Preference Queries in Large Multi-Cost Transportation Networks

Kyriakos Mouratidis [#1], Yimin Lin [#2], Man Lung Yiu [*3]

[#] *School of Information Systems, Singapore Management University*
80 Stamford Road, Singapore 178902
[1] kyriakos@smu.edu.sg
[2] yimin.lin.2007@phdis.smu.edu.sg

[*] *Department of Computing, Hong Kong Polytechnic University*
Hung Hom, Hong Kong
[3] csmlyiu@comp.polyu.edu.hk

***Abstract*— Research on spatial network databases has so far considered that there is a single cost value associated with each road segment of the network. In most real-world situations, however, there may exist multiple cost types involved in transportation decision making. For example, the different costs of a road segment could be its Euclidean length, the driving time, the walking time, possible toll fee, etc. The relative significance of these cost types may vary from user to user. In this paper we consider such *multi-cost transportation networks* (MCN), where each edge (road segment) is associated with multiple cost values. We formulate skyline and top-k queries in MCNs and design algorithms for their efficient processing. Our solutions have two important properties in preference-based querying; the skyline methods are *progressive* and the top-k ones are *incremental*. The performance of our techniques is evaluated with experiments on a real road network.**

I. Introduction

In most location based services, the users and the facilities (i.e., points of interest) lie in a road network. Since this network dictates the movement of users towards the facilities, the notion of distance between a user and a facility is defined as the total cost of the shortest path that connects them. The cost of each traversed road segment is defined as the length of the segment, the travel time required to drive (or walk) through it, etc. The traditional range and k nearest neighbor (kNN) queries have been extensively studied under this definition of proximity (based on network distance) and several storage schemes and algorithms have been proposed for their processing (e.g., [1], [2]).

All these methods consider that there is a single cost type of interest, i.e., *either* the path length, *or* the total travel time, *or* the summed toll fees, etc. In practice, however, these multiple cost types coexist and may collectively affect the decisions of users (i.e., their preference over the facilities). Furthermore, the relative importance of the costs is user-dependent; a user may be interested in minimizing the total travel time (to reach a facility), while another may be willing to accept a longer travel time in order to minimize the monetary cost of the journey (toll fees, fuel consumption). In such *multi-cost transportation networks* (MCN), the choice of a facility often affects multiple users or serves multiple purposes, each with different reachability requirements. Balancing these requirements necessitates computing a skyline [3] over the facilities, or finding the top-k [4] among them (in case different users or purposes served can be prioritized, e.g., assigned some significance weights). Preference queries in MCNs arise in a variety of logistics, location-allocation and spatial planning applications.

Consider a logistics scenario where various goods need to be transferred from the port to a warehouse. There are multiple locations where the warehouse can be built (or, there are multiple warehouses we can choose from). Some goods are sensitive and need to be transferred as fast as possible from the port to the warehouse (e.g., dairy products), i.e., the route chosen will minimize the total travel time. On the other hand, less sensitive products will be moved using the cheapest route (in terms of oil consumption, or total toll fees).

Figure 1 exemplifies this scenario, showing the port (query location) as solid point q, and the warehouse locations (facilities) as hollow points p_1, p_2. The rhombs correspond to toll gates, each charging 1 \$. For either of the two facilities p_1 and p_2, the fastest route from q is illustrated by solid arrows, while the cheapest route is represented by hollow ones. The numbers in parentheses next to p_1, p_2 correspond to these shortest paths, i.e., to the shortest travel time and the smallest monetary cost respectively. In this setting, if a warehouse (facility), say p_1, were both faster and cheaper to reach than another, say p_2, we would eliminate the latter from consideration; formally, we would say that p_1 *dominates* p_2. In our example, however, it is not possible to express a preference of a warehouse over the other, since one (p_2) is faster to reach while the other (p_1) is cheaper. Therefore, they should both be reported by our decision support system as possible facilities of choice. This motivates the definition and use of the *skyline* concept in MCNs.

The above setting also motivates another type of preference query, namely top-k processing in MCNs. If we had more information about the relative frequency of sensitive versus non-sensitive loads moved, we would possibly be able to balance the time and money criteria accordingly. For example, if 90% of the products transferred were sensitive, we could

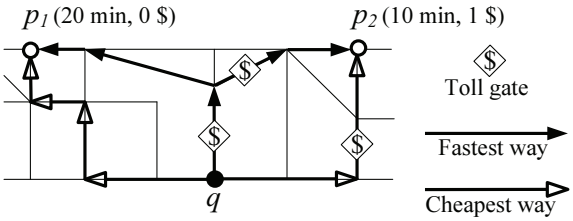


Fig. 1. Multi-cost network example

quantify the suitability of a warehouse p using aggregate cost function $f(p) = 0.9 \cdot c_t + 0.1 \cdot c_\$$, where c_t is the (normalized) time required by the fastest route from q to p, and $c_\$$ is the (normalized) monetary cost of the cheapest path between q and p. Since the lower $f(p)$ the better, our decision support system would report the facility that minimizes $f(p)$, or maybe (if more options are required) the k facilities with the lowest aggregate costs. This is an instance of a top-k query in an MCN.

In the above preference queries, the multiple criteria involved had to do with different *purposes* served by the facility (i.e., sensitive versus non-sensitive product storage). In other applications, the existence of multiple criteria is because different types of *users* will be served by the facility. As an example of this latter category consider the situation where a location must be chosen (from a set of available residential blocks) to build a housing facility for the students and instructors of a university. Clearly, the facility should be built as close to the university as possible. However, closeness is defined differently for different users, depending on their means of commuting. Specifically, some students/instructors are on foot while others drive from/to the university. Since the shortest path according to walking time may be different than that according to driving time[1], the suitability of a location depends on two different criteria. Furthermore, if pricing is also a consideration for people who drive, then the additional criterion of monetary cost needs to be taken into account.

Although this example is drawn from a different domain than the warehouse scenario, a decision would be based on similar MCN preference queries. For simplicity, consider that the two criteria involved in the current example are driving time and walking time. A facility dominates another if it is faster to reach both on foot and by car; the skyline would contain only the non-dominated locations. Assume now that we are given the percentages of walking and driving students/instructors, say 70% and 30% respectively. A meaningful aggregate cost function would be $f(p) = 0.7 \cdot c_w + 0.3 \cdot c_d$, where c_w and c_d are the shortest walking and driving times between the university q and building block p. In this case, the facility (or the facilities) of choice would be found by a top-k query.

[1]The shortest driving path is usually different from the shortest walking path, due to the existence of one-way streets, pedestrian-only streets, highways, etc. A real example where this difference is taken into account is the built-in "Maps" application of the iPhone (or similar features in other products) where a choice between walking and driving modes is required when the user requests for the shortest path between two locations.

Similar preference queries are also important in social networks. Consider for instance a network where the edges between individuals carry multiple weights, such as the degree of family relationship (if any), the frequency/length of calls between them, their spatial proximity (in terms of residential addresses), the amount of time spent in the same company/university during their career, their number of joint projects, etc. Skyline and top-k queries in such an MCN could find the people p that are closest related to a specific individual q according to the multiple affinity measures. Although social networks are not our main focus, our work directly applies to this case too.

Despite the prevalence of MCNs and the significance of the above preference queries, currently there exists no study on them neither in the spatial network nor in the operations research literature. In this paper we formally define skyline and top-k queries in MCNs, and design efficient algorithms for their processing. Our skyline methods are *progressive*, i.e., they output the first skyline facilities almost instantly and the result is populated gradually until the algorithm terminates. This property is typically useful in online systems [5], where it is essential that parts of the result become available to the application as soon as possible, without having to wait for the algorithm to terminate. On the other hand, our top-k techniques are *incremental*, i.e, having obtained the top-i facilities, the $(i+1)$-st can be retrieved without having to calculate the top-$(i+1)$ result from scratch. This is crucial in applications where k (the number of facilities to be retrieved) is not known in advance.

The rest of the paper is organized as follows. Section II surveys related work. Section III describes the setting and formalizes the problems we consider. Section IV presents our techniques for skyline processing in MCNs, while Section V extends our framework to top-k queries. Section VI experimentally evaluates our methods, and Section VII concludes the paper with directions for future work.

