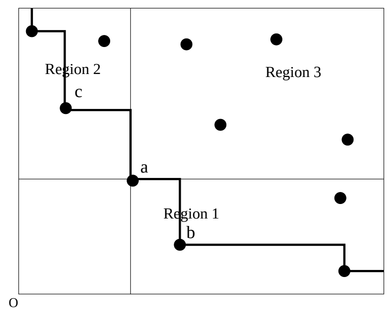


Fig. 2. Nearest Neighbours algorithm

As depicted in Fig. 2, the regions 1 and 2 are added to a *to-do* list in the algorithm for further space partitioning after a is output as the nearest neighbour point. Note that the points in region 3 are dominated by a, and thus can be discarded by the algorithm. The algorithm terminates when the to-do list becomes empty. To reduce the storage space in the to-do list, each region is simply recorded by its upper-right corner since the lower-left region resulted in a space devision is always empty.

Fig. 4 illustrates, step by step, the to-do lists of the algorithm NN based on the example in Fig. 3. In this example, b, k, h, and i form the skyline.

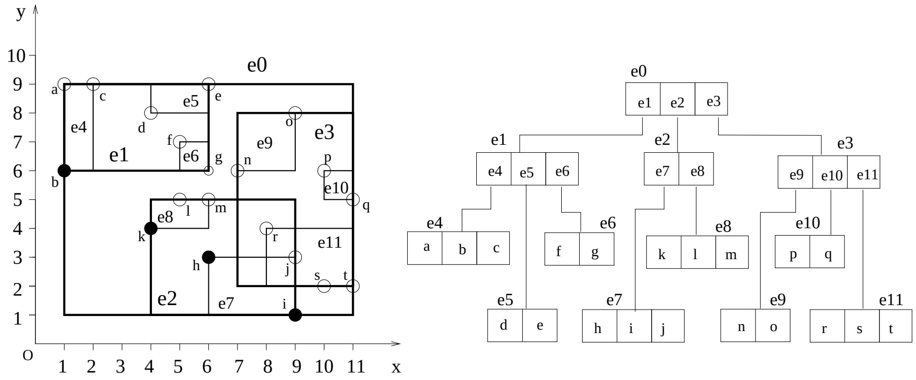


Fig. 3. R-tree example

Recall that in the to-do list every element corresponds to the coordinates of upper-right corner of a region. Note that in NN, a region is removed from the to-do list only if it is processed.

Action	**To-do List Contents**	**Skyline Points**	**To-do List Size**
initial state	$(11, 9)$	$\emptyset$	1
NN query in $(11, 9)$	$(11, 6)$,$(1, 9)$	$\{b\}$	2
NN query in $(11, 6)$	$(11, 4)$,$(4, 6)$,$(1, 9)$	$\{b, k\}$	3
NN query in $(11, 4)$	$(11, 3)$,$(6, 4)$,$(4, 6)$,$(1, 9)$	$\{b, k, h\}$	4
NN query in $(11, 3)$	$(11, 1)$,$(9, 3)$,$(6, 4)$,$(4, 6)$,$(1, 9)$	$\{b, k, h, i\}$	5
NN query in $(11, 1)$	$\cancel{(11, 1)}$,$(9, 3)$,$(6, 4)$,$(4, 6)$,$(1, 9)$	$\{b, k, h, i\}$	5
NN query in $(9, 3)$	$\cancel{(9, 3)}$,$(6, 4)$,$(4, 6)$,$(1, 9)$	$\{b, k, h, i\}$	4
NN query in $(6, 4)$	$\cancel{(6, 4)}$,$(4, 6)$,$(1, 9)$	$\{b, k, h, i\}$	3
NN query in $(4, 6)$	$\cancel{(4, 6)}$,$(1, 9)$	$\{b, k, h, i\}$	2
NN query in $(1, 9)$	$\cancel{(1, 9)}$	$\{b, k, h, i\}$	1

Fig. 4. To-do List Contents

2.2 Branch and Bound Skyline Algorithm (BBS)

As pointed out in [10], multiple applications of a nearest neighbour search algorithm in NN may lead to multiple scans of entries in the *R*-tree. For instance, regarding the example in Fig. 3, to get the skyline points b and k both the root entry and the intermediate entry e_2 have to be visited (expanded). Further, in NN empty regions may be unnecessarily kept in the to-do list for long time.

