Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.18576/amis/100532

A Precomputed Method for Skyline Evaluation

Mei-Tzu Wang[*]

Department of Information Management, Chinese Culture University, Address 55, Hwa-Kang Road, Yang-Ming-Shan, Taipei, Taiwan, R. O. C.

Received: 30 Sep. 2014, Revised: 28 Mar. 2016, Accepted: 29 Mar. 2016
Published online: 1 Sep. 2016

Abstract: In a mobile environment, the query point may take dynamic distance as a factor to determine the skyline. But few research works involve such a situation. Most efforts are focused on static attributes in static environments. From literature, we also found that most query points are assumed on an unconstrained space. To have the query that involves dynamic distance in constrained space be evaluated more efficiently, a pre-computed method is proposed. We express dominance relation with respect to static attributes as a dominance graph, which is better suited for evaluation. Local pruning is performed within equivalence classes, whereas computing the skyline is by navigating the graph. Given the number of data points n in the dataset, the analysis indicates that the proposed method has complexity of the order of O(n). Experimental studies show that the proposed method is much more efficient and more stable than the existing method BNL. Besides, the dominance graph can be reused indefinitely until some data point changes its static attribute value.

Keywords: algorithm, fixed network, moving objects database, skyline, query processing

1 Introduction

Evaluating the skyline of a certain dataset is similar to finding the maximal element of a set of n-dimensional vectors. A vector is called a maximal element if no other vector is greater than or equal to it in all components [13], however, a skyline point may have the same value as another skyline point in all attributes [2], because neither of their attributes is dominated by another. In Fig. 1, the skyline of the dataset {a, b, c, d, e, f, g, h, i, j, k, l, m} is {a, e, j, l}. There are many applications involving skyline, e.g., capturing data packets that are now within a sliding window in a sensor network [15,17,18,25]; Searching a site for a supermarket whose average distance to each person's houses is minimal in a community [8]. Some applications need return partial skyline immediately rather than after the entire skyline is determined. [12,20, 24].

The researches on the skyline are more concerned with data points' static attributes [2,3,6,7,12,20,24,27]. In some applications such as finding the skyline of a set of hotels, the data point's distance to the moving query point is often an attribute involving the decision of the skyline[11,14]. The distance to the query point is dynamic in a mobile setting, meaning that it will change with time. This distance value can be estimated from a moving objects database, which stores the query point's motion vector, including its initial time, position (sampled by GPS) and speed [9,23]. Using the motion vector, the moving objects database can compute a query point's position in the near future with small deviation (from its actual position). Of course, another motion vector needs to be transmitted to the database, that is, its position is sampled again when the deviation between the real and estimated distances is found greater than a threshold [5, 21,26]. Moreover, the distance is not always Euclidean distance [28], others like network distance, metric distance are also applied in applications [4].

When the distance to the query point is considered as a factor to determine the skyline for the query point, it must be known whether the query or data objects are on an unconstrained space, i.e., whether Euclidean distance or network distance is used [19]. If a query point is currently at (x_1,y_1) and plans to move to (x_2,y_2) on an unconstrained Euclidean space, then the distance it shall travel to arrive there is $((x_1-x_2)^2+(y_1-y_2)^2)^{1/2}$. If, otherwise, the query point is moving on a constrained space, then the distance it shall travel is a shortest path's length. This paper is intended to propose an efficient method to compute the skyline assuming that the query point is moving on a fixed network and the distance to the query point is a dynamic attribute of the data point that takes part in determining the skyline. The moving query

[*] Corresponding author e-mail: meii@faculty.pccu.edu.tw

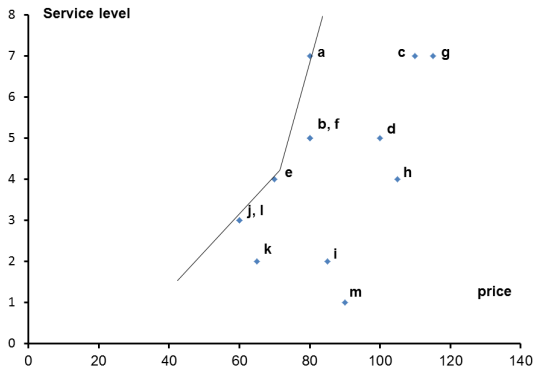


Fig. 1: The skyline {a, e, j, l} in respect of two attributes.

point is equipped with GPS, which samples the query point's current position and transmits a motion vector to the database.

The skyline query is issued at a crossroad or at the end of a line segment. This is reasonable in the road network, because only at these locations does a query point need to choose the next road segment to travel. The distance to the query point from every data point is obtained once the query point's current position is estimated from the moving objects database. As shown in Table 1, the distance to the query point from hotel f is 19.71 when the query point QP is positioned at crossroad 11 in Fig. 2. This distance value cannot be simply used to derive a new distance to the query point when QP moves to crossroad 6. It is typically computed using Dijkstra's algorithm.

The proposed Graph Based Skyline Evaluation Method (GBS) uses two layers of data structures to perform pre-computing and local pruning. All data points that have equal value in all static attributes are belonging to one equivalence class. And all equivalence classes themselves form a partially ordered set with dominance relation restricted to the static attributes as their partial ordering relation. The pre-computed graph, essentially a partially ordered set, is available for performing local pruning based on dynamic distance value whenever a skyline query is issued. The skyline can be simply obtained by navigating certain chains in the graph. Note that, once such a graph is precomputed, it can be reused until some static attribute value is changed.

The paper is organized as follows. First, we have an introduction in section 1. Then related works are reviewed in section 2. In section 3, we present basic terminologies and verify theories we have observed. In section 4, data structures are introduced and in section 5 the algorithms of constructing a dominance graph and evaluating the skyline are presented. In section 6, we have performance analysis, and in section 7 we conduct an empirical study. The conclusions are made in section 8.

2 Related Works

2.1 Block-Nested-Loops

The simple method called the nested-loops (NL) obtains the skyline by comparing every two data points in the dataset. [2] introduced an improved version of NL called block-nested-loops (BNL), which maintains a window for incomparable data points that are not dominated by anyone else so far. Each input data point is compared with those data points in the window sequentially. If the input data point is found being dominated by some data point in the window, then it must be pruned immediately. If, otherwise, it dominates some data point d_k in the window, then d_k is removed from the window. If the input data point is not dominated by all of them in the window at last, then it is incomparable with all of them and is inserted into the window or temporary file, depending on whether the window is full or not. Additional scan of dataset is necessary if there is a data point in the window that has not yet been compared with data points arrived earlier than this one in the temporary file.

The performance of this method is obviously better than NL in the average case with additional cost of maintaining a window and/or temporary file. According to the analysis of [2], BNL's complexity is of the order of O(n) in the best case and O(n^2) in the worst case, where n is the number of data points.

2.2 Divide and Conquer approach

The divide and conquer approach [13,22] was extended by [2]. The approach partitions the dataset into two smaller ones such that all data points in one partition are dominated by all those in the other partition with respect to one attribute. Then the process is repeated for another attribute on each of the two partitions and so on until all attributes are used for partitioning or a partition contains no more than one data point. Then local skylines are computed at the bottom level. After that, the neighboring partitions are merged successively with some data points of them being pruned occasionally until the skyline is obtained.

According to the study of [13], the maximum number of comparisons for dimension d >= 4 in the divide and conquer approach is lower than or equal to $O(n(log_2 n)^{d-2})$ and greater than or equal to $[log_2 n!]$, and for d = 2 or 3 the number of comparisons is lower than or equal to $O(n(log_2 n))$ and greater than or equal to $[log_2 n!]$, where n is the number of data points. [2] pointed out that the divide and conquer approach is surpassed by BNL in good cases but better than BNL in bad cases. The approach is criticized as not efficient for large dataset because it requires too much I/O operations in the partitioning process [20].