II. Related Work

In this section we review related work on preference queries, road networks, and a relevant operations research problem.

A. Skyline Processing

Consider a relation P with attributes $a_1, a_2, ..., a_d$. A tuple p dominates another p' if all the attributes of p are no larger than those of p', and at least one of them is smaller. The skyline of P contains those tuples that are not dominated by any other. [3] first considered skyline processing over disk-resident data, and proposed two methods; one based on the block nested loops and the other on the divide and conquer paradigm. The former is improved in [6] and [7] by topologically sorting P before processing. [8] extends this idea, and additionally may terminate without scanning the complete ordered input.

An index on P may accelerate processing (e.g., [5], [9], [10]). [10] is the most efficient method for indexed data. It utilizes an R-tree [11] on the d attributes of P to incrementally

retrieve the nearest neighbors (NN) to the origin of the data-space. The first NN is guaranteed to be in the skyline, and is used to prune the part of the space it dominates. Then, the next NN is retrieved in the remaining part of the space; it is also included in the skyline and used to further prune the search space. The process continues until no more NNs can be found in the non-dominated part of the space. [10] is I/O optimal, i.e., it accesses the minimal number of R-tree nodes.

None of the above methods apply to our case, because they assume that once a tuple is encountered, all its attributes are directly available (and can thus be used for dominance checks). This is not the case in MCN skyline, since acquiring the different costs of a facility is expensive, and also because knowing one of these costs does not imply that we can directly retrieve the remaining ones.

B. Top-k Processing

Consider the same relation P and a scoring function f over its attributes. A top-k query retrieves the k tuples $p \in P$ with the highest $f(p)$ scores. [4] proposes the *threshold algorithm* (TA), which is suitable for monotone scoring functions f. It assumes that the tuples are organized in d sorted lists, one for each attribute, ordered in descending preference order according to the specific attribute. TA pops the top element of each list in a round-robin fashion and computes its score. It terminates when the k-th best tuple found so far has no smaller score than threshold $T = f(t_1, t_2, ..., t_d)$, where t_i is the key of the next element in the i-th list. The no-random-access flavor of TA also uses T but it does not compute $f(p)$ directly upon encountering p in a list; instead, it does so only when p has been popped by the remaining lists. Our top-k algorithms are related to this method. However, in our case the costs (playing the role of attributes) are not available or sorted in advance, and can only be retrieved by network operations. Also, we do not maintain any global threshold T; our termination condition relies purely on the network-based properties of the search. [12] extends TA to skyline processing; [12] does not apply to MCN skylines for the reasons explained above for TA, but also because it requires random accesses to the tuple attributes.

Assuming a monotone f and the existence of an R-tree on P, [13] describes an I/O optimal top-k algorithm. It visits the R-tree nodes in descending order of their *maxscore*; the *maxscore* of a node equals the highest among the scores of its MBR corners (which is an upper bound of the score of any tuple under this node). The search terminates when the k-th best tuple encountered has score no smaller than the *maxscore* of the next R-tree node to be visited. This method is only applicable to the vector space model and assumes that the attributes (and thus the score) of a tuple can be directly retrieved upon encountering the tuple.

C. Road Network Databases

A road network is a graph $G = \{V, E, W\}$, where V contains the network nodes (road intersections), E includes the edges (road segments), and W associates a weight (e.g., Euclidean length, travel time, etc) to each edge in E. The shortest path query is the most common in this setting; it computes the edge sequence between a source v_s and a destination node v_d with the smallest sum of edge weights (this sum is called the *network distance* between these nodes). Dijkstra's algorithm [14] is the most widely used algorithm for this query due to its generality and efficiency. It pushes in a min-heap the adjacent nodes of v_s, with keys equal to the weights of the corresponding edges. It then iteratively pops the head of the heap, and en-heaps its adjacent unvisited nodes. The process continues until v_d is popped; the shortest path is formed by tracing back v_d's predecessors in the traversal all the way until v_s.

The A* algorithm [15] improves Dijkstra's performance, provided that a lower bound $LB(v, v_d)$ is available for the network distance of every node v from v_d. For example, if network distance is defined as the total (Euclidean) length of the shortest path between two nodes, then the Euclidean distance can be used as LB. A* proceeds exactly like Dijkstra's algorithm, but uses as sorting key of a node v its network distance from v_s plus $LB(v, v_d)$. We focus on Dijkstra's algorithm, because we target generic cost types for which lower bounds typically do not exist. For large graphs, shortest path search may be accelerated by pre-computing and materializing all or some of the shortest paths/distances among network nodes [16], [17], [18], [19].

[1] proposes *network expansion* (NE), an adaptation of Dijkstra's algorithm for disk-resident data. Given a query location q and a set of facilities P in the network, NE computes the k nearest facilities (NNs) to q in terms of network distance. Once an edge's end-node is popped from the search heap, the facilities on this edge are read from the disk and pushed into the heap. The first popped facility is the first NN, the second is the second NN, and so on. NE is incremental, i.e., the next NN can be iteratively retrieved by keeping popping the heap. NE may also be applied to range queries.

[20] proposes the *multi-source skyline query*, extending the concept of the spatial skyline [21] to road networks. Given a set of d query points, each facility p is mapped to a d-dimensional point, where the i-th dimension corresponds to p's network distance from the i-th query point. The result of the query is the skyline in this d-dimensional space. The main idea is to search the network concurrently for all query points and to exploit Euclidean bounds to guide the search. [20] does not apply to MCN skyline, because it solves a different problem: it considers a single cost type and is meaningful when a *query set* is defined (as opposed to a single query point). Also, in our case no Euclidean bounds can be used, because we consider general cost types for which no such bounds are available.

D. Multi-Criteria Pareto Path Computation

MCNs have been considered in the field of operations research for *multi-criteria Pareto path computation* (MCPP) over memory-resident data. Given a source v_s and a destination node v_d in the MCN, the problem is to compute the skyline among all possible paths between v_s and v_d (in terms of the d MCN cost types). A path dominates another if none of its d

costs is larger. Most approaches build on Dijkstra's algorithm, and are categorized into *label setting* [22], [23] and *label correcting* [24], [25], [26] algorithms. If lower bounds are known for the network distance (according to all cost types), the traditional A* search may also be adapted to MCPP [27], [28]. MCPP is inherently different from our MCN skyline, because (i) the former computes a skyline of *paths*, while the latter is a skyline of *facilities*, (ii) in MCPP there is a single and given destination, whereas in our case the destination could be any facility in P, (iii) MCPP considers all possible paths between v_s and v_d, but we take into account only the *shortest* paths according to each cost type.

III. Problem Formulation

In this section we formalize the problem and the queries we consider. We also state our assumptions and scope of applicability.

Multi-cost network (MCN): An MCN is a road network (weighted graph) $G = \{V, E, W\}$, where V is the set of nodes, E is the set of edges, and W is a function that associates each edge $e \in E$ with a cost vector $\vec{w}\ (e)$, where $\vec{w}\ (e) = \{w_1, w_2, ..., w_d\}$. Values w_i (for $i = 1...d$) are the costs of e according to the d cost types involved in decision making. For example, w_1 could be the Euclidean length of e, w_2 could be the walking time to reach from one end-node to the other, w_3 could be the driving time, w_4 could be the toll fee, etc. The only assumption we make about the w_i values is that they are non-negative[2]. We assume undirected edges, where the cost vector to either direction is identical. However, our methods extend trivially to directed edges. The node information may or may not include the spatial coordinates of nodes $v \in V$, i.e., our methods do not rely on the location of the nodes but only on the connectivity among them. For ease of presentation, in the following we often refer to an edge e using its end-nodes as $\langle v_i, v_j \rangle$.

Facility set P: This is the entire set of points of interest, i.e., the set of available facilities that our applications may choose from. All facilities $p \in P$ fall on the edges of the MCN. If a facility p falls between the end-nodes of an edge e, then the *partial weight* from p to either end-node of e is proportional to their Euclidean distance, while the sum of the two partial weights is equal to $\vec{w}\ (e)$. The facilities may be associated with additional (non-spatial, non-network-based) information, e.g., in the warehouse example, this could be the capacity of the warehouse, name of owner, etc.