Moreover, NN leads to many duplicated elements in the to-do list and causes many *false* hits in a high dimensional space.

To resolve the problems in NN, the algorithm BBS developed by Papadias et al. extended the best-first search technique [5] by keeping all the necessary entries in a "heap" until they are no longer useful. Elements (entries) in the heap are sorted by their $minDist$ values (to the origin), and then the skyline points are generated iteratively as the nearest neighbour points to the origin. For a dominance validation, the skyline points generated already are kept in a list T as BBS proceeds:

Algorithm BBS

Insert the root entry of R-tree into the heap H.
While $H \neq \emptyset$ **do**
{ get the first element e from H;
  $H := H - \{e\}$;
  If e is dominated by one point in T **then** discard e
  else if e is a data point **then** $T := T \cup \{e\}$
    else add to H the children entries of e
      which are not dominated by a point in T; }

Fig. 5 shows the heap contents for the example in Fig. 3 while applying BBS. The second field in an element is a value of minDist where $x + y$ is the distance function.

Action	Heap Contents	Skyline Points	Heap Size
initial state	$(e_0, 2)$	$\emptyset$	1
expand e_0	$(e_2, 5)$,$(e_1, 7)$,$(e_3, 9)$	$\emptyset$	3
expand e_2	$(e_7, 7)$,$(e_1, 7)$,$(e_8, 8)$,$(e_3, 9)$	$\emptyset$	4
expand e_7	$(e_1, 7)$,$(e_8, 8)$,$(h, 9)$,$(e_3, 9)$,$(i, 10)$,$(j, 12)$	$\emptyset$	6
expand e_1	$(e_4, 7)$,$(e_8, 8)$,$(h, 9)$,$(e_3, 9)$,$(i, 10)$, $(e_6, 11)$,$(e_5, 12)$,$(j, 12)$	$\emptyset$	8
expand e_4	$(b, 7)$,$(e_8, 8)$,$(h, 9)$,$(e_3, 9)$,$(a, 10)$, $(i, 10)$,$(c, 11)$,$(e_6, 11)$,$(e_5, 12)$,$(j, 12)$	{b}	10
expand e_8	$(k, 8)$,$(h, 9)$,$(e_3, 9)$,$(l, 10)$,$(a, 10)$, $(i, 10)$,$(m, 11)$,$(c, 11)$,$(e_6, 11)$,$(e_5, 12)$,$(j, 12)$	{b,k,h}	11
expand e_3	$(e_{11}, 10)$,$(l, 10)$,$(a, 10)$,$(i, 10)$,$(m, 11)$, $(c, 11)$,$(e_6, 11)$,$(e_5, 12)$,$(j, 12)$	{b,k,h}	9
expand e_{11}	$\cancel{(l, 10)}$,$\cancel{(a, 10)}$,$(i, 10)$,$\cancel{(m, 11)}$, $\cancel{(c, 11)}$, $\cancel{(e_6, 11)}$,$\cancel{(s, 12)}$,$\cancel{(e_5, 12)}$,$\cancel{(j, 12)}$,$\cancel{(t, 13)}$	{b,k,h,i}	10

Fig. 5. Heap Contents

It has been shown that BBS is optimal in terms of I/O; that is, only "necessary" entries in the R-tree are visited. We will formally state the I/O optimality in the next section.

3 A New Divide-Conquer Algorithm

In this section, we present a new skyline algorithm based on a divide and conquer paradigm. The algorithm is more efficient than BBS and NN. It has a low memory space requirement and guarantees the I/O optimality. We start with a description of our motivation.

