Memory-Efficient Algorithms for Spatial Network Queries

Sarana Nutanong, Hanan Samet

Center for Automation Research, Institute for Advanced Computer Studies
Department of Computer Science, University of Maryland
College Park, Maryland 20742
{nutanong, hjs}@cs.umd.edu

***Abstract*— Incrementally finding the k nearest neighbors (kNN) in a spatial network is an important problem in location-based services. One method (INE) simply applies Dijkstra's algorithm. Another method (IER) computes the k nearest neighbors using Euclidean distance followed by computing their corresponding network distances, and then incrementally finds the next nearest neighbors in order of increasing Euclidean distance until finding one whose Euclidean distance is greater than the current k nearest neighbor in terms of network distance. The LBC method improves on INE by avoiding the visit of nodes that cannot possibly lead to the k nearest neighbors by using a Euclidean heuristic estimator, and on IER by avoiding the repeated visits to nodes in the spatial network that appear on the shortest paths to different members of the k nearest neighbors by performing multiple instances of heuristic search using a Euclidean heuristic estimator on candidate objects around the query point. LBC's drawback is that the maintenance of multiple instances of heuristic search (called wavefronts) requires k priority queues and the queue operations required to maintain them incur a high in-memory processing cost. A method (SWH) is proposed that utilizes a novel heuristic function which considers objects surrounding the query point together as a single unit, instead of as one destination at a time as in LBC, thereby eliminating the need for multiple wavefronts and needs just one priority queue. These results in a significant reduction in the in-memory processing cost components while having the same reduced cost of the access to the spatial network as LBC. SWH is also extended to support the *incremental distance semi-join (IDSJ)* query, which is a multiple query point generalization of the kNN query. In addition, SWH is shown to support *landmark-based heuristic functions*, thereby enabling it to be applied to non-spatial networks/graphs such as social networks. Comparisons of experiments on SWH for kNN queries with INE, the best single-wavefront method, show that SWH is 2.5 times faster, and with LBC, the best existing heuristic search method, show that SWH is 3.5 times faster. For IDSJ queries, SWH-IDSJ is 5 times faster than INE-IDSJ, and 4 times faster than LBC-IDSJ.**

I. Introduction

The rising popularity of smartphones and their incorporation of a GPS capability has led to an increasing activity in spatial query processing. The functionality has steadily increased ranging from early systems such as QUILT [25], [32] and SAND [24] which had a browsing capability to full-fledged mapping systems such as those from, but not limited to, Google, Microsoft, and Apple where the focus is on location-based services. The most typical queries, and the ones we focus on here, involve finding nearby facilities such as super markets and gas stations from our current location. This requires the computation of distance. Since movement in our everyday life is mostly constrained by network connectivities, the distance between two locations $\boldsymbol{x}$ and $\boldsymbol{y}$ in our work is more accurately represented as the network distance $\textsc{Dist}(\boldsymbol{x}, \boldsymbol{y})$ rather than the Euclidean distance $||\boldsymbol{x} - \boldsymbol{y}||$ [4], [11], [20]. or variants of it such as a minimum distance to a block boundary (e.g., [23], [26]) or the Hausdorff distance (e.g., [17]). For example, given a user location $\boldsymbol{q}$, the cost of driving to a gas station at a location $\boldsymbol{p}$ on the opposite side of a dual-carriage highway will have the network distance $\textsc{Dist}(\boldsymbol{q}, \boldsymbol{p})$ which is much greater than the Euclidean distance $||\boldsymbol{q} - \boldsymbol{p}||$ (i.e., as the crow flies). Since the driver has to follow the shortest path along the highway, $\textsc{Dist}(\boldsymbol{q}, \boldsymbol{p})$ is considered a more meaningful representation of the cost to travel from $\boldsymbol{q}$ to $\boldsymbol{p}$ than the Euclidean counterpart.

In this paper, we study the problem of finding the k nearest neighbors (k NNs) from a query point $\boldsymbol{q}$ in a spatial network [11], [20], e.g., finding the k nearest gas stations in a road network. Due to the interactive nature and the need for high performance of many spatial applications, we focus our algorithm design effort on a setting where the in-memory processing cost is as important as the access cost. Specifically, we are particularly interested in a case where network nodes and edges are stored in a *main memory database system (MMDBS)* [5], [16], which is commonly used in high-performance analytical applications [2], [12]. In particular, designing algorithms for an MMDBS requires a careful consideration of factors such as CPU cost and memory consumption rather than disk access cost [5], [16].

