Multi-source Skyline Query Processing in Road Networks

Ke Deng Xiaofang Zhou Heng Tao Shen
University of Queensland
{dengke, zxf, shenht}@itee.uq.edu.au

Abstract

Skyline query processing has been investigated extensively in recent years, mostly for only one query reference point. An example of a single-source skyline query is to find hotels which are cheap and close to the beach (an absolute query), or close to a user-given location (a relatively query). A multi-source skyline query considers several query points at the same time (e.g., to find hotels which are cheap and close to the University, the Botanic Garden and the China Town). In this paper, we consider the problem of efficient multi-source skyline query processing in road networks. It is not only the first effort to consider multi-source skyline query in road networks but also the first effort to process the relative skyline queries where the network distance between two locations needs to be computed on-the-fly. Three different query processing algorithms are proposed and evaluated in this paper. The Lower Bound Constraint algorithm (LBC) is proven to be an instance optimal algorithm. Extensive experiments using large real road network datasets demonstrate that LBC is four times more efficient than a straightforward algorithm.

1 Introduction

Skyline queries can be found in a wide spectrum of optimization applications [4, 5, 25, 21, 17, 19, 24]. Without loss of generality, let us consider optimization as minimization here. Given a set of multidimensional points D, a *skyline query* finds the skyline points from D, such that every point on the skyline is not 'dominated' by any other point in D (i.e., if a point p is on the skyline, there exists no data point p' in D such that p' is pair-wise smaller than p for the values in all dimensions). The skyline queries can be either *absolute* or *relative*, where 'absolute' means that minimization is based on the static attribute values of data points in D, while 'relative' means that the difference between a data point in D and a user-given query point needs to be computed for minimization. Relative skyline query is also known as *dynamic skyline query* [21].

An important category of skyline query applications is related to Geographical Information Systems, to support decision making in the areas such as intelligent transport systems, urban planning, environmental applications, location-based services, mobile workforce management, and military and utility deployment. Depending on different applications, the distance between two points a and b can be approximated using their Euclidean distance, or computed using the spatial network distance or the surface distance if the distance between a and b implies that an object needs to physically move from a to b along a spatial network (e.g., roads or water systems) or on/near a terrain surface. The case of using the Euclidean distance is called *constraint-free* as the distance between any two points can be calculated by only using the coordinates of the two points. The case where an environment dataset (e.g., a road network) needs to be used for distance calculation is called *constraint-based*. These two types of queries use quite different query processing strategies. For a constraint-free query the key to efficient query processing is to reduce the number of data points to be accessed (i.e., to minimize the portion of D accessed when answering a query). For a constraint-based query, however, one additional and often more critical goal is to minimize the portion of the environment data to be accessed and the cost of network distance calculation. This is because the environment data is typically much larger and much more complex than D and network distance calculation is typically done by using a costly shortest path algorithm. While skyline query processing in a constraint-free space has been well studied [4, 5, 25, 21, 17, 19, 24], to the best of our knowledge, this paper is the first effort on relative skyline query processing in a constrained space.

Existing work on skyline query processing mainly consider the case of one query reference point only (e.g., an absolute query in relation to the beach, or a relative query in relation to a user-given location) [4, 5, 25, 21, 17, 19]. There are many applications that demand to consider multiple query points at the same time (e.g., to find hotels which are cheap and close to the University, the Botanic Garden and the China Town). While no previous work has been reported in the literature about multi-source skyline queries in road networks, some effort for simpler versions of this problem exist. The problem of multi-source nearest neighbor query processing, known as *aggregate nearest neighbor query processing*, has been studied in Euclidean space [20] and in road networks [26]. One example of such a query is to find a meeting place or a restaurant which minimizes the total traveling distance for a group of people in different locations. The problem of multi-source skyline query processing is investigated in Euclidean space recently [24].

In this paper, we focus on multi-source relative skyline queries in road networks. It is the first effort to process multi-source skyline queries in a constrained space. It is also the first attempt to process relative skyline queries that require computing network distances on-the-fly. Three algorithms, the Collaborative Expansion algorithm (CE),

the Euclidean Distance Constraint algorithm (EDC) and the Lower Bound Constraint algorithm (LBC), are proposed. CE is a straightforward algorithm using an underlying paradigm that identifies network skyline points by expanding the search space centered around each query point incrementally. EDC is an approach to control the search directions for network skyline points by using Euclidean skyline points as a guide. LBC is based on network nearest neighbor algorithms and uses a novel concept of *path distance lower bound* to minimize the cost of network distance computation. We prove that LBC is an instance optimal algorithm [9]. Extensive experiments using large real road network datasets demonstrate that LBC is more than four times more efficient than CE.

The rest of the paper is organized as follows: Section 2 reviews the related work and Section 3 reviews network shortest path algorithms and road network storage methods. Three novel algorithms are proposed in Section 4. Efficiency of these algorithms are analyzed in Section 5, and an empirical performance study is reported in Section 6. We conclude this paper in Section 7.

2 Related Work

Skyline query processing has been studied extensively in recent years [4, 5, 25, 21, 17, 19, 24]. Borzsonyi et al. [4] propose the Block-Nested-Loops algorithm (BNL) and the Extended Divided-and-Conquer algorithm (DC). Both BNL and DC need to process the entire object set before reporting skyline data. To progressively report skyline points, the Sort-Filter-Skyline algorithm (SFS) [5] improves BNL by pre-sorting the entire dataset according a monotonic preference function. In the ascending order of the score, each object is taken to compare with the dominant set. If an object cannot be dominated, it is reported directly as a skyline point because there is no other object with a lower score can dominate it. Tan et al.[25] propose a bitmap-based method which transforms each object to bit vectors. In each dimension, the value is represented by the same number of '1'. However, bitmaps can be very large for large values. In addition, it cannot guarantee a good initial response time. Another approach proposed by [25] transforms high dimensional points into a single dimensional space where objects are clustered and indexed using a B^+-tree such that a skyline point can be determined without examining the rest of the object set not accessed yet. Both the bitmap and B^+-tree methods cannot report skyline points in an order according to user's preference [21].

Kossmann et al. [17] propose an online skyline query processing method which can progressively report the skyline points in an order according to user's preference. This method is based on an observation that for a set of objects the 1^{st} nearest neighbor (NN) to the query point is always a skyline point. When a skyline point is found, the data space is split at that point into one dominated subspace and several independent non-determined subspaces. The objects in the dominated subspace is pruned, and the objects in each non-determined subspace form a to-do list. Similarly, the 1^{st}NN in each to-do list is a new skyline point and the subspace is recursively split until all skyline points are reported. However, one object may be processed several times until it is determined (i.e. dominated or dominant), and in a high dimensional space, duplicate skyline points may be reported from different to-do lists. To remedy this problem, Papadias et al. [21] propose an R-tree based algorithm, Brand and Bound Skyline (BBS), which retrieves skyline points by browsing the R-tree based on the best-first strategy. BBS only visits the intermediate nodes not dominated by any determined skyline points. This method only processes the same node once, thus has a more efficient memory consumption than the method in [17]. Another study by Lin et al. [19] process a skyline point query against the most recent N elements in a data stream.