3.1 Motivation

Note that BBS achieves the I/O optimality by reading the same entries only once. This means that once reading into the buffer, the information has to be kept there till it is no longer useful. By doing this, the maximal size of the heap will be unnecessarily large if entries are not pruned early nor the R-tree is drilled down quickly.

Regarding the example in Fig. 3, Fig. 5 showed that in BBS, entries e_5 and e_6 have to be kept in the heap for quite long time after e_2 is expanded. However, as depicted in Fig. 6 we can use e_2 (even before expanding e_2) to prune e_5 and e_6. This additional pruning step can be formally done by specifying a dominance relationship between two entries (not necessary data points). An entry e is *dominated* by another entry e' if e is outside e' and is dominated by the lower-left corner of e'. In this example, e_5 and e_6 are dominated by e_2, and thus should be discarded from a further consideration.

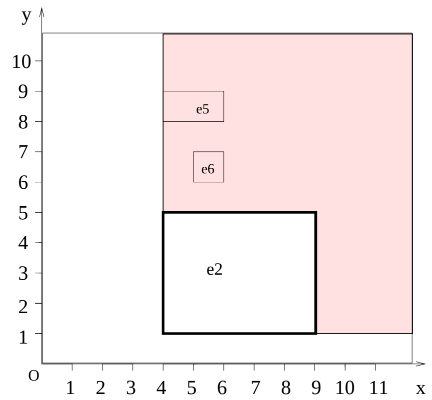


Fig. 6. Region Dominance

Fig. 5 also demonstrated that to generate b, BBS has to traverse two paths on the R-tree:

path 1: e_0, e_2, and e_7.
path 2: e_0, e_1, and e_4.

In this case, it will be ideal if only the path (e_0, e_1, e_4) is visited to generate b as this minimises the size of information concurrently held in the buffer. Our new algorithm can lead such an ideal situation in most cases.

Further, BBS maintains a list T of the current skyline points to validate, by a binary search on T, if a newly generated entry should be included for a further consideration; this expensive requirement is no longer needed in our algorithm. In fact, the new algorithm is I/O optimal (like BBS) and requires a much smaller buffer size than that in BBS. Moreover, our algorithm is much faster than BBS and NN.

3.2 DCSkyline

In this subsection, we present a novel divide-conquer skyline algorithm - DC-Skyline. The divide-conquer in DCSkyline proceeds in a different way from that in NN.

Where to divide

DCSkyline traverses the R-tree level by level like a depth-first search paradigm. After expanding an intermediate entry, it always chooses the entry in a region, with the minimum distance to the origin, to divide the region.

For instance, regarding the example in Fig. 7, e_2 is first chosen if the distance function $x + y$ is used. Then, DCSkyline uses the lower-left corner of e_2 to divide the whole space into 4 rectangular regions for further consideration, R_0, R_1, R_2, and R_3 (e_2), as depicted in Fig. 7. Note that entries outside these 4 regions are dominated by e_2 and thus should be discarded. Since the distance from e_2 to the origin is minimum, R_0 is always empty (i.e. no intersection with the entries in the R-tree). Therefore, in our algorithm we need only to consider the three regions R_1, R_2, and R_3. Below we show the properties in these three regions, which will lead to an effective divide-conquer process in DCSkyline.