Shortest path and smallest cost vectors: Given a query location q on the MCN, we denote as $s_i(q, p)$ (for some $p \in P$) the shortest path from q to p in terms of the i-th cost type, and as $c_i(q, p)$ its total cost, i.e., the sum of the w_i values of the edges (partial or not) included in the path[3]. For example, $s_1(q, p)$ is the path between q and p with the smallest Euclidean length (with $c_1(q, p)$ being its Euclidean length), and $s_2(q, p)$ is the fastest path between q and p in terms of walking time (with $c_2(q, p)$ being the total walking time along $s_2(q, p)$). The d shortest paths and their costs define vectors $\vec{s}\ (q, p)$ and $\vec{c}\ (q, p)$ respectively. For simplicity, when the query location is clear from the context, we omit q from the notation, e.g., we use $\vec{s}\ (p)$ instead of $\vec{s}\ (q, p)$.

MCN skyline: Given a query location q on the MCN, its skyline $sky(q)$ includes those and only those facilities $p \in P$ that are not *dominated* by any other point $p' \in P$; a facility p' dominates p if and only if $c_i(p') \leq c_i(p), \forall i \in [1, d]$, and $c_j(p') < c_j(p)$ for some $j \in [1, d]$. Essentially, this means that there is no facility in $\{P - sky(q)\}$ that is cheaper to reach according to all costs than a facility in $sky(q)$.

MCN top-k query: The input of this query includes the query location q (which must fall on the MCN), an aggregate cost function f, and an application-defined positive integer k. Function f maps each facility p to a real number $f(p)$, called the *aggregate cost* of p. Function f is defined over $\vec{c}\ (p)$, i.e., over the d individual costs $c_i(p)$, and is an *increasingly monotone* function. This means that if and only if $c_i(p) \leq c_i(p'), \forall i \in [1, d]$ for two facilities p, p' in P then $f(p) \leq f(p')$. The result of the query is a subset $top(q)$ of P, containing the k facilities with the smallest aggregate costs, i.e., $f(p) \leq f(p'), \forall p \in top(q)$, $p' \in \{P - top(q)\}$ and $|top(q)| = k$. Ties for the k-th facility are resolved arbitrarily. We must stress that the *incremental* top-k query does not require k as an input, but may report iteratively the facility with the immediately larger aggregate cost than its last output result.

There exists an interesting connection between the MCN skyline and the top-k query. Like in their conventional counterparts, the skyline contains all facilities that belong to the result of any top-1 query with an increasingly monotone aggregate cost function.

We assume that the MCN and the facility set are organized in secondary storage using an adaptation of the storage scheme in [2], exemplified in Figure 2. Given a node identifier (e.g., v_i), the *adjacency tree* links to the location of a flat file (the *adjacency file*) that contains the node's adjacency list. The first entry of this list implies that v_i is adjacent to v_j and provides the cost vector of edge $\langle v_i, v_j \rangle$; it also has a pointer into the *facility file* that stores the facilities lying on the edge. For each facility (e.g., p_m), the facility file includes its Euclidean distance from the first end-node of the edge (i.e., $|v_i p_m|$) so that its partial weights can be computed. The *facility tree* stores for each facility p the identifier of the edge it lies in and a pointer to p's location in the facility file.

In the following section (Section IV) we describe our skyline algorithms, which also constitute the basis for the top-k methods (Section V). Table I summarizes the notation used in the paper.

[2]We consider general costs for which no bounds exist and, thus, we rely on Dijkstra's algorithm and its disk-based counterpart, the network expansion technique (described in Section II-C).

[3]Note that if q falls in between the end-nodes of an edge, the partial weights to the end-nodes are defined as described previously for the facilities.

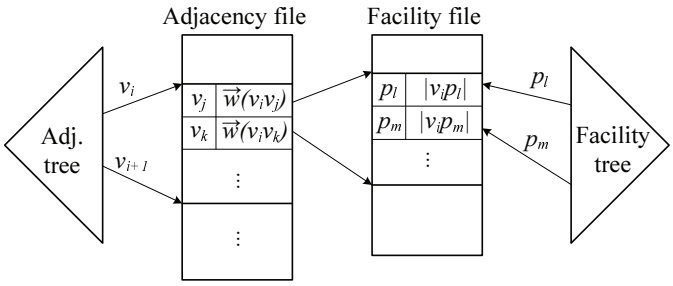


Fig. 2. Network storage scheme

TABLE I
INTERPRETATION OF NOTATION

Symbol	Description
v	An MCN node
e or $\langle v_i, v_j \rangle$	An MCN edge between nodes v_i, v_j
d	The number of cost types considered
$\vec{w}\ (e)$	The cost vector of edge e
P	The set of facilities
q	The query location on the MCN
$s_i(p)$	Shortest path from q to p w.r.t. i-th cost type
$c_i(p)$	Cost of $s_i(p)$ (in terms of the i-th cost type)
$\vec{s}\ (p)$	The shortest path vector of facility p
$\vec{c}\ (p)$	The cost vector of facility p
$sky(q)$	The skyline of q
$f(p)$	The aggregate cost of p
$top(q)$	The top-k set of q
CS	Candidate set

IV. MCN SKYLINE PROCESSING

A straightforward way to compute the skyline of a query location q is to perform d complete network expansions from q to all facilities $p \in P$, and thus compute their cost vectors $\vec{c}\ (p)$. After that, the cost vectors can be processed by any traditional skyline algorithm (see Section II-A). The problem of this method is that it reads the entire database (MCN and facility information) d times, which leads to a prohibitively long running time. Intuitively, the skyline facilities should be located close to q, and the MCN should have to be explored only around it. This is the objective of the *Local Search Algorithm* (LSA) described next.

A. Local Search Algorithm

LSA prunes the search space by performing the d cost expansions concurrently and stopping when none of them may lead to new skyline facilities. Specifically, LSA initializes one NN network expansion for each cost type at query location q. As explained in Section II-C, NNs (i.e., nearest facilities) can be discovered incrementally for each of the costs. LSA proceeds in two stages; the *growing* and the *shrinking* one.

Growing stage: During the growing stage, candidate skyline facilities are identified. LSA iteratively pops the next NN for one of the expansions, cycling through the d choices in a round-robin fashion. Every facility (NN) output is included in the *candidate set CS*. The growing stage ends when the first facility is *pinned*. We say that a facility is pinned when it has been output by all d expansions, which means that its complete cost vector $\vec{c}\ (p)$ has been computed.

To illustrate the growing stage, we use the example in Figure 3 where $d = 2$ and the network distances $c_1(p), c_2(p)$ of encountered facilities are plotted in the (conceptual) 2-dimensional space for ease of illustration. On the left of the figure we can see the order in which the NNs are popped for each of the cost types. The NN of q according to c_1 is found first (facility p_1) and is inserted into *CS*. Then, the NN according to c_2 is retrieved (facility p_5), and since it is not inside *CS*, it is inserted therein. The process continues this way until p_3 is pinned (p_3 is first found w.r.t. c_2, and when found w.r.t. c_1, its complete cost vector is derived). Growing stops here, with candidate set $CS = \{p_1, p_2, p_5\}$. The vertical (horizontal) dashed line corresponds to the frontier of network expansion according to c_1 (c_2 respectively), i.e., it indicates how far, in terms of c_1 (c_2) network distance, the search has proceeded so far.