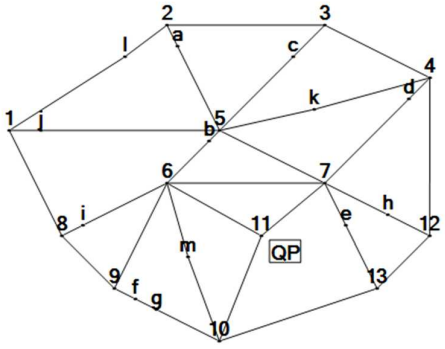


Fig. 2: A fixed network along with data points and a query object.

2.3 Nearest Neighbor (NN) Approach

The method proposed by [12] is based on the nearest-neighbor approach. By using R-tree or R*-tree index [1,10], the initial skyline points can be returned very quickly. [20] pointed out that when dimension is greater than 3, the cost incurred by the overlapping areas among partitions will increase, and will therefore produce a detrimental effect on performance.

2.4 Branch and Bound Skyline (BBS)

BBS method using R-tree was introduced by [20]. Initially all entries at the root node of the R-tree are loaded into a heap. Then an entry whose sum of coordinates is minimal in the heap is selected. If the entry is located in a leaf node of the R-tree and is not dominated by any skyline point that has been found so far, then it is claimed as a member of the skyline. If it is located in an intermediate node of the R-tree and is not dominated so far, then it is replaced by its child nodes in the heap. Otherwise, the entry is removed from the heap. BBS method can return the skyline points progressively.

2.5 Huang et al.'s Continuous Skyline

The method introduced by [11] assumes a distance function to describe the distance between a data point and a moving query point that travels on an unconstrained space. The skyline may change with time because the query point is a moving object. This method introduces an update strategy to maintain the skyline rather than reevaluating the skyline from the scratch. According to [11] the skyline changes only when some events occur. And if it occurs, some data points may be added to the skyline and some removed from the skyline. When a volatile skyline point (i.e., it is not always a skyline point) has a distance from the query point greater than that of a skyline point that dominates it with respect to static attributes, then it will be eliminated from the skyline. According to [11], the skyline will change only when the distance functions of two data points interact at some time.

3 Preliminary

The dominance relation with respect to one attribute on a dataset is based on the algebraic value as well as query point's preference on that attribute value. As shown in Table 1, hotel j's price attribute value 60 is better than hotel k's price value 65 for a query point, hence hotel k is dominated by hotel j with respect to price attribute, denoted by $j >^{price} k$. Hotel j's service_ level attribute value 3 is better than hotel k's service_level value 2 and we have $j >^{service\_level} k$.

Definition 1. *(Skyline) Given a dataset D and a set of attributes* $P = \{p_1, p_2, \ldots p_{m-1}, p_m\}$*, the data point* d_r *is said being dominated by another data point* d_i*, denoted by* $d_i > d_r$*, if it satisfies the following conditions:*

$$\neg\exists p_x(p_x \in P \wedge d_r >^{p_x} d_i)$$
$$\exists p_a(p_a \in P \wedge d_i >^{p_a} d_r)$$

The set of data points that are not dominated by all others is called the skyline denoted by Sky.

In Fig. 2, a query point QP is positioned at crossroad 11 at time t. Table 1 shows all data points' distances from QP at time t. We can see that hotel d is dominated by hotel b, but b is not dominated by hotel a, and the skyline is {a, b, e, g, k, l}. The skyline may change when the query point moves to a new crossroad.

If the skyline is obtained by comparing every data

Table 1: Data points with their static attributes and dynamic distances to QP in Fig. 2.

Hotel	Price	service_level	Distance
a	80	7	26.31
b	80	5	15.95
c	110	7	27.27
d	100	5	19.12
e	70	4	12.28
f	80	5	19.71
g	115	7	17.48
h	105	4	14.52
i	85	2	19.24
j	60	3	34.37
k	65	2	26.59
l	60	3	33.55
m	90	1	17.58

point with all others, it requires n(n−1)/2 comparisons. Fortunately, the method can be improved. According to [11], some data points called permanent skyline points, such as hotel a, can always be part of the skyline no matter how far they are from the query point. Hence it needs not be recomputed each time to verify that they are skyline points. There is a difference between our model and that proposed by [11] in pruning process. We perform pruning by identifying the set of data points that are not dominated by others with respect to static attributes, the so called permanent skyline points, mentioned by [11] and also the sets of data points that have equal value in each static attribute. When all such data points are collected together in advance, the cost of pruning can be reduced as will be verified in theorem 1.

From definition 1, we can see that two skyline points may have equal value in each attribute, and that the dominance relation with respect to static attributes is a partial ordering relation, i.e., it is reflexive, antisymmetric, and transitive [16]. The partial ordered set consists of a set of chains with a feature that these chains may be overlapped in parts with others.

Definition 2. *(Restricted dominance relation) Given a dataset D, the dominance relation restricted to static attributes on D is called a static dominance relation denoted by* $>^s$. *And y is said statically dominated by x if x* $>^s$ *y. The skyline based on the restricted relation is called Static_sky; and the remaining data points belong to the set called Static_dominated.*

From Fig. 1, we can see that Static_sky consists of a, e, j, l. Hotel b is statically dominated by hotel a, and both hotels d and h are statically dominated by hotel a, b and f. Thus Static_dominated contains data points b, c, d, f, g, h, i, k and m.

Theorem 1. *If a data point is belonging to Static_sky, then, no matter how far it is from the moving query point, it is a skyline point or dominated only by those data points whose static attributes' values are all equal to it.*

Proof. Let D be a dataset with attributes $P = \{p_1, p_2, \ldots, p_{m-1}, p_m\}$, where the first m−1 attributes are static and the last is a dynamic attribute recording the shortest distance to the moving query point. If $d_i \in Static\_sky$ and $d_i \notin Sky$, then there exists $d_j \in D$ such that $d_j > d_i$. And the two conditions are satisfied:

$$\neg\exists p_x (p_x \in P \wedge d_i >^{p_x} d_j) \quad (1)$$

$$\exists p_a (p_a \in P \wedge d_j >^{p_a} d_i) \quad (2)$$

The attribute p_a in (2) cannot be static, otherwise, by (1) and (2) and definition 2, d_i is not belonging to Static_sky, violating the assumption that $d_i \in Static\_sky$. Hence, d_j and d_i have equal values in $p_1, p_2, \ldots, p_{m-1}$ attributes. The theorem is proved.

As in the example road network, the hotels a, e, l in Static_sky = {a, e, j, l} become skyline points when the query point QP moves to crossroad 11. Hotel j does not belong to Sky, because it is dominated by hotel l: they have equal price and service level but hotel l's distance to QP is less than that of hotel j. To determine whether a data point of Static_sky is belonging to skyline, according to theorem 1, we only need to check those data points whose static attributes are all equal to this data point. All those whose static attributes are not all equal to anyone else are the so called permanent skyline points. For them there is no need to make any comparison with others by theorem 1.

Theorem 2. *A data point* $d_i \in$ *Static_dominated is becoming a member of the skyline if, and only if, its distance to the query point is less than that of those which statically dominate it and less than or equal to that of those whose values are equal to* d_i *in all static attributes.*

Proof. Let D be a dataset with attributes $p_1, p_2, \ldots, p_m$, where p_m recording a data point's shortest distance to the query point, and P' denotes the set of the first m-1 attributes, i.e., the set of static attributes $p_1, p_2, \ldots, p_{m-1}$. Let d_i be a data point ∈ Static_dominated, and let D1 = $\{d_r | d_r \in \text{D} \wedge d_r >^s d_i\}$ and D2 = $\{d_r | d_r \in \text{D} \wedge (d_r.p_k = d_i.p_k, \forall p_k \in \text{P'})\}$.