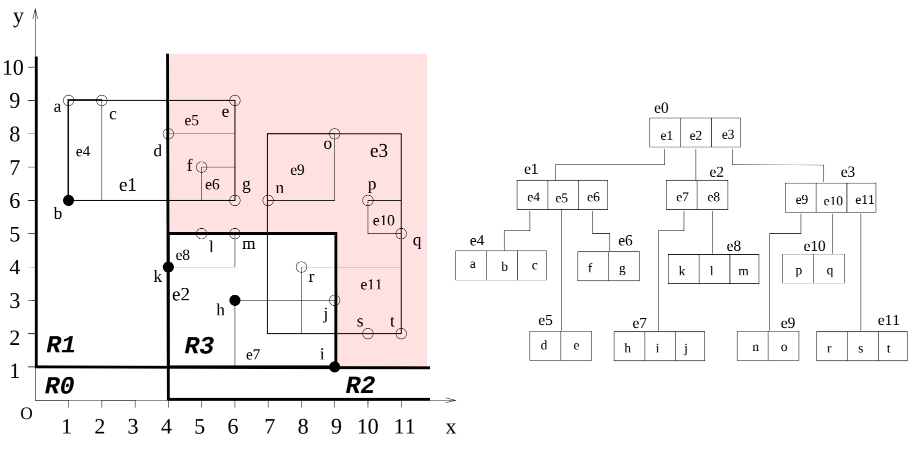


Fig. 7. Example of DCSkyline

Region Properties

Suppose that in a rectangular region R, the entry e has the minimum distance to the origin. On the dividing point (the lower-left corner of e), R is divided into the 4 rectangular regions for further consideration: R_0, R_1, R_2, and R_3, as depicted in Fig. 7.

Theorem 1. *The local skyline points in R_1 or R_2 are skyline points in R.*

Proof: Denote the dividing point by (x, y). Without loss of generality, we assume the lower-left corner of R is the origin, and $R = [0, \infty) \times [0, \infty)$. Then, $R_1 = [0, x) \times [y, \infty)$, $R_2 = [x, \infty) \times [0, y)$, and $R_3 = [x, x_h] \times [y, y_h]$ where (x_h, y_h) is the upper-right corner of e.

Without loss of generality, we prove only that the theorem holds for region R_1. Note that $x_1 < x$ for each point $(x_1, y_1) \in R_1$, $x_2 > x$ for each point $(x_2, y_2) \in R_2$, and $x_3 \geq x$ for each point $(x_3, y_3) \in R_3$.

Since $x_1 < x_2$, (x_1, y_1) is not dominated by (x_2, y_2). Similarly, (x_1, y_1) is not dominated by (x_3, y_3). Thus the theorem holds for R_1. $\Box$

Theorem 1 states that the local skyline points generated in R_1 and R_2, respectively, do not have to be verified globally. Clearly, the set of local skyline points generated in R_1 do not intersect the set of local skyline points in R_2.

Unfortunately, some local skyline points generated in the region R_3 may not be skyline points in R. To avoid a global check, we can use the right-most skyline point in R_1 and the left-most skyline point in R_2 to further restrict our search in the region R_3.

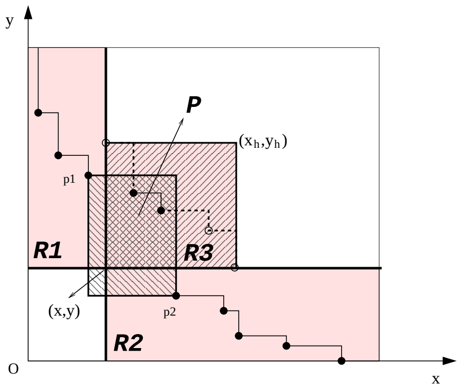


Fig. 8. The Principle of D&C

Let p_1 denote the right-most skyline point in R_1, p_2 denote the left-most skyline point in R_2, and P denote the rectangle spanning p_1 and p_2. Let $R_3' = R_3 \cap P$; see Fig. 8 for example. Note that $p_1 = (0, \infty)$ if there is no skyline point in R_1; and $p_2 = (\infty, 0)$ if there is no skyline point in R_2. The following theorem can be immediately verified.

Theorem 2. *The local skyline points in R_3' are skyline points in R. Further, the skyline of R consists of the local skyline points from R_1, R_2, and R_3'.*

Based on Theorems 1 and 2, our algorithm DCSkyline recursively applies a divide-conquer paradigm by the following ordering:

1. Find the skyline in R_1 by calling DCSkyline.
2. Find the skyline in R_2 by calling DCSkyline.
3. Use p_1 and p_2 to get R_3', and call DCSkyline on R_3' to get the skyline.

Region List in DCSkyline

In DCSkyline, we use the *region-list* to store the rectangular regions to be further divided; only regions that may contain skyline points will be pushed into the region list. In the example in Fig. 7, R_2 is empty, and thus is not in the list.