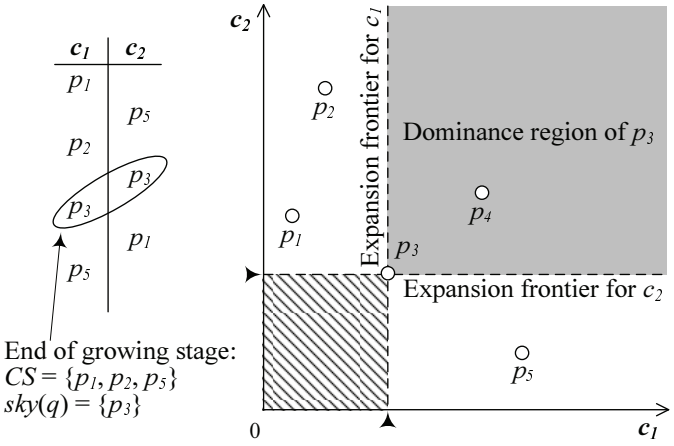


Fig. 3. Skyline computation example ($d = 2$)

The pinned facility can be output directly as part of the skyline. In our example, the pinned p_3 could only be dominated by some facility in the striped area. This area is definitely empty, because otherwise one of the contained facilities would have been pinned before p_3 (note that the stripped area has been explored for both cost types). Therefore, p_3 is a skyline facility. On the other hand, by definition the pinned facility cannot dominate any candidate, because they were all encountered before it, and thus are preferable w.r.t. at least one of the cost types.

After the first facility is pinned, no more candidates need to be considered because they are guaranteed to be dominated by the pinned facility. To illustrate this, consider again Figure 3 upon pinning p_3. The shaded area is termed the *dominance region* of p_3, and any facility (such as p_4) that falls inside it is dominated by p_3. Since the d expansions have already explored the entire space outside the dominance region of the pinned facility, all possible skyline facilities have already been encountered and included in *CS*. What remains to be done is

deciding for each candidate whether it is a skyline facility or not. This is the role of the shrinking stage[4].

Shrinking stage: The shrinking stage continues popping NNs incrementally for each of the cost types. The difference is that it ignores any facility that is not in *CS* (i.e., that is newly encountered). If a candidate facility $p \in CS$ is popped from the i-th expansion heap, then LSA records its i-th cost ($c_i(p)$) accordingly. If p is now pinned (i) it is directly output as a skyline facility, and (ii) we scan the remaining candidates and eliminate those dominated by p. The shrinking stage (and LSA) terminates when *CS* becomes empty.

The rationale behind directly reporting the newly pinned p as a skyline facility is similar to the case of the first pinned facility. If another facility p' were dominating p, then p' would have been pinned first (since all p''s costs are no larger than p's, all expansions would encounter it before p). In this case, p would have been eliminated from the candidate set (upon pinning p'), which is a contradiction. Thus, p is a skyline facility.

Regarding dominance checks against the pinned facility p, the issue is that each candidate has, by definition, at least one unknown cost. To address this, we rely on the nature of network expansion, and specifically on the fact that NNs are discovered in increasing cost order. Since p has been encountered by all d expansions, we can be sure that any unknown cost of any candidate is no smaller than the corresponding cost of p. Therefore, if p dominates a candidate p' w.r.t. the known costs of the latter, p is guaranteed to dominate p' w.r.t. all d costs.

Another point worth stressing is that, due to the nature of LSA, dominance checks are performed only when a facility is pinned. Moreover, dominance checks are performed only w.r.t. the newly pinned facility, and not with the previously discovered skyline facilities. The reason is that if a candidate were dominated by a previously discovered skyline facility, it would have been directly eliminated at the time when the latter was pinned (as described above). Due to this inherent feature, LSA performs a small number of dominance checks overall.

Continuing our example in Figure 3, after pinning p_3, it is the turn of c_2's expansion to proceed. This pops facility p_1 (which is already in *CS*) and pins it, since now all its $d = 2$ costs are known. Facility p_1 is appended to $sky(q)$ (now $sky(q) = \{p_3, p_1\}$), and then probed against the remaining candidates (p_2 and p_5). For p_2 only $c_1(p_2)$ is known, and it is larger than $c_1(p_1)$. On the other hand, $c_2(p_2)$ is not known (since the second expansion has not reached it yet), but we can be sure that it will be no smaller than $c_2(p_1)$. Therefore, we can safely infer that p_1 dominates p_2 and eliminate the latter. On the contrary, we cannot eliminate candidate p_5, because its second cost is smaller than p_1.

Next, it is the turn of the first expansion to proceed, which pops p_4. Since $p_4 \notin CS$, we ignore it, and incrementally retrieve the next NN (w.r.t. c_1), which is p_5. Facility p_5 is pinned, and thus included in $sky(q)$. Furthermore, *CS* is now empty, and LSA terminates having computed the complete skyline $sky(q) = \{p_3, p_1, p_5\}$. Algorithm 1 is the pseudo-code of LSA.

Algorithm 1 Local Search Algorithm (LSA)

```
   algorithm LSA(Query Location q)
1: sky(q) := ∅; CS := ∅; stage := Growing
2: Initialize d network expansions at q (one for each cost type)
3: for i = 1 to d do                                  ▷ Round-robin probing
4:     Fetch the next NN p from the i-th expansion
5:     if p ∉ CS then
6:         if stage == Growing then
7:             Insert entry ⟨p, Null, Null, ..., c_i(p), Null, ...⟩ in CS
8:     else
9:         Update p's entry in CS, setting its i-th cost to c_i(p)
10:        if p is pinned then
11:            if stage == Growing then
12:                stage := Shrinking                 ▷ Growing stage ends
13:            sky(q) := sky(q) ∪ {p}
14:            Eliminate CS entries dominated by p
15:    if CS == ∅ then
16:        Report sky(q) and terminate
17:    if i == d then
18:        Go to line 3 (reiterate)
```

Having sketched the LSA process, we may now enhance its performance based on some of its intrinsic properties. First, by definition the first NN for each of the d cost types cannot be possibly dominated by another facility, and can thus be directly reported as part of the skyline. This reduces the number of candidates considered, which in turn may lead to earlier termination (because LSA no longer has to pin these skyline facilities, but reports them directly). In Figure 3, for example, p_1 and p_5 would be directly reported as skyline facilities. Therefore, at the end of the growing stage, *CS* would only contain p_2. Note that if one of these non-pinned facilities that are included directly in $sky(q)$ is pinned later on, it is then checked for domination against (and potentially eliminates) facilities in *CS*. Continuing our example, after pinning p_3, facility p_1 is encountered by c_2's expansion and is pinned. p_1 dominates/eliminates the only candidate (p_2), and LSA terminates (since *CS* is empty) without having to pin p_5.

Another enhancement leverages on the fact that the shrinking stage ignores any newly encountered facility. This means that we should not have to access the facility file (described in Figure 2) during any expansion in the shrinking stage, except when a candidate facility is to be popped. To save these unnecessary I/Os, while avoiding missing any candidate facilities, we do the following.

When the first facility is pinned (i.e., when growing ends), we probe the facility tree and retrieve the edge identifier of each candidate. With this information in hand, our d

[4]Note that it is possible for some of the candidates to be directly eliminated. This happens when all their known costs are equal to those of the pinned facility, which means that they are dominated by it (because their unknown costs are guaranteed to be no smaller than those of the pinned facility). Such facilities should be eliminated directly before the shrinking stage commences. For ease of presentation, we ignore this case, and assume for simplicity that no ties exist.

expansions proceed through the network edges *without* accessing their contained facilities, but only do so when some candidate's edge is encountered. The corresponding candidate is en-heaped[5] and the expansion proceeds as normal, ignoring any non-candidate facilities. A side improvement achieved with this optimization, is that the number of heap operations (translating to CPU time), as well as memory requirements, are reduced by avoiding en-heaping non-candidate facilities.

Another enhancement is that we may (safely) terminate some of the d expansions before LSA returns. Specifically, if we are in the shrinking stage, and the i-th cost of every candidate has been computed, then the expansion for the i-th cost type stops, because it can contribute nothing to the search. This optimization is particularly useful in cases where the cost distribution of the facilities is such that many skyline facilities happen to be among the first NNs w.r.t. a specific cost type.

Discussion: LSA could work with a different expansion probing policy, other than round-robin. For example, we could be choosing to probe the expansion for which the top heap element has the smallest key. The problem with this approach is that if a cost happens to be low in the neighborhood of q, it would monopolize probing, i.e., other expansions would be probed late or infrequently. This would lead to pinning the first facility late. This prolongs the growing stage itself, but also slows down shrinking because more candidates are encountered. To exemplify, consider Figure 4. A smallest-first probing strategy would keep exploring the NNs for c_1, because they have consistently lower c_1 cost than the c_2 cost of the first NN for c_2 (facility p_7); i.e., $c_1(p_1), c_1(p_2), ..., c_1(p_7)$ are smaller than $c_2(p_7)$. Similar problems occur when probing the expansion with the maximum top element (largest-first). Since we assume no a priori knowledge about the cost distributions, we choose the round-robin technique that favors no cost, and would pin a facility early in most cases.