All the methods mentioned above are designed for single-source skyline query processing in Euclidean space. In Euclidean space, it is straightforward to extend those algorithms that can process relative skyline queries to process multi-source queries, as the distances to multiple sources can easily be computed using the same algorithm ([21] gives a classification of skyline algorithms according to whether they can process relative queries or not). These algorithms, however, are not suitable for processing relative nor multi-source skyline queries in a constraint-based space (e.g., a road network). This is because that, as mentioned before, the optimization focus in Euclidean space is on minimizing the amount of D data to be accessed, while in a constrained space the focus is shifted to minimizing the amount of environment data used.

Multi-source NN query processing has attracted some recent attention for Euclidean space (e.g., the group NN query, GNN[20]), and for road networks (e.g., the aggregated NN query, ANN[26]). Both group NN and aggregated NN queries find the k objects with the minimum aggregated credit, such as the minimum total distance to a group of query points. In [20], several R-tree based algorithms are developed to progressively search for the GNN results in the ascending order of the aggregated credit. This work is used in [26] in the Incremental Euclidean Restriction algorithm (IER) as the first step to find the ANN points in road networks. Multi-source skyline query in Euclidean space is recently studied by [24]. In this work, the query points are approximated by a convex hull and the data objects are indexed using a Voronoi diagram in order to reduce data access cost related to both the query points and the data objects. However, their methods are based on the Euclidean distance and cannot be applied to where network distances are needed.

Balke et al. [3] propose a unified method to process the skyline query and NN query in Euclidean space. This method needs to pre-sort all objects by each attribute, thus it is not suitable for relative queries in road networks.

3 Network Distance and Disk Storage Scheme

A road network can be modeled as a graph $G = (E, V)$, where V is a set of nodes corresponding to road junctions and E is a set of (non-directional) edges between two nodes

in V corresponding to road segments. An edge can be a straight line or a polyline. If there is an edge in E linking two nodes in V, these two nodes are adjacent to each other. Let $Adj(v)$ denote all the adjacent nodes of $v \in V$. Let $d(v, v')$ be the distance along a path between two points v and v' (i.e., the total length of the edges along a network path between the two nodes). If there is no path connecting v and v', $d(v, v') = \infty$. The network distance between v and v', denoted as $d_N(v, v')$, is defined using the network shortest path between the two nodes. In addition, we use $d_E(v, v')$ to denote their Euclidean distance.

Two representative network shortest path algorithms are Dijkstra's algorithm [8] and the A* algorithm [18]. They both propagates a search "wavefront" from a source node v_s until a destination node v_d is reached. A heap H is used to keep all the nodes on the wavefront. The initial step of Dijkstra's algorithm is to put every node $v \in Adj(v_s)$, together with the distance $d(v_s, v)$ (set to the length of the edge linking v_s and v), into H. Then, the algorithm iterates an expansion process by replacing a node $v \in H$, where $d(v_s, v)$ is the minimum for all nodes in H, by all the nodes in $Adj(v)$. For each node $v' \in Adj(v)$, $d(v_s, v')$ is set to $d(v_s, v) + d(v, v')$ if v' is not yet in H or (in case v' in H) the existing estimate of $d(v_s, v')$ is larger than $d(v_s, v) + d(v, v')$. This process terminates when v_d is selected from H to expand (and $d_N(v_s, v_d) = d(v_s, v_d)$). Dijkstra's algorithm can compute the shortest pathes from a source node to multiple destination nodes.

If there is only one destination, the wavefront expansion process can be optimized towards the direction of the destination node. This is the key idea of A*. Instead of selecting $v \in H$ with the minimum $d(v_s, v)$, a node $v' \in H$ with the minimum $d(v_s, v') + d_E(v', v_d)$ is selected to expand. That is, when selecting a node v', not only the computed network distance from v_s to v' is considered, the Euclidean distance from v' to v_d is also used as a directional guide. For any node $v \in H$, $d(v_s, v) + d_E(v, v_d)$ is called the *distance lower bound* of v from v_s to v_d (denoted as $lb(v, v_s, v_d)$, or $v.lb$ when not causing ambiguity). Clearly, any valid network path from v_s to v_d via v cannot be shorter than $v.lb$. One important property of A* is that it can find the network shortest pathes to multiple destinations by traversing the network only once. To do that, besides maintaining the previous state of H, all intermediate network nodes need to be kept with their network distances to the source [26].

Both Dijkstra's algorithm and A* compute the shortest path on-the-fly. There also exists several efficient algorithms that pre-compute the network distances between all pairs of nodes in order to avoid on-the-fly computation. One approach is to develop a transitive closure of G, at the cost of storage overhead. This type of approach is not suitable for large road networks, as they typically require $O(|V|^2)$ disk space [6]. Another approach divides a large graph into smaller subgraphs and organizes them in a hierarchical fashion [11, 16]. For each subgraph, the border nodes are the entrances and only the distances between these border nodes are pre-computed. Therefore, while computing the network distance, intermediate nodes inside the subgraph are skipped over. However, our problem often requires to compute network distances from one source to several destinations. Thus, it is desirable to keep the distance information for the intermediate nodes, as one can see later in this paper. A recent study [13] stores and indexes the network distances between every network node and data object with different level of accuracy according to the distance between them. The drawback of this approach is the high maintenance cost when the network is updated.

The shortest network distance from any object located on an edge to a source node can be directly derived if the network distance for the both ends of the edge to the source node are known. In [22], the network model and the data object set are independently stored. Such a storage scheme supports data independency for these two types of data. However, the operation of online location mapping for each object on the network is an expensive geometric computation. Another storage scheme [26] avoids the cost of online mapping by making the network and the object set linked with each other. One disadvantage of this scheme is that a network model is hard coded with a specific data object set. In this paper, we use a middle layer by partially materializing the mapping between two datasets. If an object p is on a network edge e between two adjacent nodes v, v', the distances $d(v, p)$ and $d(v', p)$ are pre-computed, and the id of e is stored in the middle layer with the id of p and the two pre-computed distances. This middle layer can be indexed using a B^+-tree on edge ids. For example, in either Dijkstra's algorithm or the A* algorithm, an edge can be efficiently checked to see if there are any data objects on it from the middle layer using the B^+-tree index when the edge is visited by the wavefront.