In the region-list, we also store the expanded entries associated with each region; that is, for each region, we store the entries already visited in the R-tree that intersect the region.

Note that an entry may appear in more than one region. Consider the example in Fig. 9. In the first step when the whole space is divided by the lower-left corner of e_2, e_1 is contained in both R_1 and R_3. To guarantee to read e_1 only once and expand e_1 at most once, e_1 is linked to R_1 and R_3. Further, after expanding e_1, the children entries of e_1 may also be involved in R_1 and R_3; for example e_3 in Fig. 9. Therefore, if e_1 is expanded when processing R_1, we also need to check whether or not its children should be linked to R_3.

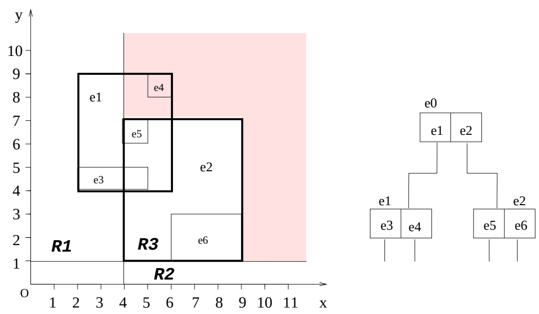


Fig. 9. Entries cross different regions

Note that R_1 and R_2 never intersect the same entry in the R-tree.

Expanding Entries

An entry is expanded if it is either the only entry intersecting R_1 (or R_2) or it is the entry corresponding to R_3. Note that in DCSkyline if the entry to be expanded is a data point, then it is output as a skyline point.

Pruning Regions and Entries

- If an entry e is not in any region, then the entry is discarded.
- If an entry is expanded, then the entry is discarded.
- If a region is empty, then the region is discarded.

DCSkyline Algorithm

The pseudo-code of the algorithm is shown in Fig. 10.

Initialization.
$E :=$ set of children entries of the root of R-tree;
$R :=$ the data space;
$RT := R$-tree;
$S := \emptyset$ // *skyline points*

Algorithm DCSkyline (E, R, S)
  if R is a type 3 Region **then**** // *expand an entry*
   {remove e (the entry defining the region R) from E
   as well as from the other intersecting regions;
   add the appropriate children entries of e to E and the other intersecting regions; }
  while $|E| = 1$ and the entry e in E is not a data point **do**** // *expand an entry*
   {remove e from E and the other intersecting regions;
   add the appropriate children entries of e to E and the other intersecting regions; }
  if $|E| = 1$ and the entry e in E is a data point **then** $S := \{e\}$
  else if $|E| = 0$ **then** $S := \emptyset$
    else // $|E| > 1$
    { choose the closest entry e from E;
     divide R into R_1, R_2 and R_3 using e (as depicted in Fig. 7);
     $E_1 := E \cap R_1$; $E_2 := E \cap R_2$; $E_3 := E \cap R_3$; // *entry lists*
     $S_1 := \emptyset$; $S_2 := \emptyset$; $S_3 := \emptyset$; // *skyline points*
     if $|E_1| > 0$ **then** DCSkyline (E_1 , R_1 , S_1);
     if $|E_2| > 0$ **then** DCSkyline (E_2 , R_2 , S_2);
     let p_1 the right-most skyline point from S_1; //*If* $S_1 = \emptyset$ *then* $p_1 = (0, \infty)$
     let p_2 the left-most skyline point from S_2; //*If* $S_2 = \emptyset$ *then* $p_2 = (\infty, 0)$
     let P the region spanning p_1 and p_2 (as depicted in Fig. 8);
     $R_3 := R_3 \cap P$; $E_3 := E_3 \cap P$;
     if $|E_3| > 0$ **then** DCSkyline (E_3 , R_3 , S_3);
     $S := S_1 \cup S_2 \cup S_3$ }
end

$**$ As depicted in Fig. 9, an entry may appear in more than one region. To guarantee to read each entry only once, extending an entry should be carried out in all the current regions intersecting the entry. In our algorithm, we linked an entry to the intersecting regions for efficiency.