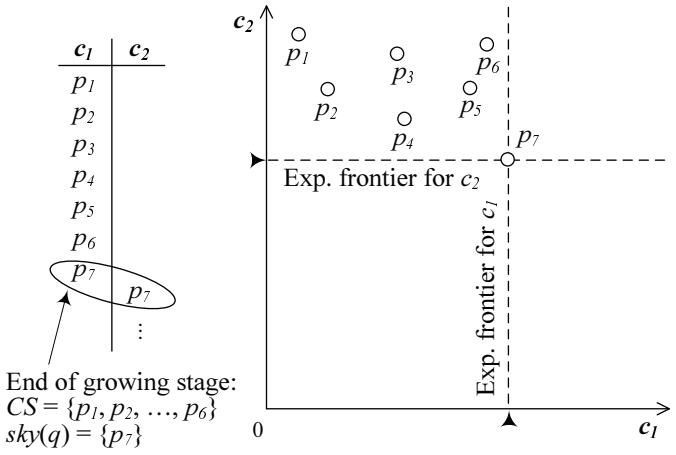


Fig. 4. Smallest-first probing ($d = 2$)

Based on the observation that the earlier a facility is pinned the faster LSA terminates, one could argue that once a NN p is retrieved for one cost, its remaining $d - 1$ costs should be computed directly so that p can be used for candidate elimination right away. This, however, would not pay off, since individual cost (i.e., shortest path) computation is itself based on network expansion, and is costly. Consider again Figure 4. Facility p_1 is the first NN retrieved (by c_1's expansion). Computing its remaining cost $c_2(p_1)$ would require a network expansion for c_2 that extends further than all other facilities $(p_7, p_4, ..., p_6)$. This incurs a high CPU and I/O cost, without particularly assisting LSA in pruning the search space.

Before presenting our next skyline method, we stress that LSA is *progressive*; any pinned facility is guaranteed to belong to the skyline, and can thus be output directly, without having to wait for LSA to terminate.

B. Combined Expansion Algorithm

LSA clearly improves performance compared to the straightforward method described in the introduction of Section IV, since search only considers the neighborhood of q. LSA however leaves significant space for improvement in terms of the I/Os incurred. Specifically, the adjacency information of an accessed network node or the facility information of a traversed edge is typically loaded from the disk multiple times (up to d) for different expansions. This motivates the *Combined Expansion Algorithm* (CEA), which is guaranteed to access each encountered node's adjacency information or edge's contents no more than once. This is achieved by information sharing among the d expansions as described below.

We begin our description using a plain (albeit flawed) version of information sharing, and then introduce the optimized CEA. Assume that we allow only one of the d expansions to access the disk-resident structures of Figure 2; let this be the expansion for cost c_1. The expansion starts incrementally en-heaping new nodes and facilities. Whenever a node is de-heaped, its adjacent nodes are loaded; in addition to c_1, this also fetches the remaining $d - 1$ costs of the corresponding edges. Therefore, every adjacent node is also pushed into the remaining $d - 1$ heaps (with key set according to the corresponding cost type). The same policy applies to the encountered facilities too.

The expansion for c_1 proceeds normally as described above, until the first NN is discovered (w.r.t. c_1) and placed into *CS*. We call each popped network node *expanded*, because its adjacency information has been retrieved and its neighboring nodes have been en-heaped. Note that term expanded applies only to network nodes popped for cost c_1.

Once the first NN for c_1 is found, we suspend c_1's expansion, and expand the network for each of the other costs. In particular, for each other cost type (say for c_2), we keep popping the head of its expansion heap, until a non-expanded network node v_{ne} is de-heaped. If any facilities have been popped meanwhile, they are inserted into *CS* or (if they are already in *CS*) their c_2 costs are recorded. The expansion for c_2 is suspended at this point, and can only proceed when v_{ne}

[5]Recall that the standard network expansion en-heaps every encountered facility, treating it as a network node.

is expanded by c_1's search. When c_2's search is suspended, the expansion of c_3 commences and proceeds until it pops a non-expanded node, and so on for the remaining cost types. We call this the first *pass* of the algorithm.

In the beginning of the second pass, c_1's expansion resumes and proceeds until its next NN is found. Any node or facility en-heaped for c_1, is also pushed into the heaps of the other $d-1$ costs[6]. The newly found NN is placed into *CS*. The other expansions are considered again one by one, and stop when they pop a non-expanded node.

This process continues until the first facility is pinned. This event signifies the end of the growing stage, and forbids the insertion of any new candidates into *CS*. Having entered the shrinking stage, expansions proceed as above until all candidates are either eliminated or reported as skyline facilities.

To conclude the above process, we must explain (i) why the expansions are suspended and (ii) how we can be sure that the NNs retrieved so far for costs $c_2, c_3, ..., c_d$ are correct. Consider the example in Figure 5, representing the situation in the second pass where the next (second) NN of c_1 has been found (facility p_2) and its expansion is suspended. Network nodes appear as squares. Solid nodes are expanded, i.e., they have been popped by c_1's expansion (at any point up to its current suspension), and thus their adjacent nodes have been pushed into all expansion heaps. Hollow nodes have been en-heaped but not de-heaped by c_1's expansion, i.e., they are non-expanded (such an example is node v_3, whose complete adjacency information is unavailable). Nodes omitted from the figure are nodes not encountered at all by the expansion of c_1. Consider now the expansion of cost type c_2. According to the process described previously, c_2's expansion will be suspended once v_3 is popped; this is the closest (w.r.t. c_2) non-expanded node. Clearly, the process cannot proceed further because not all neighboring nodes of v_3 are known. On the other hand, the complete information for all nodes and facilities within distance $c_2(v_3)$ from q is available; the shaded area in the figure covers this part of the network (typically there is no correspondence to a circle or any ordinary geometric shape, but it is drawn as such for the sake of demonstration). Given full information for this subgraph, the correctness of network expansion therein guarantees that the NNs retrieved prior to suspension are correct (i.e., identical to those of a c_2 expansion in the entire network).

The above algorithm does achieve information sharing, but it provides no control over the order of NN retrieval among the different expansions and suffers from a similar problem to the smallest-first/largest-first variants of LSA. Specifically, if c_1 happens to be very skewed, leading the expansion west from q (i.e., the c_1 costs of west edges are considerably smaller than other directions) then the remaining expansions may remain suspended for long periods, because they will have to wait for c_1 to de-heap nodes towards, say, the east. In the example of Figure 5, c_1 is exhibiting such a behavior, expanding to the west more so than to the east. In general, a similar situation occurs whenever c_1 follows a different distribution than some of the remaining costs. This is expected to be often the case for some costs; e.g., the driving time through main streets is usually shorter, but the total toll cost increases.

[6]The fact that these searches are suspended does not prevent us from pushing new nodes/facilities into their expansion heaps.

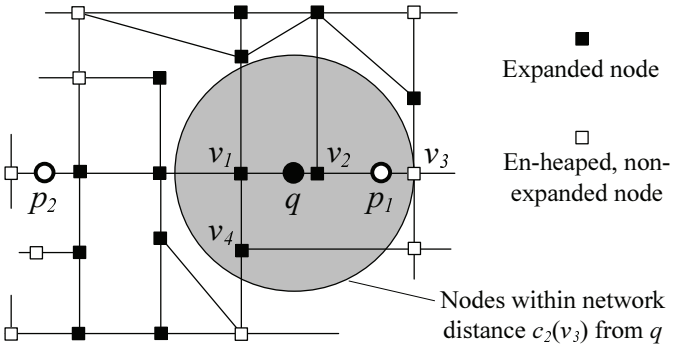


Fig. 5. Second pass example

To avoid the above problem, we should somehow multiplex the d expansions and let them (to some degree) proceed independently from others, while at the same time sharing information among them. Additionally, we would like to achieve round-robin retrieval of NNs among the d expansions. CEA meets both these targets.