4 Our Approaches

In this section, we propose three multi-source road network skyline query processing algorithms, with different strategies to minimize the cost of network distance computation. $Q = \{q_i | i = 1..n\}$ and $D = \{p_i | i = 1..m\}$ denote the sets of query points and data objects respectively.

4.1 Collaborative Expansion Algorithm (CE)

The basic idea of this algorithm is to find the next nearest neighbor based on the network distance alternatively to each query point using Dijkstra's shortest path algorithm. That is, the search space of each query point, defined as a circle around the query point, is expanded in a collaborative way until all skyline data objects are found. When an object p is identified as the next nearest neighbor of a query point q, it is said that p is visited by q (and the network distance from q to p is computed at that time). In such a way, the objects around a query point is visited in the ascending order according to their network distance to this query point.

Conceptually, there are two phases in this algorithm. The first phase, called the *filtering* phase, ends when the first object visited by all query points is found. During this phase, all nearest neighbor objects encountered (from all query points) are considered as candidates of the network skyline

points and are stored in a candidate set C. When this phase terminates (i.e., an object p is visited by all query points), for any object p' not in C, it is clear that for every query point q, $d_N(p,q) \leq d_N(p',q)$. In other words, all points not in C when this phase terminates are dominated by p. Notice that p is a skyline point. This is obvious: let the last query point to visit p be q, then there exists no other objects which are closer to all other query points as well as to q.

In the second phase (called the *refinement* phase), the points in C will be checked, using the same process of alternative expansion of the search space from each query point to find their next nearest neighbor. Following the same observation above, the next object encountered is a skyline point as long as it is not dominated by the skyline points that are already identified. Notice that the new objects encountered during this phase (i.e., not already in C) are simply discarded. Assume now p is identified as a skyline point by CE. Let $C(p,q) \subseteq C$, $q \in Q$, be the set of objects that are visited by q after q visits p. Then, the candidates in $\cap_{q \in Q} C(p,q)$ can be safely pruned, as they are dominated by p.

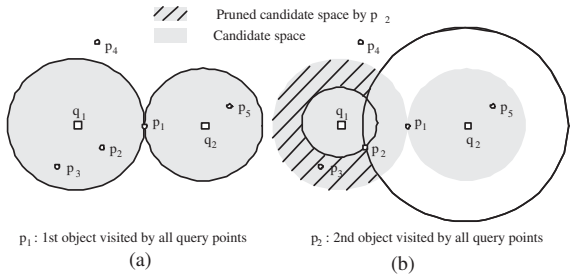

Figure 1. An Example for Algorithm CE.

Figure 1 is an example for two query points (q_1, q_2) and five objects (p_1 to p_5) (for presentation simplicity, Euclidean distances are used to illustrate the idea). p_1 is the first object visited by all query points, thus is the first skyline point. All the points within the two circles in Figure 1(a), $\{p_2, p_3, p_5\}$, are visited before p_1 is visited by both q_1 and q_2, thus are included in C. Note that p_4 cannot be a skyline point, thus can be safely pruned. During the refinement phase, p_2 is found as the next object in C visited by all query points (shown in Figure 1(b)). p_2 is examined by comparing with all previously computed skyline points. In this example, as p_2 is not dominated by p_1, it is reported as the next skyline point. All objects in C which are dominated by p_2 are pruned from C, as shown in the shaded area in Figure 1(b). In this case, p_3 is pruned from C, but p_5 remains to be checked (and later be identified as a skyline point when q_1 visits it).

4.2 Euclidean Distance Constraint Algorithm (EDC)

CE is a straightforward algorithm for multi-source network skyline query processing. One major problem with this algorithm is that the search space of each query point expands towards all directions. This can include too many candidate objects and cause unnecessary network distance computation. To control expansion directions, the multi-source skyline points in Euclidean space can be retrieved first. Then the network distances to each query point from these Euclidean skyline points are computed by using the A* algorithm, which expands towards those known Euclidean skyline points. Such a strategy that performs preprocessing in Euclidean space is common for network query processing [22, 26].

Given a set of query points Q and a set of data objects in D in a road network, these points can be considered both in Euclidean space (if the distances among a pair of points is measured using their Euclidean distance), or in the network space (when the distance is computed from the shortest network path). The basic idea of the EDC algorithm, consisting of the following five steps, is to explore this space duality.

- *Step 1 - Euclidean space multi-source skyline query processing*: The skyline points for all query points in Q in Euclidean space are retrieved into candidate set C. We will discuss our Euclidean space multi-source skyline query processing algorithm later.
- *Step 2 - network distance calculation*: The network distances between each query point in Q and each point in C are computed using the A* algorithm (i.e., the network distance are computed by expanding the search space from all query points towards the points in C using the directional expansion heuristics of A*). The intermediate network distances computed during this step are kept for the use in step 4.
- *Step 3 - Euclidean space window query processing*: At the end of step 2, each point $p \in C$ can be represented as a n dimensional point where the i^{th} attribute is the network distance from p to q_i. These network distances shift p in Euclidean space into a new position $\overline{p}$ in the same space (p_1 in Figure 2 (a) is shifted to $\overline{p}_1$). $\overline{p}$ and the origin of the space o form a hypercube, and only the data objects in Euclidean space within this hypercube can possibly dominate $\overline{p}$. Therefore, in this step, a new (complex) hypercube R is formed by the union of the hypercubes defined by $(o, \overline{p})$ for each $p \in C$. Then, all data objects within R in Euclidean space are retrieved and added into set C. Now it is easy to see that any data objects not in C cannot be part of the network skyline.
- *Step 4 - network distance calculation*: For all the points in C, their network distances to all query points will be calculated (using A* as in step 2). Note that A* records all intermediate distances computed, thus a significant portion of network distance calculations in step 2 are reused in this step.
- *Step 5 - network skyline points reporting*: By pair-wise comparing all objects in C according to their network distances to query points, the network skyline points can be reported.

The correctness of EDC is easy to see. While an Euclidean skyline point may not be a network skyline point, all the data objects potentially dominating the shifted Euclidean skyline points are retrieved in step 3.

Now let us give an algorithm for multi-source skyline query processing in Euclidean space, based on a simple extension to the approach proposed in [21] for processing Euclidean single-source dynamic skyline queries. Following this approach that uses the R-tree as a representative of multidimensional indexing on data objects, we also give an R-tree based algorithm. Starting from the root of the R-tree, all accessed entries are kept in a heap ordered by their $mindist$. The $mindist$ of an object is the sum of its Euclidean distances to all query points, and the $mindist$ of an MBR (i.e. intermediate entry) is the sum of the minimum distances from each query point to this MBR. The $mindist$ values are computed on-the-fly. In the heap, the entry with minimum $mindist$ is replaced by its children nodes in the R-tree. If the entry to be replaced is a leaf node, the object in the entry is a skyline point thus recorded in S. When the next entry is replaced by its children, only the children which are not dominated by any S object is inserted into the heap. This process continues until the heap is empty. The objects in S are the multi-source skyline points in Euclidean space.