For every data point $d_r \in$ D1, if we have $d_i.p_m < d_r.p_m$, that is, $d_i >^{p_m} d_r \ \forall d_r \in$ D1, then d_i is not dominated by any data point in D1 by definition. Next, for every $d_r \in$ D2, if we have $d_i.p_m \leq d_r.p_m$, then $\neg\exists d_r \in$ D2 such that $d_r >^{p_m} d_i$. So, d_i is not dominated by any data point in D2. Finally, for every data point $d_j \in$ (D − D1 − D2), by definition 1, there exists $p_k \in$ P' such that $d_i >^{p_k} d_j$, and hence d_i is not dominated by d_j. We therefore have the consequence that d_i is a member of the skyline.

If $d_i \in$ Sky and $\exists d_r \in$ D1 such that $d_i.p_m \geq d_r.p_m$ or $\exists d_s \in$ D2 such that $d_i.p_m > d_s.p_m$, then $d_i \not>^{p_m} d_r$ or $d_s >^{p_m} d_i$. And, by definition of D1, $\exists p_k \in$ P', $d_r >^{p_k} d_i$, hence we have $d_r > d_i$. Also, by definition of D2, $\neg\exists p_k \in$ P', $d_i >^{p_k} d_s$, hence we have $d_s > d_i$. Both contradict the assumption that d_i is a member of the skyline. It is concluded that if $d_i \in$ Static_dominated and also $d_i \in$ Sky, then $\forall d_r \in$ D1, $d_i.p_m < d_r.p_m$, and $\forall d_s \in$ D2, $d_i.p_m \leq d_s.p_m$. The theorem is proved.

4 Data structure

The use of dominance graph to describe dominance relation among data points has been found in literature, such as [29] in the topic of Top-k queries. But there is a difference between the dominance graph of [29] and ours. First, each node of our dominance graph corresponds to a group of data points, whereas the dominance graph proposed by [29] corresponds each node to a data point.

Second, the navigation method used in our dominance graph is based not only on the structure of the graph, i.e., representing the occurrence of dominance between two data points, but also on the dynamic attribute values of data points, whereas in [29] the search method is based on an aggregate monotone function without involving the dynamic attribute. Third, the two dominance graphs have different purposes. We use it to find the skyline, whereas [29] answers the Top-k query.

The dominance graph is proposed to represent the static dominance relation $>^s$ on a dataset, in which every vertex in the graph corresponds to an equivalence class. The fixed network is also represented by a graph, which is integrated with data points to facilitate the computation of distance between two vertexes /data points.

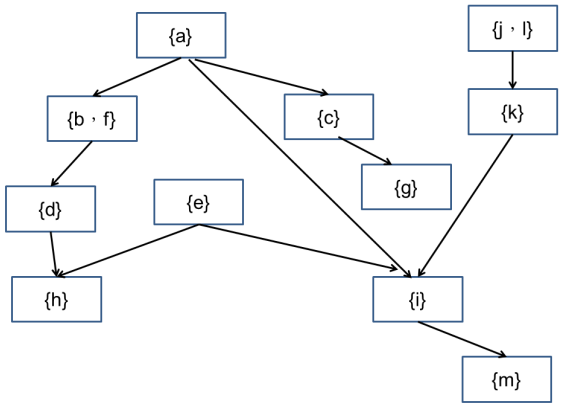


Fig. 3: Static dominance graph example.

4.1 Equivalence class

According to the theorems above, a data point cannot become a skyline point without having its distance to the query point less than or equal to those whose values are equal to it in all static attributes. It leads us to an idea of partitioning the dataset into equivalence classes for pruning. And this method is particularly useful for medium size of equivalence class.

Definition 3. *(SD-Equivalence class) Let D be a dataset with attributes* $p_1, p_2, \ldots, p_{m-1}, p_m$*, where the first* $m-1$ *attributes are static. An SD-Equivalence class* D_i *is composed of those data points that have equal values in all static attributes, i.e.,* $\forall d_i, d_r \in D_i$, $\forall p_k \in \{p_1, p_2, \ldots, p_{m-1}\}$*, we have* $d_i.p_k = d_r.p_k$*. This induced equivalence relation between* d_i *and* d_r *is denoted by* $d_i \equiv d_r$*.*

The equivalence relation is reflexive, symmetric and transitive. This makes it possible to partition the dataset into equivalence classes. There are eleven equivalence classes derived from the data points in Fig. 1. Local pruning in an equivalence class is triggered by a query point at a crossroad. As an example, hotels f , b in an equivalence class have new distances 19.71, 15.95 to the query point, respectively, when the query point arrives at crossroad 11. By Theorem 2, hotel f is pruned. All data points are to be pruned in an equivalence class except those that have minimum distance to the query point. After pruning, the remaining data points are candidates and shall be checked further for becoming skyline points according to Theorem 1 and Theorem 2.

4.2 Static dominance graph

The candidates resulting from local pruning need to compare with those that statically dominate them for determining the skyline. The static dominance graph is suggested for this purpose. First, the dominance relation on equivalence classes $>^c$ is introduced as follows.

Definition 4. *(Dominance relation on SD-Equivalence classes) An equivalence class* e_2 *is statically dominated by another equivalence class* e_1*, denoted by* $e_1 >^c e_2$*, if* $\forall d_i \in e_1, \forall d_j \in e_2$, $d_i >^s d_j$*. Furthermore, if there does not exist other vertex* e_3 *such that* $e_1 >^c e_3$ *and* $e_3 >^c e_2$*, then* e_2 *is called immediately dominated by* e_1*.*

The definition is consistent because all data points in an equivalence class have equal valuse in all static attributes.

Definition 5. *(Static dominance graph) A static dominance graph is a directed graph G = (V, E), where each vertex of V corresponding to an equivalence class, and each edge* (v_1, v_2) *of E corresponding to a static dominance relation.*

As an example, a static dominance graph consisting of six chains is shown in Fig. 3, which is constructed using data points in Fig. 1. When the query point reaches crossroad 11, a member h of Static_dominated becomes a candidate. To know whether it is a skyline point, the candidates above h in two chains are checked, i.e., a, b, d, e. It is similar to other cases: for b, the hotel needs be checked is a; for a, there is no predecessor and thus none are checked.

Both pruning data points in an equivalence class and navigating a static dominance graph to find skyline points involve comparing dynamic distances of data points to the query point.

4.3 Implementing the vertex structure

The vertex of a static dominance graph implemented as an object of class Vertex is described below:

```
Class Vertex (extent V){
   attribute integer in_degree;
   attribute float bound;
   attribute integer count;
   attribute boolean flag_domi;
   relationship
      set<Vertex> adjacency_list;
   relationship
      list<DataObject> equ_member
      inverse DataObject:: equ_class;
   void vertex(){in_degree=0,
                 flag_domi=FALSE}}
```

The in_degree attribute records the number of edges incident to this vertex, e.g. vertex {i} has value 3, and vertex {j, l} has value 0 in Fig. 3. The equ_member relation associates this vertex with an SD-Equivalence class that consists of a list of data points with the first one having the minimum distance to the query point in the SD-equivalence class. As shown in Fig. 4, vertex {j, l} has an SD-Equivalence class containing data points l, j and l is the first element having minimum distance 33.55 to the query point among {33.55, 34.37} at the query time. The adjacency_list relation associates this vertex with its immediate descendants in the static dominance graph, e.g. the adjacency_list of vertex {e} is consisting of {h} and {i}.

The bound attribute records the minimum of the distances of the candidates in equ_member and their predecessors to the query point. The count attribute records the number of chains not yet being visited for this vertex while navigating the dominance graph. And the flag_domi attribute is initialized as FALSE and used to indicate the status about whether the candidates in the vertex's equ_member are dominated by others by now.

All attributes' values may change during the navigation except for the attribute in_degree. Fig. 4 shows three vertexes' states after the pruning is completed.

vertex: { i }	3	19.24	3	False	{m}	i
vertex: {j,l}	0	33.55	0	False	{k}	l, j
vertex: {e}	0	12.28	0	False	{h}, {i}	e

Fig. 4: States of three vertexes after local pruning.