Fig. 10. Pseudo Code

Fig. 11 provides a sample run of the algorithm DCSkyline based on the example in Fig. 7, where only the region-list is illustrated. Note that every element in the region-list has two fields representing the region and the set of corresponding entries.

Action	**Region list Contents**	**Skyline Points**	**Region + Entry List Size**
initial state	$[R, \{(e_0, 2)\}]$	$\emptyset$	1 + 1
expand e_0 & divide R	$[R_1, \{(e_1, 7)\}]$,$[R_3, \{(e_2, 5), (e_3, 9)\}]$	$\emptyset$	2 + 3
expand e_1	$[R_1, \{(e_4, 7)\}]$,$[R_3, \{(e_2, 5), (e_3, 9)\}]$	$\emptyset$	2 + 3
expand e_4	$\cancel{[R_1, \{(b, 7)\}]}$,$[R_3, \{(e_2, 5), (e_3, 9)\}]$	{b}	2 + 3
expand e_2 & divide R_3	$[R_{31}, \{(e_8, 8)\}]$,$[R_{33}, \{(e_7, 7), (e_3, 9)\}]$	{b}	2 + 3
expand e_8	$\cancel{[R_{31}, \{(k, 8)\}]}$,$[R_{33}, \{(e_7, 7), (e_3, 9)\}]$	{b,k}	2 + 3
expand e_7 & divide R_{33}	$\cancel{[R_{333}, \{(h, 9)\}]}$,$[R_{332}, \{(e_3, 9), (i, 10)\}]$	{b,k,h}	2 + 3
divide R_{332}	$\cancel{[R_{3322}, \{(i, 10)\}]}$,$[R_{3323}, \{(e_3, 9)\}]$	{b,k,h,i}	2 + 2
expand e_3	$[R_{3323}, \{(e_{11}, 10)\}]$	{b,k,h,i}	1 + 1
expand e_{11}	$\emptyset$	{b,k,h,i}	0 + 0

Fig. 11. Region-List Contents

In Fig. 11, $R_{d_1..d_l i}$ $(i = 1, 2, 3)$ denotes the region with type R_i in the space division of $R_{d_1..d_l}$.

3.3 Analysis of DCSkyline

By the theorems 1 and 2, it immediately follows that DCSkyline is correct; that is, the points generated form the skyline of the dataset. Next, we prove that DCSkyline accesses the minimum number of entries in the R-tree. We first need the following concept.

Suppose that the skyline of a dataset S is given. The part of space that have to be searched is called *skyline search region* (SSR). The SSR is the area defined by the skyline and the two axes, as depicted by the shaded area in Fig. 12.

In [10], it has been shown that every correct skyline algorithm based on R-trees must access all the entries intersecting SSR. As enforced, DCSkyline access an entry only once. Therefore, to prove the optimality of I/O in DCSkyline we need only to prove the following theorem.

Theorem 3. *Suppose that e is an entry accessed by DCSkyline. Then e intersects SSR.*

Proof: As discussed in the last subsection, an entry e accessed by DCSkyline must satisfy one of the two conditions: 1) it is the only entry left in the current R_1 (or R_2), or 2) it is the entry corresponding to R_3. Note that R_1, R_2, and R_3 may not at the same level of a space division.

Assume that on the contrary, e does not intersects SSR. There is only one possibility: e is dominated by a skyline point p. It can be immediately shown that one of the above two conditions for accessing e eliminates the possibility. □

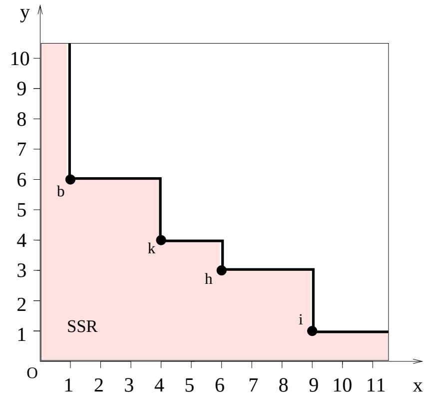


Fig. 12. SSR

Clearly, the algorithm DCSkyline is *progressive*; that is, the skyline points are generated iteratively without processing the whole dataset. We may expect that DCSkyline is more efficient than BBS since BBS has to perform a global check every time when a candidate skyline point is generated and BBS has to process a large heap list many times. Further, we may expect that DCSkyline is more efficient than NN as NN has very high I/O overheads. These, together with a small buffer requirement, make our algorithm DCSkyline combine all the advantages in the existing techniques.