The EDC algorithm given above is essentially a batch skyline query algorithm - no network skyline points can be reported until step 5. It is, however, straightforward to modify EDC for it to report network skyline points incrementally. Let us use the example in Figure 2(a) to describe the idea. $p_1 ... p_5$ are objects in Euclidean space (the dynamic space as termed in [21]). p_1 is an Euclidean skyline point retrieved from the R-tree and kept in C. Then, the network distance from p_1 to all query points are computed so p_1 is shifted $\overline{p}_1$. The grey area defined by $\overline{p}_1$ and the origin in Figure 2(a) is used to fetch all data objects in that region, thus $p_2, ..., p_4$ are added into C. Their network distances to all query points are then computed, and these points are shifted to their new positions $\overline{p}_2, ..., \overline{p}_4$. Skyline points can then be determined by pair-wise comparing all objects in C ($\overline{p}_1, .., \overline{p}_4$). If an object dominates $\overline{p}_1$ and it cannot be dominated by other objects, it is a network skyline point. As shown in Figure 2(a), $\overline{p}_4$ is a network skyline point. $\overline{p}_4$ and all the objects dominated by $\overline{p}_4$ are removed from C (i.e., $\overline{p}_1, \overline{p}_3$). Any undetermined objects (i.e., $\overline{p}_2$) remain in C. Then, the next iteration starts by retrieving the next Euclidean skyline point. Notice that an entry is not inserted into the heap if it is within the region defined by $\overline{p}_1$. In Figure 2(b), p_5 is the next Euclidean skyline point and $\overline{p}_5$ is the shifted p_5. Any dominant candidates from the candidate space defined by $\overline{p}_5$ should compare with previously determined network skyline points ($\overline{p}_4$ so far) before determining whether they are network skyline points. If no more Euclidean skyline points can be retrieved, each of the undetermined candidates in C is a network skyline point and EDC terminates.

4.3 Lower-Bound Constraint Algorithm (LBC)

While EDC improves the strategy for search space expansion using the directional expansion property of the A* algorithm, it has two problems. First, using the shifted point

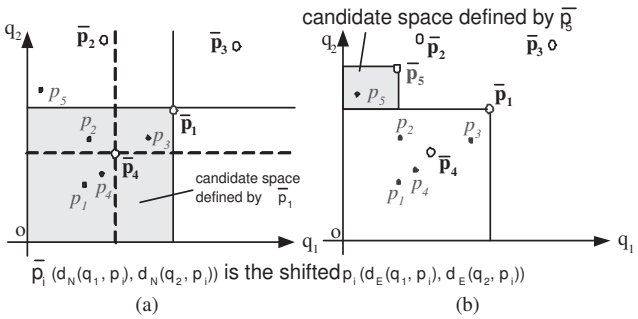

Figure 2. An Example for Algorithm EDC.

of an Euclidean skyline point can lead to a unnecessarily large search hypercube, thus may include too many objects in C in step 3 that can cause extraneous network distance computation later. For example, in Figure 2(a), $\overline{p}_1$ is dominated by $\overline{p}_4$, so it is desirable to use $\overline{p}_4$ instead of $\overline{p}_1$ in step 3. This is not possible in EDC because p_4 is not an Euclidean skyline point. Second, EDC computes the network distances for every candidate in C. Due to the very high cost of network distance computation, it is necessary to reduce the number of network distance calculations. These two observations motivate us to design a new algorithm that searches for candidates using a search hypercube defined by network skyline points only, and performs partial network distance calculations which can be terminated earlier if a candidate is found to be dominated by other points.

Let's define an important concept of *path distance lower bound*, which is the key for LBC to minimize the cost of network distance computation. Recall that A* propagates a search wavefront from a given source node v_s to a given destination node v_d. Every node on the wavefront is kept in a heap H with its distance lower bound lb, and if $\forall v', v \in H$, $v.lb \leq v'.lb$, v is selected as the expansion node. Clearly, $v.lb$ cannot exceed the length of the shortest path from v_s to v_d, even if the shortest path, which is not known yet, does not pass v. Therefore, $v.lb$ is a distance lower bound of the shortest path, and is called the *path distance lower bound*, denoted as plb. Let v' and v'' be the expansion node just before and after v is used as an expansion node, we have $v'.lb \leq v.lb \leq v''.lb$. In other words, the path distance lower bound increases during the process of search space expansion (the initial path distance lower bound is the Euclidean distance between v_s and v_d, and the final value when the shortest path is found is the actual network distance).

Now we describe the LBC algorithm. Recall that n is the number of query points in Q. Let S be the set of network skyline points, and C be the set of candidate network nearest neighbors (S and C are empty at the beginning). Choosing any query point q as the source query point, at a high level LBC iterates the following two steps until no more network nearest neighbors of q can be found.

- *Step 1 - find the next network nearest neighbor of* q: While any efficient network nearest neighbor algorithm can be used here, a more efficient method is proposed next to avoid computing network shortest distances for those points which are obviously cannot be

the next network skyline point.

- *Step 2 - network skyline points reporting*: Let p be the next network nearest neighbor of q found in step 1. The network distances from p to all other query points are computed (using an optimized algorithm described later). If p has not been dominated by any point in S, p is a network skyline point and thus added into S.

Our method to find the next network nearest neighbor point, used in step 1 above, consists of two sub-steps.

- Step 1.1 - find the next Euclidean space nearest neighbor p of q in the area that has not been dominated by any S point. LBC terminates if no such p can be found. This sub-step can be done using any efficient Euclidean space nearest neighbor algorithm, as long as the areas dominated by any S point are pruned during the search process. For example, if an R-tree index is used by the Euclidean space nearest neighbor algorithm, the whole sub-branch can be skipped if that branch is dominated by any point in S. In order to check if a point a (or an MBR b) is dominated by $s \in S$, a needs to be transformed into a n dimensional vector where the i^{th} attribute is the Euclidean distance between a and q_i (or the minimum Euclidean distance between b and q_i). Note that s is also a n dimensional vector but the i^{th} attribute is the network distance to q_i. Thus, a (or b) can be examined whether it is dominated by s. It is clear if a point represented by its Euclidean distances to the query points is already dominated by s, its counterpart in network distances will be dominated too, thus no need to compute these network distances.

- Step 1.2 - find the next network nearest neighbor. Let p be the next Euclidean nearest neighbor returned by step 1.1. The network distance between q and p needs to be computed using the A* algorithm, and p is added into C with $d_N(p, q)$. If there exists $c \in C$ such that $d_N(c, q) \leq d_E(p, q)$, the point currently in C with the minimum network distance to q is returned as the next network nearest neighbor point. Otherwise, go to step 1.1.