4.4 Implementing the data point structure

The data point implemented as an object of class DataObject is described below:

```
Class DataObject (extent D) {
   attribute integer service_level;
   attribute integer price;
   attribute float distance;
   attribute float x_coordinate
   attribute float y_coordinate
   attribute boolean mark_equ;
   relationship
      vertex equ_class
      Inverse Vertex:: equ_member;
   void DataObject(){mark_equ=FALSE}}
```

The data object's two attributes service_level, price are assumed to be static and used to construct the static dominance graph. The attribute distance to the query point is dynamic, because the query point is a moving object. Once the query point's position is known from the moving objects database, the distance is obtained from the pre-computed array SD(s, d_i), where s is the crossroad the query point is positioned and d_i is a data point. The x_coordinate and y_coordinate attributes record the data point's position on the fixed network. The attribute mark_edu is auxiliary for computation. The equ_class relation is the inverse of the relation equ_member on class Vertex, and it associates the data point with its belonging equivalence class.

4.5 Implementing the fixed network

For the purpose of obtaining the values of array SD(s, d_i), we integrate data points into the fixed network. Each node in the fixed network represents either a crossroad or data point, and both are implemented as an object of class Node as described below:

```
Class Node (extent N) {
attribute float distance;
attribute float x_coordinate;
attribute float y_coordinate;
attribute boolean flag;
relationship
   set<NeighborNode>neighbor_list;
   Inverse NeighborNode:: neighbor;
relationship
   Node previous}
```

The three attributes distance, x_coordinate, y_coordinate of class Node have the same meaning as those of class DataObject. The flag attribute is an indicator of the status of the node object used in ComputeDistanceAll algorithm. The neighbor_list relation associates the node with a set of neighbors on the fixed network. A neighbor of the node is either a crossroad or a data point. The

previous relation associates the node with its previous node in a shortest path from the node back to the query point.

The neighbor of a Node object is implemented as an object of class NeighborNode as described below:

```
Class NeighborNode (extent NB){
attribute float weight;
relationship
    Node node;
relationship
    Node neighbor;
    inverse Node:: neighbor_list
relationship
    DataObject dataobject}
```

As a Node object's neighbor, the NeighborNode object relates this Node object through neighbor relation, and records its distance to this Node object in the weight attribute. A NeighborNode object corresponds to a node in the fixed network via node relation, and also corresponds to a data point via dataobject relation if it is a data point.

In Fig. 2, crossroad 2 is a Node object associated with a set of three NeighborNode objects via neighbor_list relation. The first NeighborNode object corresponds to Node object a via node relation with weight attribute value equal to 2.24 (i.e., the distance between crossroad 2 and node a), and it also corresponds to data point a via dataobject relation, because it is a data point. Similarly, the second NeighborNode object corresponds to node object 1 via node relation and corresponds to data point l via dataobject relation. The third neighbor is crossroad 3 and there is no associated instance for it on dataobject relation, i.e., its value is null.

5 Algorithms

Based on the developed theorems and data structures above, skyline can be evaluated by the following rule.

Rule: If a data point is becoming a candidate in its equivalence class and its distance to the query point is less than all candidates above it in chains of the static dominance graph, then it is becoming a skyline point.

The proposed Graph Based Skyline Evaluation Method has two phases. The first phase is to construct the static dominance graph for the road network and dataset, and compute the distance between every crossroad and every data point. The second phase is to find skyline through pruning in equivalence classes and navigating related chains in the graph. While the first is performed once, the second is performed whenever skyline query is issued.

5.1 Constructing the static dominance graph

There are three steps to construct a static dominance graph as shown in Algorithm 1. First, the equivalence classes are determined from data points and are corresponding to vertexes in the graph. Second, the edges incident from a vertex are determined by finding vertexes whose corresponding equivalence classes are immediately dominated by that vertex's corresponding equivalence class. There are as many edges incident from a vertex as elements in that vertex's adjacency_list. Third, we count the number of edges incident to a vertex, and store it as the initial value of count attribute to help maintain chains that are by now not yet being visited for that vertex while evaluating skyline.

Algorithm 1 GenDomiGraph.

```
1: call GenEquClass (D)
2: call GenAdjencyList (V)
3: call ComputeInDegree (V)
```

Algorithm 2 GenEquClass.

```
Require: D: dataset
 1: for all d_i ∈ D do
 2:     if d_i.mark_equ = true then
 3:         continue {skip following steps}
 4:     end if
 5:     v_i = new vertex (), add d_i to v_i.equ_member
 6:     d_i.mark_equ = true
 7:     for all d_r ∈ D do
 8:         if d_r = d_i then
 9:             continue {skip following steps}
10:         end if
11:         if d_r.mark_equ = true then
12:             continue {skip following steps}
13:         end if
14:         flag_equ = true
15:         for all static attribute p_k in DataObject do
16:             if d_r.p_k ≠ d_i.p_k then
17:                 flag_equ = false, break
18:             end if
19:         end for
20:         if flag_equ = true then
21:             insert d_r into v_i.equ_member
                d_r.mark_equ = true, d_r.equ_class = v_i
22:         end if
23:     end for
24: end for
```

In algorithm 2, a vertex is created (line 5) if a data point with mark_equ attribute value equal to FALSE is found, i.e., its equivalence class is by now not yet being created (line 2-4). Then the data point is inserted into the new vertex's corresponding equivalence class (line 5) with mark_equ attribute value changed to TRUE (line 6). Note that mark_equ attribute value is initialized as FALSE by the constructor of DataObject class. Next all other data points with static attributes' values equal to those of that

data point (line 14-19) and mark_equ attribute values equal to FALSE (line 11-13) are inserted into the same equivalence class. Finally, their mark_equ attribute values are changed to TRUE (line 20- 21).

In algorithm 3, all edges incident from a vertex are created and stored in that vertex's adjacency_list. There are two steps. First, every vertex v_i created by algorithm GenEquClass is compared with every other vertex v_r to determine whether v_r is statically dominated by v_i (line 6-11). The first element of $v_r's$ equivalence class, i.e., $v_r.equ\_member.0$, and that of $v_i's$ equivalence class, i.e., $v_i.equ\_member.0$, are selected to compare with each other. There is no difference about which element is selected from an equivalence class for comparison because they all have equal static attributes' values. If vertex v_r is dominated by v_i, then it is temporarily added to $v_i's$ adjacency_list (line 12-14). Note that v_i and v_r have equal static attributes' values, except they are the same vertex. Second, we must remove those v_r that are not immediately dominated by vertex v_i from $v_i's$ adjacency_list (line 17-21). After that, the remaining vertexes in $v_i's$ adjacency_list are vertex $v_i's$ immediate descendants, and we obtain the adjacency_list of v_i. It means that all edges incident from v_i are obtained.

Algorithm 3 GenAdjacencyList algorithm

Require: V: vertexes
1: **for all** $v_i \in V$ **do**
2: **for all** $v_r \in V$ **do**
3: **if** $v_r = v_i$ **then**
4: continue {skip following steps}
5: **end if**
6: flag = true
7: **for all** *static attribute* p_k *in DataObject* **do**
8: **if** $v_r.equ\_member.0 >^{p_k} v_i.equ\_member.0$ **then**
9: flag = false, break {v_r not dominated by v_i}
10: **end if**
11: **end for**
12: **if** $flag = true$ **then**
13: insert v_r into $v_i.adjacency\_list$
14: **end if**
15: **end for**
16: **end for**
17: **for all** $v_i \in V$ **do**
18: **for all** $v_r \in v_i.adjacency\_list$ **do**
19: $v_i.adjacency\_list -= v_r.adjacency\_list$
20: **end for**
21: **end for**

In algorithm 4, we count the number of edges that are incident into a vertex in the graph, i.e., counting the adjacency_lists of all vertexes that contain this vertex (line 2-4). This is the initial value of count attribute that will be used by ComputeSkyline algorithm described below.