In CEA, only one expansion is allowed to access the disk at any time, while the information retrieved is also en-heaped by the remaining $d-1$ expansions. However, the right to access the disk cycles among the d expansions in a round-robin fashion. When an expansion is given this right, it proceeds until it retrieves its next NN. Note that if a non-expanded node is encountered during this search, then the expansion is not suspended, but proceeds (and accesses the disk-resident adjacency/facility information) as per normal. Then, the next expansion resumes until it discovers its next NN, and so on. We must stress that now a node may be expanded by any expansion, as long as this expansion has the disk access right.

Figure 6 shows an example of combined expansion where $d = 2$, the first cost type (c_1) is driving time and the second (c_2) is the Euclidean length of the path (referred to as *time* and *distance* in the figure). The first expansion is probed first, which en-heaps v_1 and v_2 with keys 4 and 6 (minutes) respectively. v_1 is popped and expanded, i.e., its adjacent nodes and the facilities in the corresponding edges are pushed (into all $d = 2$ expansion heaps). The next element popped is facility p_1 with $c_1(p_1) = 5$ min, which is the first NN for c_1. To achieve round-robin probing, we now need to retrieve the NN for c_2; its heap contains all nodes/facilities fetched for c_1, i.e., v_1, v_2, and all the nodes and facilities adjacent to v_1. The head of the heap is v_1, which has already been expanded for c_1, thus expansion proceeds without any disk accesses. The next element popped is v_2, which was not previously expanded; its adjacency information is fetched from the disk and the corresponding nodes and facilities are en-heaped (to both expansion heaps). Next, p_2 is popped with $c_2(p_2) = 4$

km. Observe that v_1 was expanded for c_1, while v_2 for c_2. Round-robin probing continues this way, never accessing the adjacency and neighboring facility information more than once for the same node.

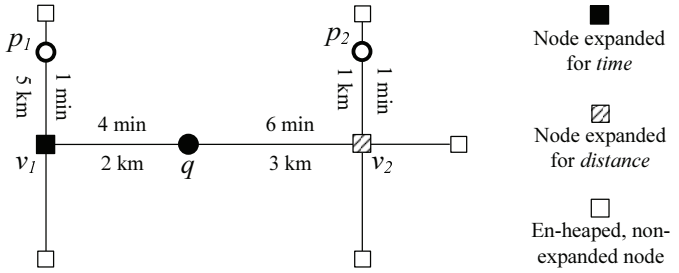


Fig. 6. CEA example ($d = 2$)

Other than the above, CEA proceeds like LSA, using a growing and a shrinking stage. Moreover, it encounters and pins facilities in exactly the same order as LSA (due to its round-robin NN probing), which implies that it has the same candidate set and skyline reporting order, and that it is also progressive. The correctness of CEA is derived following a similar reasoning to Section IV-A. Algorithm 2 is the pseudo-code of CEA.

Algorithm 2 Combined Expansion Algorithm (CEA)

```
    algorithm CEA(Query Location q)
 1: sky(q) := ∅; CS := ∅; stage := Growing
 2: Initialize d network expansions at q (one for each cost type)
 3: for i = 1 to d do                          ▷ Round-robin disk access right
 4:     repeat
 5:         Pop the head of i-th expansion heap
 6:         if the popped element is a node v then
 7:             Fetch v's adj. nodes and facilities on incident edges
 8:             En-heap fetched nodes/facilities in all d exp. heaps
 9:         else                              ▷ The next NN p was popped
10:             if p ∉ CS then
11:                 if stage == Growing then
12:                     Insert ⟨p, Null, ..., c_i(p), Null, ...⟩ in CS
13:             else
14:                 Update p's entry in CS, setting its i-th cost to c_i(p)
15:                 if p is pinned then
16:                     if stage == Growing then
17:                         stage := Shrinking ▷ Growing stage ends
18:                     sky(q) := sky(q) ∪ {p}
19:                     Eliminate CS entries dominated by p
20:                     if CS == ∅ then
21:                         Report sky(q) and terminate
22:     until the next NN for c_i has been found
23:     if i == d then
24:         Go to line 3 (reiterate)
```

V. MCN Top-k Processing

Top-k processing is largely based on our skyline computation techniques, with pinned facilities and expanded nodes playing similar roles as in Section IV. Both LSA and CEA apply to top-k retrieval; here we focus on the greater picture of processing, regardless of which of the two techniques is used to achieve the round-robin retrieval of NNs.

Processing again comprises a growing and a shrinking stage. The growing stage is identical to Section IV, the difference being that growing stops when k facilities are pinned (instead of only one). Every encountered facility in this stage is a candidate and is placed into *CS*. Every pinned candidate is placed into $top(q)$. To see why the union of *CS* and $top(q)$ contains all possible top-k candidates, recall that when a facility is pinned, it is guaranteed to dominate all facilities that will be encountered after its pinning. When k facilities are pinned, they all dominate any non-encountered facility. By definition, this means that each of the k pinned facilities has smaller aggregate cost than any non-encountered facility, for any increasingly monotone function f. In Figure 7, for example, a top-3 query is considered. The first facility pinned is p_1, the second is p_3, and the third is p_4. Upon pinning p_4, any facility in its dominance region (the darkest shaded area in the figure) is guaranteed to be dominated also by p_1 and p_3. Thus, any non-encountered facility (such as p_5) cannot belong to the top-3 result of any increasingly monotone cost function. In this example, growing ends with one candidate (p_2) and a *tentative* top-k set comprising the $k = 3$ pinned facilities.

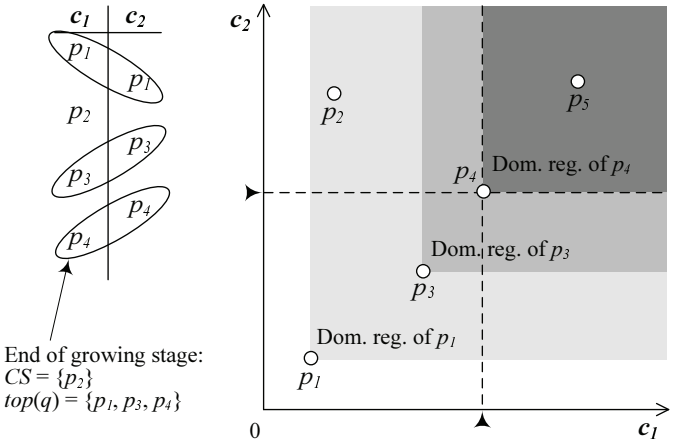


Fig. 7. Top-k processing example ($d = 2$, $k = 3$)

The $top(q)$ set formed by the growing stage is tentative because some of the candidates may have lower aggregate cost. In Figure 7, for example, if f favors a smaller c_1 considerably more than a small c_2, candidate p_2 may have a smaller aggregate cost than, say, p_4, and may thus replace p_4 in the final $top(q)$ result. The task of determining the final top-k facilities is performed in the shrinking phase.

Shrinking continues probing the expansion heaps, without however en-heaping (or inserting into *CS*) any additional facilities. If at some point a candidate p is pinned, then its aggregate cost $f(p)$ is computed. If $f(p)$ is smaller than the largest (k-th) score in $top(q)$, then p replaces the k-th best facility in $top(q)$ and the evicted facility is eliminated. Otherwise (i.e., $f(p)$ is no smaller than the k-th score in $top(q)$), p is eliminated. Note that similar optimizations to the

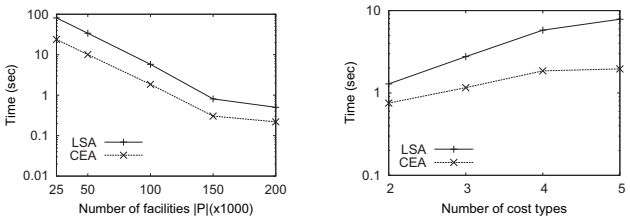
skyline algorithms apply to top-k processing; the shrinking stage does not access any facility information from the disk, and if no candidate has a missing value for a specific cost type c_i, the expansion for c_i stops permanently and is ignored by round-robin probing.