In step 2, a naive way of checking whether p is dominated by any point in S is to compute the network distances from p to all query points using, for example, the A* algorithm (note that only the network distance from p to q is known at the end of step 1). To reduce the number of network distance calculations, we can take advantage of the path distance lower bound. For each non-source query point q' (i.e., $q' \neq q$), a sorted listed $q'.L$ is created to store all known network skyline points, which are sorted by their network distances to q' in ascending order. When the next network nearest neighbor p is found, it is inserted into the sorted list of every non-source query point, but using only the Euclidean distance to these query points. If there exists a known network skyline point s which is in front of p in the sorted list for all non-source query points, p is dominated by s therefore can be removed from all the sorted list and discarded. This is obvious because all previously found network skyline points dominate p in terms of their network distances to the source query point.

If there does not exist any such $s \in S$ that dominates p in all the sorted lists, choose a non-source query point q' to expand to p if q''s current path distance lower bound to p is the minimum comparing to the path distance lower bounds from other non-source query points to p. Note that initially these path distance lower bounds are the Euclidean distances, and are incrementally expanded as we discussed before. This process pushes p down in these sorted lists, and p can be discarded if any $s \in S$ is found dominating p in all the sorted lists. Once the network distances from p to all non-source query points are computed and p is still not dominated by any $s \in S$, p is a network skyline point.

For simplicity of description, the case of one specified query point is discussed above. It must be pointed out that LBC can easily be extended to process multiple source query points, by selecting network nearest neighbor points from multiple query points alternatively. Clearly, the skyline points close to the source query points are reported earlier than others. Therefore, LBC can use different strategies for selecting the source query points to support the applications with user preferences.

Note that for CE, EDC as well as LBC, they can be directly extended to efficiently process more general cases where non-spatial attributes are also considered (e.g., hotel prices). The non-spatial attributes are static values. Therefore, they can be treated as normal attributes which have pre-computed "network distances" to all data objects.

5 Analysis

In this section, we compare candidate sizes and network access costs (i.e., the amount of network data accessed) for CE, EDC and LBC.

First, we compare candidate sizes. Denote the candidate set of algorithm A as $C(A)$. Both EDC and LBC retrieve an object in Euclidean space as a candidate if it is in the defined candidate space. EDC finds skyline points in Euclidean space and uses their shifted positions to define the candidate space as discussed in 4.2, while only network skyline points are used in LBC for this purpose. Since Euclidean skyline points are not necessary to be network skyline points, the candidate space of EDC cannot be smaller than that of LBC. That is, $C(LBC) \subseteq C(EDC)$. However, none of $C(EDC)$ and $C(LBC)$ has a definitive relationship with $C(CE)$ because both $C(EDC)$ and $C(LBC)$ are influenced by the average ratio between the network distance and Euclidean distance between all pairs of nodes in the network, but $C(CE)$ is not. We denote the average ratio as $\delta = d_N/d_E$. Generally, a larger δ will lead to a larger candidate space and in turn a larger candidate set.

Next, we compare the network access cost with a focus on analyzing the effect of path distance lower bound. Denote the network access cost of algorithm A as $N(A)$, which is the network nodes visited by A. As previously discussed, $C(LBC) \subseteq C(EDC)$ and both of them use the A* algorithm, therefore $N(LBC) \subseteq N(EDC)$ even without

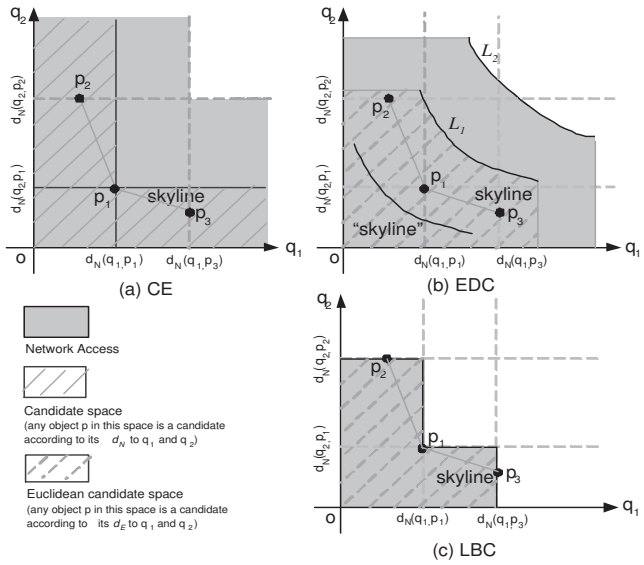

Figure 3. Candidate Space and Network Access Space.

considering the application of path distance lower bound in LBC. By considering the benefit gained from path distance lower bound in LBC, we have $N(LBC) \subseteq N(CE)$ even though $C(LBC)$ and $C(CE)$ have no definitive relationship, i.e. the network nodes accessed by CE is a superset of the network nodes accessed by LBC to process all candidates (even some of them are not candidates of CE). To prove this, suppose p is a skyline point (it must be a candidate in both LBC and CE). LBC uses the A* algorithm to compute the network distance between p to each query point q and the network node v will not be visited if $d(q,v) + d_E(v,p) > d_N(q,p)$. For the same q and p, Dijkstra's algorithm is used in CE by traversing a network region, denoted as R, such that any network node $v' \notin R$ has $d(q,v') > d_N(q,p)$. For such a node v', $d(q,v') + d_E(v',p) > d_N(q,p)$ must be held such that LBC will not visit v' either. That is, R is enough for LBC to process the skyline point. Suppose p' is a candidate in LBC but not in CE, p' must be dominated by some skyline point, say p. In LBC, p is already known before processing p'. Therefore, for p', LBC just computes the path distance lower bound as long as $d_N(q,p)$ for each query point q. That is, R is enough for LBC to process p'. Therefore, $N(LBC) \subseteq N(CE)$. However, there is still no definitive relationship between $N(EDC)$ and $N(CE)$ since EDC computes the network distances for all candidates and some candidates may be not the candidates of CE.

Figure 3 is an example to compare the candidate space and network access space for CE, EDC and LBC. This example is about two query point q_1, q_2 and the final solution includes three skyline points p_1, p_2, p_3. In CE, p_1 is the first object visited by all query points and the objects closer than p_1 to any query point are candidates. The network access space is the grey area. Since Dijkstra's algorithm is used, any network node closer than the outmost skyline point to each query point q will be visited by the wavefront started by q. In EDC, "$skyline$" consists of the skyline points in Euclidean space. L_1 is formed by their new positions after shifting. In Euclidean space, any object in the bottom-left of L_1 is a candidate. To find the network distances of all candidates (which can be estimated by δ in average), the network access space is indicated by line L_2. Obviously, the candidate size and network access of EDC are affected by δ of the network. In LBC, the candidate space is bounded by the skyline points and any object in Euclidean space in this space is a candidate. More interesting, the network access is bounded by the skyline points as well due to the application of path distance lower bound.