Algorithm 4 ComputeInDegree

Require: V: vertexes
1: **for all** $v_i \in V$ **do**
2: **for all** $v_r \in v_i.adjacency\_list$ **do**
3: $v_r.in\_degree += 1$
4: **end for**
5: **end for**

5.2 Computing distances between two nodes

In algorithm 5, the shortest distance between every crossroad and every data point is computed based on Dijkstra's algorithm and stored in array SD for later use.

Algorithm 5 ComputeDistanceAll

Require: N: Node extent, NB: NeighborNode extent
1: ND = empty
2: **for all** $nb \in NB$ **do**
3: **if** $nb.dataobject \neq null$ **then**
4: insert nb.node into ND {collect data points}
5: **end if**
6: **end for**
7: **for all** $s \in N - ND$ **do**
8: **for all** $n \in N$ **do**
9: n.distance= ∞, n.flag = false {initialization}
10: **end for**
11: s.distance = 0 {a crossroad's distance to itself}
12: **while** true **do**
13: m_d = ∞
14: **for all** $n \in N$ **do**
15: **if** $n.flag = false \wedge m\_d > n.distance$ **then**
16: u = n, m_d = n.distance
17: **end if**
18: **end for**
19: **if** $m\_d = \infty$ **then**
20: break {all nodes are visited}
21: **end if**
22: u.flag = true {u is marked as a visited node}
23: **for all** $a \in u.neighbor\_list$ **do**
24: **if** $a.node.distance \leqq u.distance + a.weight$ **then**
25: continue {skip following steps}
26: **end if**
27: a.node.distance = u.distance + a.weight
28: a.node.previous = u
29: **if** $a.dataobject \neq null$ **then**
30: SD(s, a.dataobject) = a.node.distance
31: **end if**
32: **end for**
33: **end while**
34: **end for**

The algorithm starts with checking all NeighborNode objects' dataobject relationship instances. If they are not null, then the NeighborNode objects corresponds to data points and the corresponding Node object are inserted into ND (line 1-6). After that, all crossroads are collected in N-ND. Each Node object's distance to the source

crossroad, s, is initialized as infinite and the flag attribute is initialized as FALSE (line 8-10). The source crossroad's distance to itself is initialized as 0 (line 11). Next, we check all Node objects that are still not visited, and select the one with minimum distance to s as the next to visit (line 13-18). All neighbors of that selected Node object are checked (line 23-32). If a neighbor's distance to the source crossroad is greater than its distance to this Node object (the weight attribute value) plus this Node object's distance to the source crossroad, then the neighbor's distance to the source crossroad is updated (line 24-27). The previous node back to the source crossroad in the shortest path is recorded (line 28). If the neighbor corresponds to a data point, then the distance together with the NeighborNode object and source crossroad is added to the SD array (line 29-31).

Note that the shortest path from a data point to a crossroad s is found by following the previous relation of the Node class back to s. As an example, a shortest path from hotel a to crossroad 11 is (a, 5), (5, b), (b, 6), (6, 11) with weight attribute values 8.94, 1.41, 5.66, 10.3 respectively. Therefore the path length is 26.31. Note that array SD obtained by this algorithm will be used by ComputeCandidate algorithm.

5.3 Evaluating the skyline

Algorithm 6 first calls ComputeCandidate to find candidates from dataset and then calls ComputeSkyline to obtain skyline by navigating the static dominance graph.

Algorithm 6 SkylineQuery

```
1: call ComputeCandidate (s, V, SD)
2: call ComputeSkyline (V, SKY, Q)
```

In algorithm 7, Sky and Q are initialized as empty (line 1) and all distances between query point and data points are assumed being different. In every equivalence class, the data point with minimum distance to the query point is found (line 3-8). We switch the data point and the first member in the equivalence class and replace the bound attribute value by the data point's distance to the query point if the data point's distance to the query point is less than the first member's distance (line 4-6). All vertexes with in_degree 0, i.e. the maximal elements of the graph, are put into Q for visit by ComputeSkyline algorithm and the corresponding candidates are put into SKY (line 9-11).

In algorithm 8, we find all skyline points except for those that have been found in algorithm 7, i.e., the candidates stored in SKY. We first remove a vertex v_i from Q (line 6). Then we check each child v_j of v_i (line 7). We have two cases.

First, all chains that include vertex v_j have already been visited for vertex v_j except for the one including

Algorithm 7 ComputeCandidate algorithm

```
Require: s: a crossroad, V: vertexes, SD: an array
 1: Sky = empty, Q = empty
 2: for all v_i ∈ V do
 3:    for all d_r ∈ v_i.equ_member do
 4:       if SD(s,d_r) < SD(s,v_r.equ_member.0) then
 5:          switch d_r and equ_member.0
 6:          v_i.bound = SD(s,d_r)
 7:       end if
 8:    end for
 9:    if v_i.in_degree = 0 then
10:       add v_i to Q, add v_i.equ_member.0 to SKY
11:    end if
12: end for
```

both v_i and v_j (line 8). If $v'_j s$ bound attribute value is less than that of all predecessors along the chain and the candidate in $v'_j s$ equivalence class has never been dominated by some in other chains (line 9), then the candidate in $v'_j s$ equivalence class is a skyline point and is added to Sky (line 10). If, otherwise, $v'_j s$ bound attribute value is greater than $v'_i s$ bound value, then the bound attribute value is replaced by $v'_i s$ bound attribute value (line 12-13). Obviously, under this condition the candidate in $v'_j s$ equivalence class must be dominated by some candidate in $v'_j s$ predecessors. In both conditions, v_j is added to the queue Q for a visit later as long as vertex v_j has at least one edge being incident from it (line 16-18).

Second, there are other chains not yet being visited for v_j. Then $v'_j s$ count attribute value is decremented by 1 (line 20). If $v'_j s$ bound attribute value is not less than $v'_i s$ bound attribute value (line 21), then the candidate in $v'_j s$ equivalence class must be dominated by some candidates in its predecessors in this chain. So, $v'_j s$ flag_domi attribute value is replaced by TRUE and bound attribute value changed to $v'_i s$ bound attribute value (line 21-23).

6 Analysis of algorithms

Both GenEquClass and GenAdjacencyList algorithms have complexity of the order of $O(n^2)$ for the worse case, where n is the number of data points.The best case for GenEquClass is $O(n^2)$. And the best case for GenAdjacencyList is O(1), where the number of equivalence classes is one. The complexity of ComputeInDegree is of the order of $O(n^2)$ for the worse case and O(1) for the best case. The ComputeDiatanceAll algorithm has complexity of the order of $O(c \times n^2)$, where c is the number of crossroads. The algorithms GenEquClass, GenAdjacencyList, ComputeInDegree in phase 1 are perform once for constructing the static dominance graph, which can be used indefinitely until some data point's static attribute values are changed or some data point vanishes or move to other locations.

Algorithm 8 ComputeSkyline

Require: V: vertexes, SKY: skyline points, Q: a queue
1: **for all** $v_i \in V$ **do**
2: $v_i.count = v_i.in\_degree$
3: $v_i.flag\_domi = false$
4: **end for**
5: **while** $Q \neq empty$ **do**
6: remove v_i from Q
7: **for all** v_j in adjacency_list of v_i **do**
8: **if** $v_j.count = 1$ **then**
9: **if** $v_j.bound < v_i.bound \wedge v_j.flag\_domi = false$ **then**
10: add $v_j.equ\_member$.0 to SKY
11: **else**
12: **if** $v_j.bound > v_i.bound$ **then**
13: $v_j.bound = v_i.bound$
14: **end if**
15: **end if**
16: **if** $v_j.adjaceny\_list \neq empty$ **then**
17: add v_j to Q {v_j has descendant}
18: **end if**
19: **else**
20: v_j.count = v_j.count - 1 {more chains to visit}
21: **if** $v_j.bound \geq v_i.bound$ **then**
22: v_j.bound = v_i.bound, v_j.flag_domi = true
23: **end if**
24: **end if**
25: **end for**
26: **end while**

In the second phase, the algorithm ComputeCandidate has complexity of the order of O(n). And the algorithm ComputeSkyline is of the order of O(n) for the worse case and O(1) for the best case. Hence, the algorithm SkylineQuery has complexity of the order of O(n) for the worse case and O(1) for the best case.