4 Experimental Results

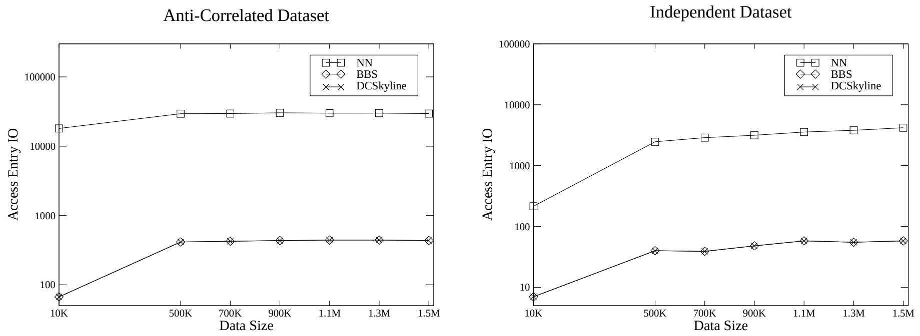
In this section, we evaluate the performance of DCSkyline. In our experiment, NN and BBS are used as the benchmark algorithms. *Independent* and *anti-correlated* [7] datasets (up to 1.5M points) are used and indexed by the $R*$-tree algorithm in [8] with node capacity of 200. Our experiment was carried out on a PENTIUM III/864 CPU with 256MB Ram.

I/O Costs

Fig. 13 illustrated the number of entries accessed in the three algorithms. Both BBS and DCSkyline accesses the minimum number of entries, which is much smaller than that accessed by NN.

Memory Utilisation

To evaluate the buffer requirements of NN, BBS, and DCSkyline, in our experiment we recorded the maximum number of elements in the to-do list of NN, in the heap of BBS, and in the region-list of DCSkyline, respectively. Note that when we count the number of elements in the region-list we count a region and an entry, respectively, as an element. Our experiment results are reported in Fig. 14.

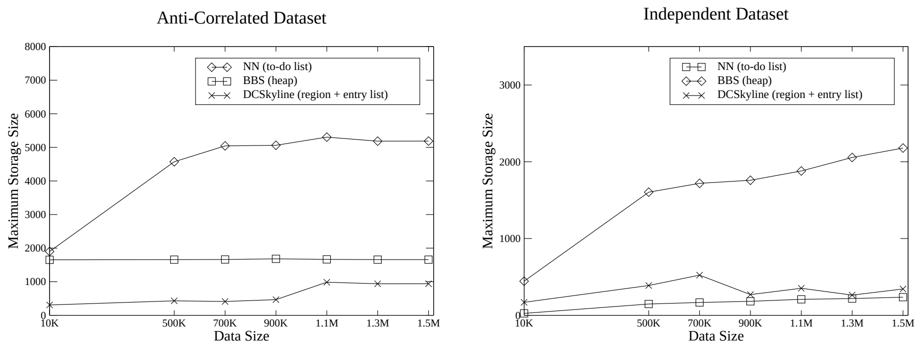


(a) Anti-correlated Datasets

(b) Independent Datasets

Fig. 13. Access Entry IO

As clearly demonstrated, DCSkyline requires much smaller buffer space than that by BBS. It is interesting to note that DCSkyline requires slightly larger buffer spaces than those required by NN for independent datasets; however DC-Skyline clearly outperforms NN for anti-correlated datasets.