In the rest of this section, we prove that LBC is instance optimal for the network access cost over a class of algorithms. The concept of instance optimality [9] is different from the worst case or the average case complexity measures to specially consider the cases where an algorithm is worst-case optimal but not optimal for every instance of input. For example in binary search, for some instance of input, the result can be returned in one probe. But in the worst case, the binary search needs $\log_2 k$ probes for k items. It can be formally defined as follows: for a class of correct algorithms **A** and a class of valid input **D** to the algorithms, $cost(\mathcal{A}, \mathcal{D})$ represents the amount of a resource consumed by running algorithm $\mathcal{A} \in \mathbf{A}$ on input $\mathcal{D} \in \mathbf{D}$. An algorithm $\mathcal{B} \in \mathbf{A}$ is instance optimal over **A** and **D** if $cost(\mathcal{B}, \mathcal{D}) = \mathbf{O}(cost(\mathcal{A}, \mathcal{D}))$ for $\forall \mathcal{A} \in \mathbf{A}$ and $\forall \mathcal{D} \in \mathbf{D}$. As the network access cost is the major performance concern to our problem, it is defined as the cost of the algorithms.

Theorem 1: *Let $\boldsymbol{D}_o$ be the class of all possible networks and $\boldsymbol{D}_n$ be the class of all possible object datasets on any networks. Let **A** be the class of any correct algorithms processing multi-source skyline query in road networks. For each $\mathcal{A} \in \boldsymbol{A}$, each query point expands on network to search the network shortest path to any data object in $\boldsymbol{D}_o$ and there is no pre-computed distance information is used. LBC is instance optimal over **A**, $\boldsymbol{D}_o$ and $\boldsymbol{D}_n$, in terms of the network access cost.*

Proof: Since no pre-computed distance information is used, each $\mathcal{A} \in \mathbf{A}$ can only use Euclidean distance to approximate the network distance if necessary. Let's first prove LBC access optimal network data to process the skyline point. For all $\mathcal{A} \in \mathbf{A}$, if an object is a skyline point, the network distances to all query points need to be computed. Given a source node a and a destination node b, LBC uses A* algorithm to compute the network distance. If a frontier node v has $d(a,v) + d_E(v,b) < d_N(a,b)$, all adjacent nodes of v will be access sooner or later before returning $d_N(a,b)$. Assume an algorithm $\mathcal{A}'$ can find $d_N(a,b)$ without accessing one of v's adjacent nodes, say v'. Then v' and v''s neighbors are not known to $\mathcal{A}'$. In case v' is extremely close to the destination node b (i.e. $d_E(v',b) \approx 0$), $d_N(a,b) = v'.lb \approx v.lb$ and both v, v' are on the network shortest path. In this situation, $\mathcal{A}'$ is not correct since v' is ignored. Note that the class $\mathbf{D}_n$ includes all possible networks such that this case is a valid instance of $\mathbf{D}_n$. That is, the network data accessed by LBC cannot be further reduced.

Next, we prove that LBC is optimal to process each dominated object. Before processing each dominated object p',

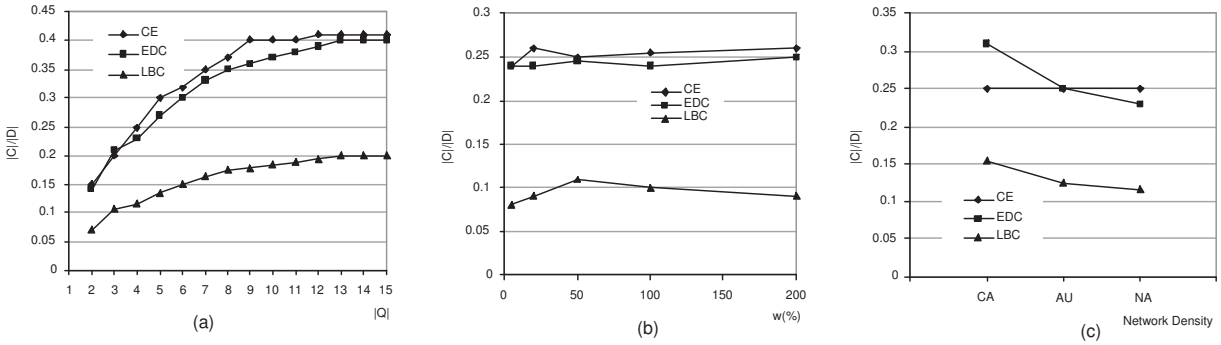

Figure 4. Candidate Ratio $|C|/|D|$ vs. (a)$|Q|$ (ω = 50%,NA) (b)Object Density ω($|Q|$=4,NA) (c)Network Density ($|Q|$=4, ω=50%).

since the dominating skyline point(s) are already known, LBC only finds the path distance lower bounds just long enough to prove p' is dominated. So, we need to prove, LBC access optimal network nodes among all $\mathcal{A} \in \mathbf{A}$ to compute a certain value of path distance lower bound. In LBC, given the source node a and the destination node b, the distance lower bound of next expansion node v is the path distance lower bound, i.e. $plb = v.lb$. If another node v' has $v'.lb < v.lb$, all v''s adjacent nodes are accessed by LBC before plb is returned (i.e. before expanding v) such that we are sure plb is not longer than the network distance between a and b. Assume an algorithm $\mathcal{A}'$ returns plb without accessing one of v''s adjacent nodes, say v''. In this situation, we are not sure $plb < d_N(a, b)$ is held in every case of $\mathbf{D}_o$ and $\mathbf{D}_n$. The reason is that $\mathcal{A}'$ has no knowledge of v'' and v'''s neighbors. If in case v'' is extremely close to the destination node b, $d_E(v'', b) \approx 0$ such that $d_N(a, b) = v''.lb \approx v'.lb < plb$. In this case, $\mathcal{A}'$ is not correct. That is, network access by LBC cannot be further pruned for processing dominated objects. In other words, LBC is instance optimal over $\forall \mathcal{A} \in \mathbf{A}$ and $\forall \mathcal{D} \in \mathbf{D}$ (including $\mathbf{D}_n$ and $\mathbf{D}_o$) for network access cost. □