7 Experiments

In this section, the algorithm SkylineQuery in the second phase of GBS is evaluated. The experiments are executed on Intel(R) Core(TM) 2 CPU 1.83 GHz with 0.99 GB RAM. The operating system is Microsoft Windows XP Professional version 2002, and implementation language is VB.NET 2005.

The SkylineQuery algorithm including ComputeCandidate and ComputeSkyline algorithms is measured against BNL. In the experiments, GBS's execution time is compared with BNL's execution time. There are three parameters in the experiments: cardinality n, representing the number of data points; degree m, representing the number of attributes in the dataset, including m-1 static attributes and one dynamic distance attribute; and the number of distinct values, d, in each attribute. We design six scenarios for experiments: varying one parameter's value with the others fixed at two values.

Like GBS, BNL is implemented with the dynamic distance attribute values extracted from the array SD. Both SkylineQuery and BNL are executed 10 times in each scenario. Then they are averaged for both methods. The experiments are repeated five times and four of them are selected and averaged. The reason is to eliminate the effect caused by occasionally running system tasks in a multitasking environment. The four average values are very close in almost all cases. For BNL, two window sizes are set: small window size 2, and large window size defined by the maximal number of skyline points for that scenario.

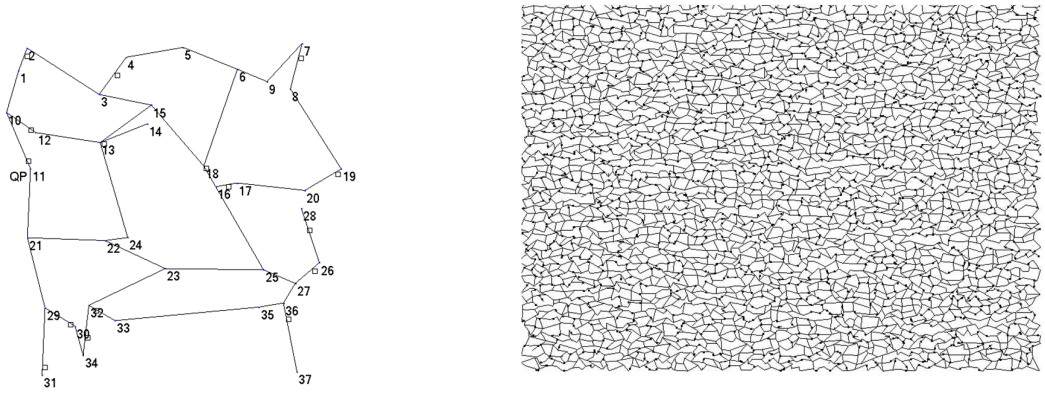
7.1 Set up experiments

The fixed network together with data points is generated synthetically as follows. First of all, a two-dimensional array of blocks is created. Then we create randomly one to three crossroads in each block based on uniform distribution. For the block with two crossroads, an edge is created to connect them. For the block with three crossroads, two edges are created to connect them. Then two horizontally neighboring blocks are connected by creating an edge between a block's most left crossroad and its left neighbor's most right crossroad. Unlike horizontally neighboring blocks, two vertically neighboring blocks are connected with probability 0.8. Once two blocks are to be connected, an edge is created between a block's bottom crossroad and the other one's top crossroad. For blocks in the first or last columns, two vertically neighboring blocks are always connected to prevent isolated blocks. There is a probability 2/3 that an edge contains a data point. The sample fixed network with 4×4 blocks is shown in Fig. 5(a), and 55×55 blocks in Fig. 5(b).

For simplicity, ComputeCandidate, ComputeSkyline and SkylineQuery algorithms are denoted by GBS_L, GBS_C and GBS, respectively. The execution time of GBS is approximated by the sum of GBS_L's execution time and GBS_C's execution time.

7.2 Effect of the cardinality of dataset

The effect of the cardinality of dataset on the performance of GBS and BNL is investigated. First, the cardinality of the dataset, n, is varied from 2K to 16K with degree m fixed at 4 and the number of distinct attribute values, d, fixed at 3. Fig. 6(a) shows execution time versus the cardinality of dataset. GBS's execution time is raised almost linearly from 0.2 ms to 6.3 ms as the cardinality of dataset varies from 2K to 16K. GBS's execution time and GBS_L's execution time are very close because GBS_C's execution time is nearly 0 at n = 2K and 0.1 at n = 4K to 16K. It means that local pruning in the equivalence class is insensitive to the cardinality of dataset in the execution



(a) (b)

Fig. 5: Fixed networks, with 4×4 blocks (a), 55×55 blocks (b).

time. GBS takes less time than BNL by about half of an order of magnitude on average.

Second, the experiments are conducted with degree m fixed at 6 and the number of distinct attribute values, d, fixed at 5. The execution time of GBS in Fig. 6(b) is raised linearly from 2.4 ms to 15.6 ms as the cardinality of dataset varies from 2K to 16K and GBS takes less execution time than BNL for two window sizes by about half of an order of magnitude on average. GBS takes longer execution time than that in the first case due to more attributes and more distinct values in each attribute. Both may cause more equivalence classes, e.g., it is raised rapidly to 3100 at n = 16K as compared with 27 for the first case as shown in Fig. 7. GBS's execution time at n = 12K and n = 14K are equal (i.e. 12.4 ms). The reason may possibly be explained as randomness: the advantage of less comparison due to smaller n may be offset by more switching operations in equivalence classes, which occurs randomly in an equivalence class. It can be seen that GBS behaves much more stable than BNL. The produced skyline points are shown in Fig. 8.

7.3 Effect of the degree of dataset

The effect of the degree of dataset on the performance of GBS and BNL is investigated. The degree parameter m is varied from 2 to 6 with cardinality n fixed at 10K and the number of distinct attribute values, d, fixed at 2. The execution time of GBS is about 6-8 times faster than that of BNL for a small window size 2, and 7-11 times faster than that of BNL for a large window size 6, as shown in Fig. 9(a). The second experiment is conducted with cardinality n fixed at 16K and the number of distinct attribute values d fixed at 6. It is seen from Fig. 9(b) that the execution time of GBS is about 3-16 times faster than that of BNL for a small window size 2, and 4-17 times faster than that of BNL for a large window size 112. Both experiments show that GBS is more stable than BNL for various degree, especially for the second experiment. When degree varies from 2 to 6, there is roughly 247% increase (i.e. 15.6 ms) in the execution time of GBS. But for BNL with a large window, the increase in the execution time is roughly 911% (i.e. 336.3 ms) significantly greater than that of GBS.

Higher degree of dataset, i.e., more attributes in a dataset, tends to have more distinct values in a dataset, and will therefore lead to more equivalence classes as shown in Fig. 10. When the number of equivalence classes increases, the static dominance graph will have more vertexes, and thus GBS_C will take longer execution time. As degree varies from 5 to 6 in the second experiments, the increase in GBS's execution time is 12.5 ms, which is significantly less than the increase 163.9 ms for BNL with a small window size, and less than the increase 233.9 ms for BNL with a large window size, even when the number of equivalence classes is greatly increased by 5472 as shown in Fig. 10. Moreover, higher degree tends to have more skyline points as shown in Fig. 11. When the degree varies from 5 to 6 in the second experiment, BNL's execution time increases rapidly, so is the increase 85 of the number of produced skyline points.