(a) Anti-correlated Datasets

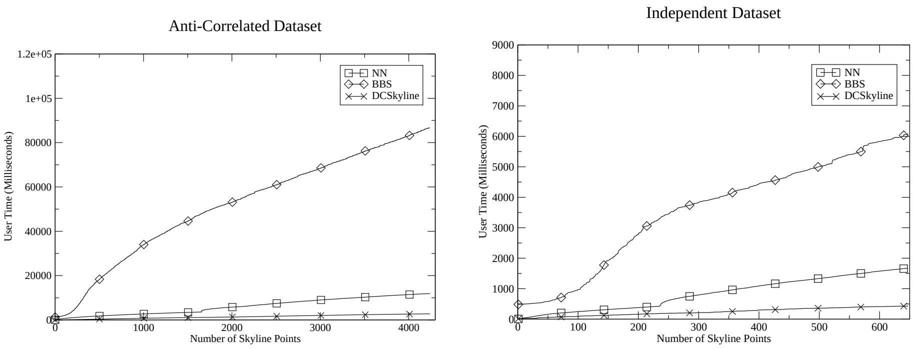
(b) Independent Datasets

Fig. 14. Maximum Storage Size

Efficiency

Fig. 15 demonstrates CPU time used in these algorithms as a function of the number of skyline points returned. DCSkyline improves the time efficiency of

BBS by orders of magnitude, even with respect to the progressive behaviours. DCSkyline is also much faster than NN.



(a) Anti-correlated Datasets

(b) Independent Datasets

Fig. 15. User Time

In summary, our experiment results demonstrated that the new algorithm DCSkyline, presented in this paper, is more efficient and requires much less memory space than those in the existing algorithms. Further, our experiments also verified the one important property of DCSkyline - a guarantee of the minimum I/O costs as what is proved in the paper.

5 Conclusion and Future Works

In this paper, we presented a novel algorithm based on R-trees for computing the skyline of a dataset in the two dimensional space. Our new algorithm guarantees the minimum I/O costs, small buffer requirements, and more efficient than the existing techniques.

As a possible future study, we will explore the ways to apply our techniques developed in the paper to a high dimensional space by using effective variations of R-tree, such as X-tree.

References

1. S. Borzsonyi, D. Kossmann and K. Stocker. "The Skyline Operator", *International Conference on Data Engineering ICDE*, 2001.
2. N. Beckmann, H. Kriegel, R. Schneider, B. Seeger. "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", *SIGMOD*, 1990.
3. H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, A. Abbadi. "Constrained Nearest Neighbor Queries", *SSTD*, 2001.

4. V. Gaede and O. Gunther. “Multidimentional Access Methods”, *Computing Surveys*, 30(2): 170-231, 1998.
5. G. Hjaltason, H. Samet. “Distance Browsing in Spatial Databases”, *ACM TODS*, 24(2):265-318, 1999.
6. H.T. Kung, F. Luccio and F.P. Preparata. “On finding the maximum of a set of vectors”, *Journal of the ACM, 22(4):469-476*, 1975.
7. D. Kossmann, F. Ramsak and S. Rost. “Shooting Stars in the Sky: An Online Algorithm for Skyline Queries”, *VLDB'02*, 2002.
8. Marios Hadjieleftheriou. “Spatial Index Library”, http://www.cs.ucr.edu/ marioh/spatialindex/index.html, 2002.
9. F.P. Preparata and M.I. Shamos. “Computational Geometry: An Introduction”, *Springer-Verlag, New York, Berlin, etc.*., 1985.
10. D. Papadias, Y. Tao, G. Fu, B. Seeger. “An Optimal Progressive Alogrithm for Skyline Queries”, *SIGMOD*, 2003.
11. N. Roussopoulos, S. Kelly, F. Vincent. “Nearest Neighbor Queries”, *SIGMOD*, 1995.
12. R. Steuer “Multiple Criteria Optimization”, *Wiley New York*, 1986.
13. K.L. Tan, P.K. Eng and B.C. Ooi. “Efficient Progressive Skyline Computation”, *VLDB*, 2001.