6 Performance Evaluation

6.1 Experiment Setup

The experiments are conducted using a PC (ADM Athlon XP 2400+ CPU, 1.3GB memory). Three real world road networks are obtained from www.maproom.psu.edu/dcw. They are the road networks for California (CA: 3607 edges, 3044 nodes), Australia (AU: 30,289 edges, 23,269 nodes), and North America (NA: 103,042 edges, 86,318 nodes). The third one is merged from multiple originally separated road networks. These networks are unified into a $1km \times 1km$ region to represent different network densities. Similar to [22], the adjacent lists of the network nodes are clustered on the disk to minimize the I/O cost during network distance computation. The edges are indexed by an R-tree on edge MBRs. For efficient mapping between objects and the network, we implement the middle layer as described in Section 3. The disk page size is set to 4KB and a 1MB LRU buffer is used in all experiments. The data object set D consists of the points extracted uniformly from the edges by traversing the edge R-tree of the network. Thus, a dense road network in an area means more objects in the area. The size of D is a percentage of $|E|$ (the number of network edges), and the ratio $\omega = |D|/|E|$ is called the object density. Five ω: $\{5\%, 20\%, 50\%, 100\%, 200\%\}$ are tested. For a more accurate performance comparison, the query points ranging from 1 to 15 are selected within a relative small region (10%) of the network such that the maximum search region will not go beyond the given network. The objects are also indexed by an R-tree. In order to just traverse the network once, for both A* and Dijkstra's algorithm, the frontier nodes on the wavefront are maintained such that the expansion can continue from a previous state. In addition, as described in [26], when applying A*, each query point keeps a hash table to store the intermediate nodes visited, together with their network distances to the query point. The performance data reported in this section are the average of ten tests.

6.2 Candidate Size

Figure 4 (a)(b) illustrate the candidate size of each algorithm in network NA against $|Q|$ and ω. The candidate size is represented as a ratio $|C|/|D|$ where $|C|$ is the number of candidates. A higher candidate ratio indicates a larger candidate set, i.e. more objects in the object dataset are candidates. In (a), the candidate ratio increases in all algorithms when the number of query points increases from 2 to 15, with a slowing increase rate. In skyline query processing, the candidates are usually in the vicinity of each query point. If query points are close, their candidate spaces may overlap. Therefore, in a given region, if many query points already exist, the candidate space of a new query point is more likely to overlap with some existing candidate spaces such that fewer new candidates will be introduced. In (b), when object density ω changes from 5% to 200%, the candidate ratio changes little. This demonstrates that the candidate size changes proportionally to the change of object dataset size in a network. Comparing to EDC in (a)(b), LBC has a remarkably low candidate ratio for all settings. This is expected as $C(LBC) \subseteq C(EDC)$ (see Section 5). In addition, the experiment results show LBC usually has a much better filtering efficiency than CE although there does not exist definitive relation between $C(LBC)$ and $C(CE)$. In (a)(b), EDC is slightly more efficient than CE. But this efficiency disappears when the experiments are conducted in

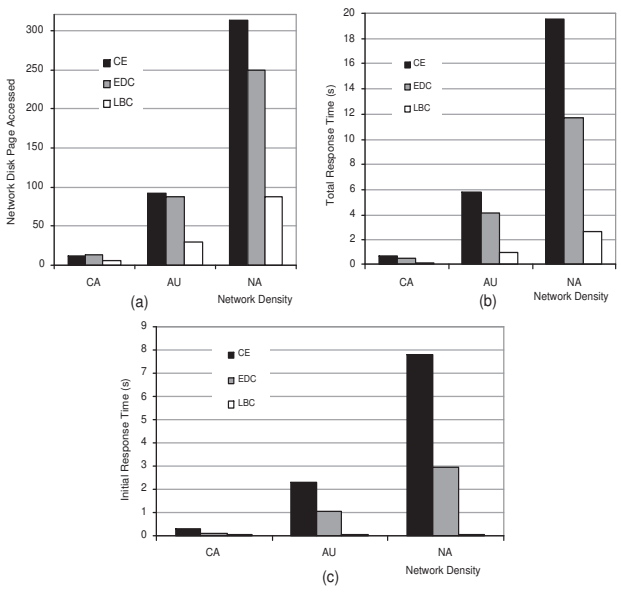

Figure 5. Performance vs. Network Density ($|Q|$=4, ω=50%).

other networks as shown in Figure 4(c). Especially, EDC is even worse than CE in network CA. The reason is that the network density decreases from NA to CA in Figure 4(c). In the case of low density (such as CA), δ (the average ratio of network distance and the Euclidean distance) is usually large such that EDC shows a worse filtering efficiency. We will discuss more about network density in the next experiment. On the contrary, CE keeps a reliable candidate size since CE is not affected by δ. With CA, LBC loses some efficiency due to the same reason as EDC, but it still has the best performance.

6.3 Effect of Network Density

Now we further examine how network density affects the performance in terms of disk access and response time. In a given region, if more roads are distributed such as in dataset NA, it is likely that there are more choices for path planning and the resultant network distance tends to decrease such that the difference between the Euclidean distance and network distance becomes narrower (i.e. δ decreases). In Figure 5(a)(b), the disk access and response time of all algorithms increase with the increase of network density. But the increase rate is different. While LBC increases slowly, CE increases very fast. The increase rate is influenced by both candidate size and network distance computation. We have shown the candidate size in Figure 4(c). Note that the difference between CE and LBC in Figure 4(c) becomes larger in Figure 5(a)(b). This is because LBC uses the path distance lower bound such that the network access is minimized to a just-enough region, while the network distance computation of CE has not been optimized to reduce the network access. Moreover, EDC also has a better performance comparing to CE in terms of disk access and total response time than in terms of candidate size. Especially, in the case of low network density (CA), the performance of EDC is quite close to that of CE in Figure 5(a)(b) but it is much worse in Figure 4(c). This can be explained by the use of different network shortest path searching algorithms. EDC uses A* while CE uses Dijkstra's algorithm. To compute the network distance for a same pair of objects, the network access cost using A* is smaller than that using Dijkstra's algorithm. Figure 5(c) is the experiment result of the initial response time. Clearly, in the case of high network density, more network data will be accessed before the first skyline point is returned. In particular, LBC returns the first skyline point immediately since the initial response only involves the source query point and its first network NN is a skyline point. In CE, although the first network nearest neighbor of each query point is a skyline point, its distances to other query points are unknown such that it is less useful. In this paper, we only report skyline points when their network distances to all query points have been calculated.

6.4 Effect of $|Q|$

Figure 6 (a)-(c) show the performance when $|Q|$ varies from 2-15. Since it is necessary to traverse the network for every query point independently, the overall cost (for both the disk access cost and the total response time) increases linearly for all the algorithms with the increase of $|Q|$. We observe that the increase trends of the disk access and total response time are similar. That confirms that I/O is the overwhelming factor to the overall performance. In Figure 6 (c), the initial response times for all algorithm are compared. LBC can return the first skyline point immediately at every setting of $|Q|$ since it only uses one query point. On the contrary, the initial response time of CE and EDC increases linearly since it needs to consider all query points.