7.4 Effect of the number of distinct attribute values

The effect of the number of distinct attribute values on performance is measured by varying the number of distinct values in each attribute, d, from 2 to 6 with cardinality n fixed at 10K and degree m fixed at 3 in the first experiment as shown in Fig 12(a). The second experiment is conducted with cardinality n fixed at 16K

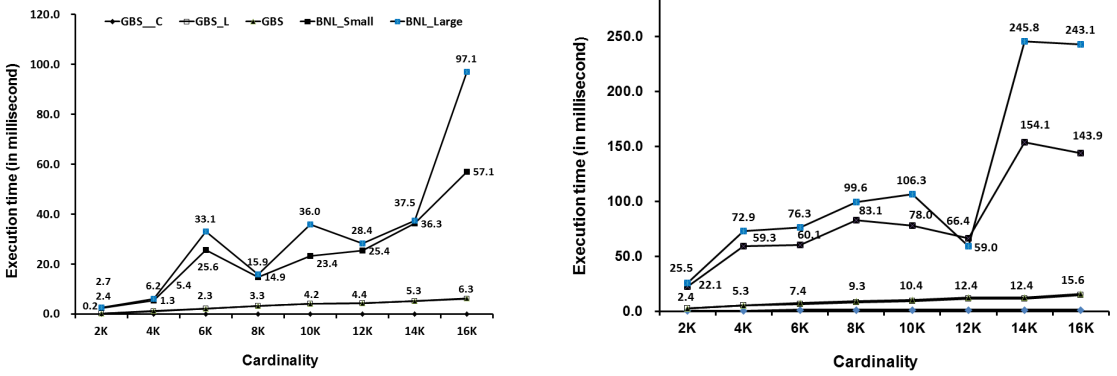


Fig. 6: Execution time versus cardinality with m=4, d=3 (a), m=6,d=5 (b).

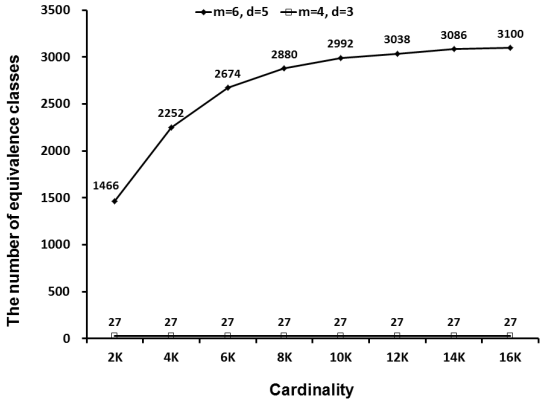


Fig. 7: The number of equivalence classes versus cardinality.

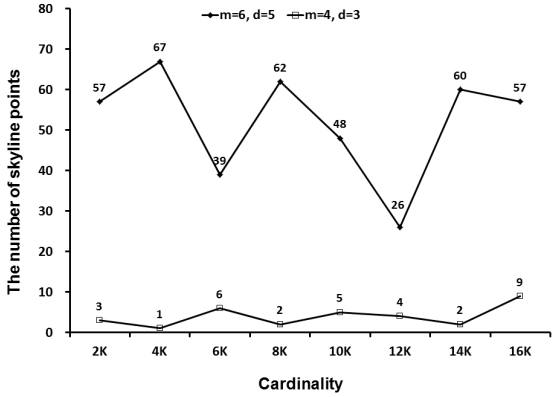


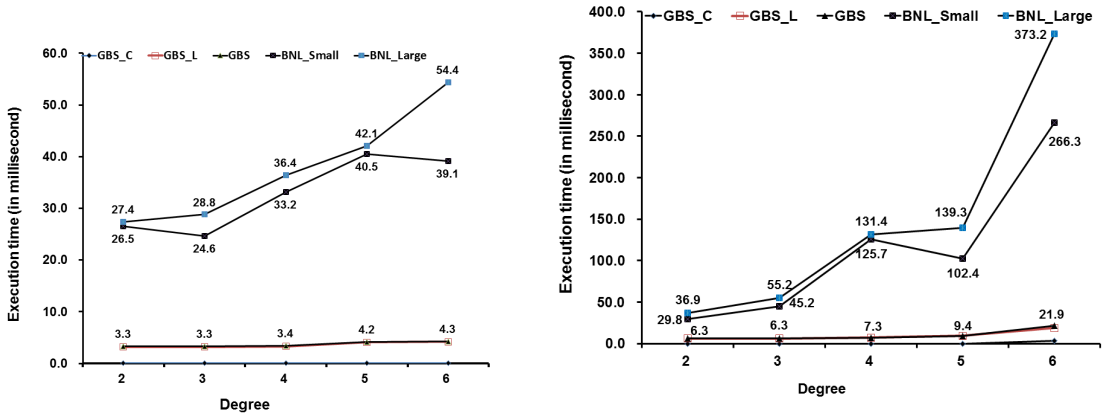
Fig. 8: The number of skyline points versus cardinality.

and degree m fixed at 6 as shown in Fig. 12(b). GBS shows 5-23 times faster than BNL and it is also more stable. When we vary the number of distinct values in each attribute from 2 to 6 in the second experiment, the increase in BNL's execution time with a large window size is 241.8 ms, which is significantly greater than that of GBS, 15.7ms.

More distinct attribute values tends to produce more equivalence classes as shown in Fig. 13, and will therefore have more skyline points as shown in Fig. 14. More skyline points imply more incomparable data points, and will therefore need more dataset scanning for BNL. The experiment points out that more distinct attribute values has much more effect on the performance of BNL than that of GBS.

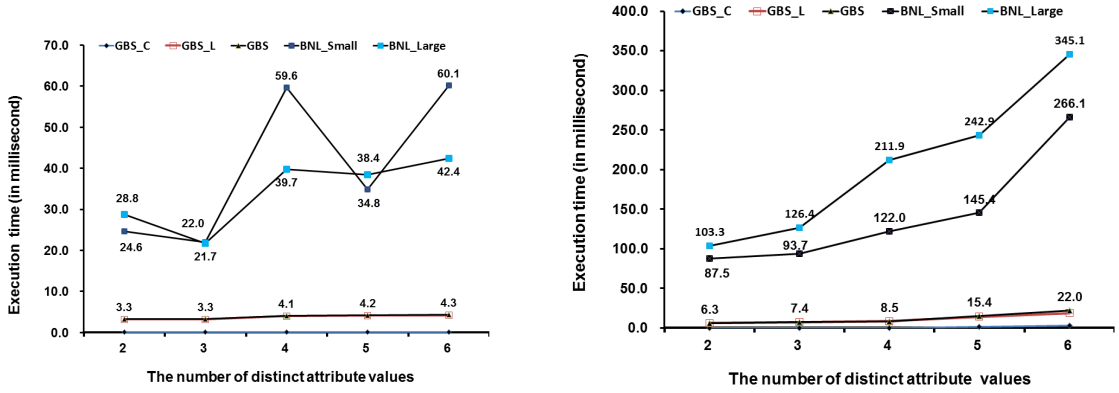
8 Conclusions

In this paper, we propose the static dominance graph to serve as a foundation for local pruning and navigation for skyline evaluation. The dominance graph, along with an array of stored distances between crossroads and data points, is pre-computed and reused until some data point's static attribute value is changed. Experimental evidence shows that navigating graph for determining skyline takes much less time than local pruning for finding candidates, and that GBS is more efficient than BNL by approximately half an order of magnitude on average. Varying the value of cardinality, degree, or the number of distinct attribute values causes less effect on the performance of GBS than that of BNL. In addition, GBS can return partial skyline points immediately after performing local pruning. The future direction of the research is location uncertainty that may affect our skyline evaluation.