The algorithm terminates (and reports the current $top(q)$ as result) when all candidates have been either eliminated or included in $top(q)$. A straightforward way to do this is to continue expanding until all the candidates are pinned and their aggregate costs computed. This, however, would incur unnecessary I/O overhead, because some candidates can be eliminated without being pinned. Let $t_1, t_2, ..., t_d$ be the keys at the heads of the d expansion heaps. Assume that for a candidate p we have computed all costs except for $c_1(p)$ and $c_2(p)$. Due to the incremental nature of NN search, we know that $c_1(p) \geq t_1$ and $c_2(p) \geq t_2$. This provides a lower bound for p's aggregate cost. Specifically, it holds that $f(p) \geq f(t_1, t_2, c_3(p), ..., c_d(p))$, i.e., the lower bound is derived by applying f on p, after replacing its *Null* costs with the corresponding t_i values. If the derived lower bound is no smaller than the k-th largest aggregate cost in $top(q)$, candidate p can be safely eliminated (without waiting for its pinning). We perform this lower bound check for each candidate after every complete pass of expansion probing; in the shrinking stage, each expansion is suspended after popping one node from its heap (recall that in this stage there are few facilities in the heaps, i.e., only candidates, and probing until the next NN is found would be costly).

The above technique applies to cases where k is known in advance. Its incremental version (where k is not given) requires several modifications. First, there is no shrinking phase, i.e., no candidate or pinned facility can ever be eliminated; even the facility p with the largest aggregate cost in P will need to be reported if the incremental algorithm is invoked $|P|$ times. Second, we need to know when a facility p is safe to report as the next result. For a facility p to be the next result it must (i) have been pinned, (ii) have the smallest aggregate cost among all pinned facilities that have not yet been reported, and (iii) the lower bound aggregate cost *of every candidate that was encountered before pinning* p must be no smaller than $f(p)$. While the second condition is obvious, the first and third are not. The first condition is necessary because no matter how small the known costs of a facility are, the remaining ones can be arbitrarily large, leading to an arbitrarily large aggregate cost. Regarding the third condition, p dominates any facility (candidate or pinned) encountered after its pinning. Therefore, the only candidates that may potentially have smaller aggregate score are those encountered before its pinning. An additional remark about the third condition is that the aggregate cost lower bounds for candidate facilities still apply, derived exactly as described previously for the known k case.

VI. Experiments

We evaluate the performance of LSA and CEA using the road network of San Francisco (obtained from [29]), which contains 174,956 nodes and 223,001 edges. We generated facility set P to form 10 Gaussian clusters centered around 10 random nodes in the network. This simulates a real scenario, where most of the facilities are located around specific locations in a city (e.g., the business district, the port area, etc). The number of facilities $|P|$ varies between 25K and 200K, with default value 100K. We assigned d costs to each edge ($d \in [2...5]$, with default $d = 4$). The costs of an edge follow independent, anti-correlated or correlated distribution (with anti-correlated being the default); these are standard distributions in preference query evaluation [3]. In anti-correlated, when one cost is low, the rest tend to be high. In correlated, when one cost is low, the others tend to be low too. We use an LRU buffer with size from 0% to 2% of the total number of pages occupied by the MCN information (1% by default). For top-k queries, k is between 1 and 16, with $k = 4$ being the default. The aggregate cost function f has the form $f(p) = \sum_{i=1..d} \alpha_i \cdot c_i(p)$, where coefficients α_i are randomly and independently generated with values between 0 and 1. In each experiment we vary one parameter while setting the remaining ones to their defaults. We measure the total processing time (inclusive of I/O and CPU cost).[7] The reported results are average values over 100 different query locations randomly and uniformly chosen in the network.



(a) Effect of $|P|$ (b) Effect of d

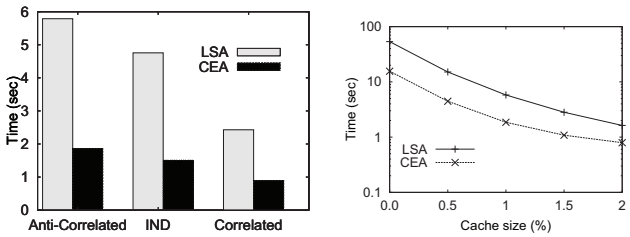
Fig. 8. Processing time versus P and d (skyline)

A. Skyline Experiments

We first consider skyline queries. Figure 8(a) investigates the effect of the number of facilities $|P|$, varying it from 25K to 200K, and setting the remaining parameters to their defaults. Time measurements are plotted in logarithmic scale. Interestingly, the processing time for both LSA and CEA is longer for small $|P|$. The reason for this is that when the network is very sparse (with facilities), then the expansions of both algorithms need to consider many edges before the next NNs are found, and thus incur many I/Os on the adjacency tree and the adjacency file (see Figure 2). CEA is more than 2.3 times faster than LSA in all cases.

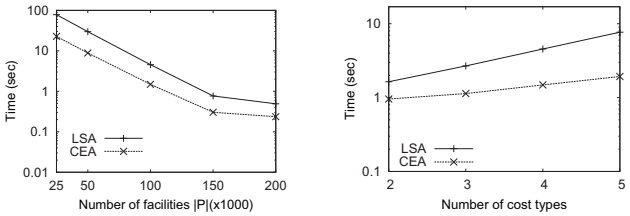
Figure 8(b) plots the processing time as a function of the number of the cost types d in the MCN, for $d = 2...5$. The performance of both methods deteriorates with d, because (i) more (i.e., d) expansions are executed, and (ii) as d increases,

[7]The processing time is vastly dominated by the I/O cost, with the CPU time accounting for only 5% of the total time of LSA and 16% of CEA in our default setting. Thus, we plot only the total time for brevity.



(a) Effect of cost distribution (b) Effect of buffer size

Fig. 9. Processing time versus cost distribution and buffer size (skyline)



(a) Effect of $|P|$ (b) Effect of d

Fig. 10. Processing time versus P and d (top-k)

it is harder (i.e., it takes longer) to pin the first facility, which leads to a larger candidate set. CEA is between 1.7 to 4 times faster than LSA (note the logarithmic scale in the processing time axis). The performance gap increases with d. For a given d, LSA may access the same node's adjacency information (or the same edge's facilities) up to d times. Therefore, CEA's advantage of accessing such information no more than once is more pronounced for larger d.

Figure 9(a) experiments with the cost distributions within the edges. As expected, the running time of both methods is the longest for anti-correlated costs, because facilities that are close to the query location according to a cost type, tend to be further according to others. This reduces the probability that a facility dominates another, and therefore leads to more candidates and more skyline facilities. On the other hand, the shortest processing time is for correlated costs, because the first pinned facility tends to be close according to all cost types, and is thus pinned early. This means that the search considers fewer candidates and a smaller part of the network, and it outputs fewer skyline facilities. The largest difference between CEA and LSA is observed for independent costs (3.15 times improvement), and the smallest for correlated costs (2.7 times).

Figure 9(b) studies the effect of the buffer size, varying it from 0% to 2% of the MCN size, and setting the remaining parameters to their defaults. The processing cost is plotted in logarithmic scale. Both methods benefit from the buffer, but more so LSA; this is expected, because its multiple-read problem is smoothened, as more of its multiple requests to the same disk page find it already in the buffer. CEA is between 2 and 3.4 times faster (for buffer size 2% and 0% respectively). Note that the cost of both methods is much higher in the 0% case (no buffer), because adjacency information requests for different nodes that reside on the same disk page lead to reading the page multiple times.