6.5 Effect of ω

Finally, we examine the effect of object density in Figure 6 (d)-(f). Object density is set to be a percentage of the total network edges. From 5% to 200%, we observe that all algorithms have a trivial increase for the disk access cost and the total response time. This result illustrates that object density is not an important factor to the performance. The skyline points are usually in the vicinity of each query point. Therefore, the distribution of query points decides the network distances computation required (i.e. how many and how long). If the query points are distributed in a given region and each query point traverses the network once to several destinations, object density has little impact on the performance.

7 Conclusion

In this paper, three novel algorithms have been proposed for processing multi-source relative skyline queries in road networks. It is not only the first effort to process relative skyline queries in road networks, but also the first study on skyline queries by considering relative network distances to multiple query points at the same time. LBC is proven to be instance optimal in terms of the network search space over all algorithms where network distances are computed by expanding the searching region from query points without using pre-computed distance information. Our experiments confirmed that LBC has the best performance consistently

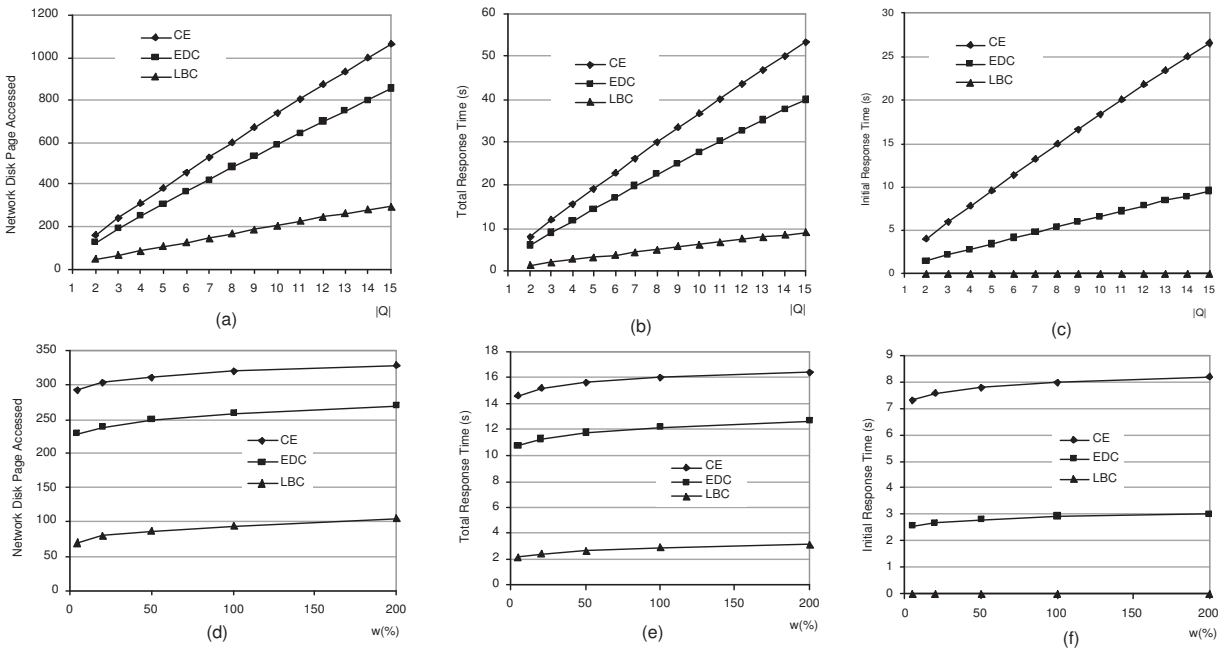

Figure 6. (a)-(c) Performance vs. $|Q|$ (ω=50%, NA), (d)-(f) Performance vs. ω ($|Q|$=4, NA).

for various test settings. The path distance lower bound approach, based on which LBC is designed, can be applied to benefit other types of road network queries where network distance comparison is needed.

Acknowledgment: The work reported in this paper is supported by grants DP0663272 and DP0666428 from the Australian Research Council.

References

[1] R. Agrawal, S. Dar, and H. Jagadish. Direct transitive closure algorithms: Design and performance evaluation. *TODS*, 15(3):427–458, 1990.

[2] R. Agrwal and H. V. Jagadish. Algorithms for searching massive graphs. *TKDE*, 6(2):225–238, 1994.

[3] W.-T. Balke and U. Güntzer. Multi-objective query processing for database system. *VLDB*, 2004.

[4] S. Borzsonyi, D. Kossmann, and K. Stoker. The skyline operator. *ICDE*, 2001.

[5] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with processing. *ICDE*, 2003.

[6] S. Dar and R. Ramakrishnan. A performance study of transitive closure algorithms. *SIGMOD*, 1994.

[7] K. Deng, X. Zhou, H. Shen, K. Xu, and X. Lin. Surface knn query processing. *ICDE*, 2006.

[8] E. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66:614–656, 2003.

[10] R. W. Floyd. Algorithm 97 (shortest path). *CACM*, 5(6):345, 1962.

[11] R. Goldman and N. Shivakumar. Proximity search in databases. *VLDB*, 1998.

[12] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[13] H. Hu, D. L. Lee, and V. Lee. Distance indexing on road networks. *VLDB*, 2006.

[14] Y. Huang, N. Jing, and E. Rundensteiner. Hierarchical path views: A model based on fragmentation and transportation road types. *Proc. 3rd ACM Workshop Geographic Information Systems*, pages 93–100, 1995.

[15] N. Jing, Y. Huang, and E. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *TKDE*, 10(3):409–432, 1998.

[16] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *TKDE*, 14(5):1029–1047, 2002.

[17] D. Kossmann, F. Ranmsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. *VLDB*, 2002.

[18] R. Kung, E. Hanson, Y. Ioannidis, T. Sellis, L. Shapiro, and M. Stonebraker. Heuristic search in data base system. *Proc. 1st International Workshop on Expert Database Systems*, 1986.

[19] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. *ICDE*, 2005.

[20] D. Papadias, Q. Shen, Y. Tao, and K. Mouratids. Group nearest neighbor queries. *ICDE*, 2004.

[21] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. *SIGMOD*, 2003.

[22] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. *VLDB*, 2003.

[23] C. Papadimitriou and M. Yannakakis. Multiobjective query optimization. *PODS*, 2001.

[24] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. *VLDB*, 2006.

[25] K.-L. Tan, P.-K. Eng, and B. Ooi. Efficient progressive skyline conputation. *VLDB*, 2001.

[26] M. Yiu, N. Mamoulis, and D. Papadias. Aggregate nearest neighbor queries in road networks. *TKDE*, 17(6):820–833, 2005.