(a) (b)

Fig. 9: Execution time versus degree with n=10K, d=2(a), n=16K,d=6(b).



(a) (b)

Fig. 12: Execution time versus the number of distinct attribute values with n=10K, m=3(a), n=16K,m=6(b).

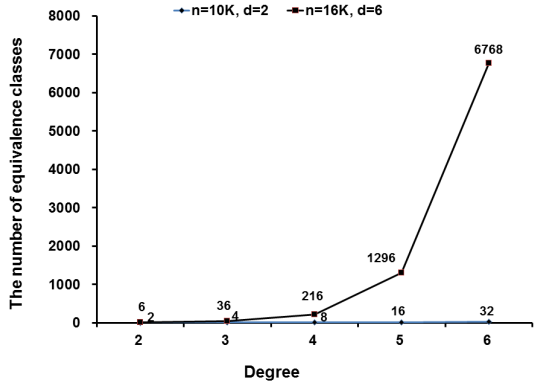


Fig. 10: The number of equivalence classes versus degree.

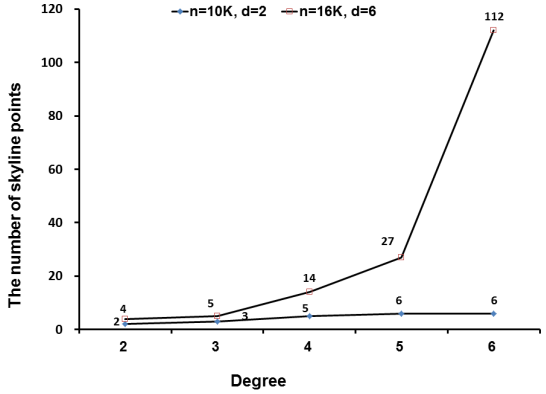


Fig. 11: The number of skyline points versus degree.

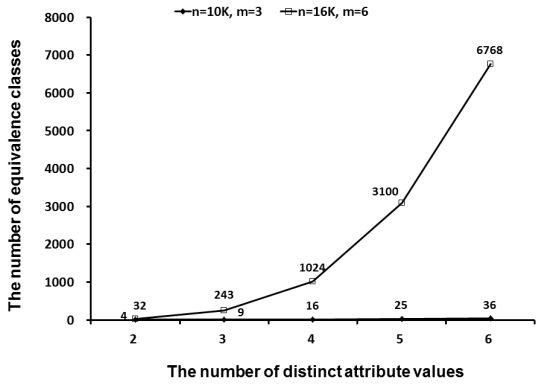


Fig. 13: The number of equivalence classes versus the number of distinct attribute values.

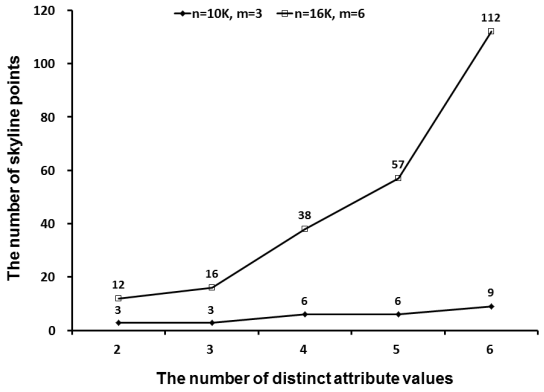


Fig. 14: The number of skyline points versus the number of distinct attribute values.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, Proc. ACM SIGMOD 322-331 (1990).
[2] S. Börzsönyi, D. Kossmann, K. Stocker, Proc. 17th Int'l Conf. Data Eng. (ICDE), 421-430 (2001).
[3] C.-Y. Chan, P.-K. Eng, K.-L.Tan, Proc. ACM SIGMOD, 203-214 (2005).
[4] L. Chen and X. Lian, IEEE Trans. Knowledge and Data Eng. **21**, 351-365 (2009).
[5] R. Cheng, D. V. Kalashnikov, S. Prabhakar, IEEE Trans. Knowledge and Data Eng. **16**, 1112-1127 (2004).
[6] J. Chomicki, P. Godfrey, J. Gryz, D. Liang, Proc. 19th Int'l Conf. Data Eng. (ICDE), 717-719 (2003).
[7] J. Chomicki, P. Godfrey, J. Gryz, D. Liang, Proc. Int'l Inf. Sys. Conf. (IIS), 595-604 (2005).
[8] K. Deng, X. Zhou, H. T. Shen, Proc. 23rd Int'l Conf. Data Eng. (ICDE), 796-805 (2007).
[9] R. H. Güting and M. Schneider, Moving Objects Databases, Morgan Kaufmann Publishers, San Francisco, CA, 2005.
[10] A. Guttman, Proc. ACM SIGMOD, 47-57 (1984).
[11] Z. Huang, H. Lu, B. C. Ooi, A. K. H. Tung, IEEE Trans. Knowledge and Data Eng. **18**, 1645-1658 (2006).
[12] D. Kossmann, F. Ramsak, S. Rost, Proc. 28th int'l Conf. Very Large Data Bases (VLDB), 275-286 (2002).
[13] H. T. Kung, F. Luccio, F. P. Preparata, Journal of the ACM **22**, 469-476 (1975).
[14] M.-W. Lee and S.-W. Hwang, Proc. 25th Int'l Conf. Data Eng. (ICDE), 1568-1575 (2009).
[15] X. Lin, Y. Yuan, W. Wang, H. Lu, Proc. 21st Int'l Conf. Data Eng. (ICDE), 502-513 (2005).
[16] C. L. Liu, Elements of Discrete Mathematics, 2nd Edition, McGraw-Hill, Singapore, 1985.
[17] M. Morse, J. M. Patel, W. I. Grosky, Information Sciences **177**, 3411-3437 (2007).
[18] K. Mouratidis, S. Bakiras, D. Papadias, Proc. ACM SIGMOD, 635-646 (2006).
[19] D. Papadias, J. Zhang, N. Mamoulis, Y. Tao, Proc. 29th int'l Conf. Very Large Data Bases (VLDB), 802-813 (2003).
[20] D. Papadias, Y. Tao, G. Fu, B. Seeger, ACM Trans. Database Systems **30** 41-82 (2005).
[21] D. Pfoser and C. S. Jensen, Lecture Notes in Computer Science, **1651**, 111-131 (1999).
[22] F. P. Preparata and M. I. Shamos, Computational Geometry: An Introduction, Springer-Verlag, New York, 1985,
[23] W. Stallings, Wireless Communications and Networks, Pearson Education Inc., Upper Saddle River, NJ, 2005,
[24] K. -L.Tan, P.-K. Eng, B. C. Ooi, Proc. 27th int'l Conf. Very Large Data Bases (VLDB), 301-310 (2001).
[25] Y. Tao, and D. Papadias, IEEE Trans Knowledge and Data Eng. **18**, 377-391(2006).
[26] O. Wolfson, A. P. Sistla, S. Chamberlain, Y. Yesha, Distributed and Parallel Databases **7**, 257-287 (1999).
[27] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, Q. Zhang, Proc. 31st Int'l Conf. Very Large Data Bases (VLDB), 241-252 (2005).
[28] B. Zheng, K. C. K. Lee, W.-C. Lee, The Ninth International Conference on Mobile Data Management, 148-155 (2008).
[29] L. Zou, and L. Chen, Proc. 24th Int'l Conf. Data Eng. (ICDE), 536-545 (2008).

Mei-Tzu Wang. received her B.S. degree in mathematics from National Cheng-Kung University in 1977, her M.S. degree in information engineering from Tamkang University in 1980, and her Ph.D. degree in management science from Tamkang University in 1993. She was an instructor in the Department of Computer Science, Soochow University from 1981 to 1987. Currently, she is an associate professor of the department of information management, Chinese Culture University. Her research interests include moving objects database, network security, and information theory.