B. Top-k Experiments

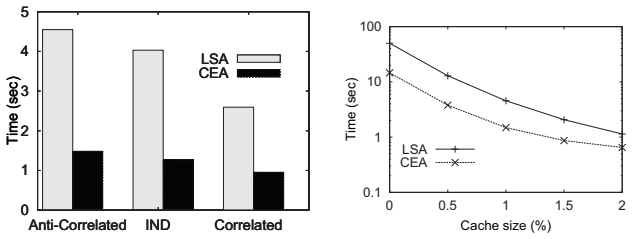
Having empirically explored skyline queries, we now evaluate our top-k algorithms using the same road network (San Francisco). In Figure 10(a) we vary the number of facilities, using $d = 4$ anti-correlated cost types, aggregate cost function f with independent coefficients α_i, and number of requested facilities $k = 4$. Like in Figure 8(a) in the skyline case, both CEA and LSA benefit from a large facility population, because the search discovers the top-k results close to the query location, thus saving accesses on the MCN connectivity information (adjacency tree and adjacency file). CEA is 2.1 to 3.4 times faster than LSA. The difference between them increases for a sparse network, because more nodes/edges are accessed, which in turn implies more multiple-reads for LSA. Juxtaposing the results in this experiment with Figure 8(a), an interesting observation is that top-4 processing is slightly less expensive than skyline computation. Although top-4 processing requires pinning four facilities (instead of one in the skyline case), most of the "extra" facilities are located inside the dominance region of the first pinned facility, they are pinned soon after the first pinned facility, and they have comparably low aggregate costs to the first pinned facility; thus, the growing stage is not significantly longer than in the skyline case, while the low costs of the pinned facilities allow for more effective pruning in the shrinking stage, leading to an earlier termination overall.

In Figure 10(b) we use the default parameters for top-k processing, and measure the effect of the number of costs d on the processing time. As in the skyline case (Figure 8(b)), more costs imply a longer running time for both our methods, as more expansions are executed and pinning a facility becomes harder. CEA is faster than LSA in all cases, and their difference increases with d, due to the same reasons explained for the skyline case.

Figure 11(a) examines the effect of the cost distribution within the edges on top-k processing time. Similarly to the skyline query (Figure 9(a)), correlated edge costs lead to the shortest processing time, while anti-correlated costs to the longest one. CEA is 3, 3.2, and 2.7 times faster than LSA for anti-correlated, independent and correlated edge costs respectively.

Figure 11(b) varies the buffer size, while keeping the remaining parameters to their defaults. The trends are similar to Figure 9(b) for skyline processing; i.e., the performance of both methods improves as the buffer size grows. In this case (top-k processing), CEA is up to 3.4 times faster than LSA for 0% buffer size (no buffer). The smallest relative difference is in the 2% case, with CEA however still being 1.8 times more efficient than LSA.

Figure 12 varies the number k of required facilities between 1 and 16, while the other parameters are set to their defaults. As k increases, more facilities need to be pinned in the



(a) Effect of cost distribution (b) Effect of buffer size

Fig. 11. Processing time versus cost distribution and buffer size (top-k)

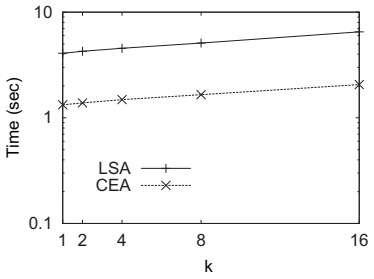


Fig. 12. Processing time versus k (top-k)

growing stage and more candidates need to be considered in the shrinking stage. Since a larger k implies a broader expansion, the multiple-read issue of LSA is exacerbated, leading to 3.4 times longer execution time than CEA (note the exponential scale in the plot).

To summarize the experiments, the search for both skyline and top-k queries is more expensive for sparser networks (in terms of facilities). The processing time also raises with the number of cost types involved. CEA consistently outperforms LSA by a wide margin for both query types. Finally, we must mention that we also experimented with other real road networks, with results similar to the ones presented above. We also tried varying the number of facility clusters; CEA was the clear winner in all cases. The corresponding figures are omitted due to lack of space.

VII. Conclusions

In this paper we introduce the skyline and top-k queries in multi-cost road networks (MCN). Such queries arise in a variety of decision making applications that involve multiple types of coexisting transportation costs. We formalize these queries and design algorithms for their processing. Extensive experiments on real road networks verify the efficiency of our methods.

An interesting direction for future work is to extend our techniques to incrementally updating the skyline or top-k set in the presence of facility/query location updates. Another challenging topic is preference queries in MCNs where the costs of the edges are functions of time (e.g., [30], [31]). Such queries would retrieve preferred (i.e., skyline or top-k) facilities for every time instance within a given period.

References

[1] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," in *VLDB*, 2003, pp. 802–813.

[2] M. L. Yiu and N. Mamoulis, "Clustering objects on a spatial network," in *SIGMOD*, 2004, pp. 443–454.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.

[4] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *J. Comput. Syst. Sci.*, vol. 66, no. 4, pp. 614–656, 2003.

[5] K.-L. Tan, P.-K. Eng, and B. C. Ooi, "Efficient progressive skyline computation," in *VLDB*, 2001, pp. 301–310.

[6] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting: Theory and optimizations," in *IIS*, 2005, pp. 595–604.

[7] P. Godfrey, R. Shipley, and J. Gryz, "Maximal vector computation in large data sets," in *VLDB*, 2005, pp. 229–240.

[8] I. Bartolini, P. Ciaccia, and M. Patella, "Efficient sort-based skyline evaluation," *ACM Trans. Database Syst.*, vol. 33, no. 4, pp. 1–49, 2008.

[9] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: An online algorithm for skyline queries," in *VLDB*, 2002, pp. 275–286.

[10] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 41–82, 2005.

[11] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD*, 1984, pp. 47–57.

[12] W.-T. Balke, U. Güntzer, and J. X. Zheng, "Efficient distributed skylining for web information systems," in *EDBT*, 2004, pp. 256–273.

[13] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou, "Branch-and-bound processing of ranked queries," *Information Systems*, vol. 32, no. 3, pp. 424–445, 2007.

[14] E. W. Dijkstra, "A note on two problems in connection with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[15] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[16] H.-P. Kriegel, P. Kröger, M. Renz, and T. Schmidt, "Hierarchical Graph Embedding for Efficient Query Processing in Very Large Traffic Networks," in *SSDBM*, 2008, pp. 150–167.

[17] S. Jung and S. Pramanik, "An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps," *IEEE TKDE*, vol. 14, no. 5, pp. 1029–1046, 2002.

[18] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable Network Distance Browsing in Spatial Databases," in *SIGMOD*, 2008, pp. 43–54.

[19] H. Hu, D. L. Lee, and V. C. S. Lee, "Distance Indexing on Road Networks," in *VLDB*, 2006, pp. 894–905.

[20] K. Deng, X. Zhou, and H. T. Shen, "Multi-source skyline query processing in road networks," in *ICDE*, 2007, pp. 796–805.

[21] M. Sharifzadeh and C. Shahabi, "The spatial skyline queries," in *VLDB*, 2006, pp. 751–762.

[22] P. Hansen, "Bicriterion path problems," in *Multiple criteria decision making: theory and applications. Lecture notes in Economics and Mathematical Systems*, 1980, pp. 109–127.

[23] E. Q. V. Martins, "On a multicriteria shortest path problem," *European Journal of Operational Research*, vol. 16, no. 2, pp. 236–245, 1984.

[24] A. J. V. Skriver and K. A. Andersen, "A label correcting approach for solving bicriterion shortest-path problems," *Computers & OR*, vol. 27, no. 6, pp. 507–524, 2000.

[25] M. I. Henig, "The shortest path problem with two objective functions," *European Journal of Operational Research*, vol. 25, no. 2, pp. 281–291, 1986.

[26] J. Brumbaugh-Smith and D. Shier, "An empirical investigation of some bicriterion shortest path algorithms," *European Journal of Operational Research*, vol. 43, no. 2, pp. 216–224, 1989.

[27] B. S. Stewart and C. C. White, III, "Multiobjective a*," *J. ACM*, vol. 38, no. 4, pp. 775–814, 1991.

[28] L. Mandow and J.-L. P. de-la Cruz, "A new approach to multiobjective a* search," in *IJCAI*, 2005, pp. 218–223.

[29] T. Brinkhoff, "A framework for generating network-based moving objects," *GeoInformatica*, vol. 6, no. 2, pp. 153–180, 2002.

[30] E. Kanoulas, Y. Du, T. Xia, and D. Zhang, "Finding fastest paths on a road network with speed patterns," in *ICDE*, 2006, p. 10.

[31] B. Ding, J. X. Yu, and L. Qin, "Finding time-dependent shortest paths over large graphs," in *EDBT*, 2008, pp. 205–216.