Many methods have been proposed to find the kNNs. The classical method simply applies Dijkstra's algorithm and is the basis of the INE (Incremental Network Expansion) method of Papadias et. al [20]. In particular, INE visits the nodes of the network in order of their increasing network distance from the query object. This means that it is optimal in the sense that it does not visit a node or object that is farther away than the k-th NN. Its drawback is that it does not make use of any heuristic information to prune the exploration of nodes from which the k NNs cannot be possibly reached. Such needless visits are avoided by judicious invocation of the IER (Incremental Euclidean Restriction) method of Papadias et. al [20] which uses a best-first algorithm to find the k NNs in terms of their Euclidean distance (and thus does not use the network), and then computes their corresponding network distance using a shortest path algorithm, one at a time. These k

978-1-4673-4910-9/13/$31.00 © 2013 IEEE

objects form the initial set of candidate k NNs after which the next nearest objects using the Euclidean distance are retrieved incrementally and their network distance computed using a shortest path algorithm until encountering an object whose Euclidean distance is greater than the current k-th nearest object in terms of network distance. The shortest paths can be calculated using the A* algorithm [4].

The main drawback of IER is that computation of the network distance of each candidate object by IER requires reinvocation of the network distance calculation process for all network nodes on the shortest paths to previously encountered objects. This means that some of the parts of the network graph must be accessed repeatedly, as is the case when the shortest paths to the objects have common subgraphs.

The LBC (Lower Bound Constraint) method of Deng et. al [4] improves on both INE and IER by attempting to overcome their drawbacks. In the case of INE, LBC makes use of heuristic information and applies the A* algorithm [8] to calculate shortest paths from the query point to candidate destination objects instead of using Dijkstra's algorithm to incrementally explore network nodes around the query point in a single search, thereby avoiding the visit of nodes that cannot possibly lead to the k nearest neighbors. Specifically, the heuristic cost function is the Euclidean distance from the current node to the destination object, which is guaranteed to be a lower bound on the network distance from the current node to the destination object. The advantage of this approach is that the nodes of the network are now visited in increasing order of the lower bound on the distance to the nearest objects from the query object on a path that passes through them. In particular, this approach is more likely to explore nodes that lie on the paths to the nearest objects than those that do not. In other words, it visits a node $\boldsymbol{n}$ on the basis of an estimated total distance from the query object $\boldsymbol{q}$ to a destination object $\boldsymbol{p}$ on a path through $\boldsymbol{n}$, rather than just on the basis of the distance of $\boldsymbol{n}$ from $\boldsymbol{q}$.

In the case of IER, LBC avoids the repeated access of some parts of the spatial network by computing the shortest paths in terms of network distance to the nearest k Euclidean distance neighbors (i.e., the kNN candidates) in such a way that each node in the network is accessed just once. This is achieved by making each of the shortest path calculations (one per candidate) visit the nodes of the network in the same order. This is possible because for an arbitrary node $\boldsymbol{n}$, the heuristic cost function used by LBC is the minimum of the Euclidean distances to each of the kNN candidates reached through all edges emanating from $\boldsymbol{n}$. The drawback of LBC is that although each node in the graph/network is accessed just once, as in the IER method, it still needs separate priority queues to compute the shortest path to each of the kNN candidates. In essence, LBC can be characterized as making use of multiple wavefronts (corresponding to an instance of heuristic search called A* search), one for each of the kNN candidates, where each wavefront is managed by a priority queue for a total of at least k priority queues. Maintenance of multiple priority queues incurs additional memory space requirement and priority queue insertion/deletion operations, which can be costly in terms of in-memory processing.

In this paper, we formulate algorithms to process spatial queries that are efficient in terms of the access cost as well as in-memory processing costs. We have the following design objectives: First, the search order should be maintained by a single priority queue to reduce the memory consumption and priority queue maintenance cost. Second, we want to make use of heuristic information to minimize the graph/network traversal cost, since a smaller number of nodes to consider generally translates to less memory consumption and a smaller effort to maintain the search order. Third, we want the way in which we compute heuristic values to be computationally inexpensive. In other words, the effort to compute heuristic values should not outweigh the benefit in the reduction of the graph/network traversal cost.

Specifically, we propose the SWH (Single Wavefront Heuristic) which removes the multiple wavefront drawback of LBC thereby significantly reducing its storage requirements and main memory computing costs (priority queue operations) while still having the same reduced cost as LBC for accessing the spatial network. This is done by devising a novel heuristic function that considers destination objects surrounding the query point together as a single unit (relying on previously found nearest neighbors to find nearest neighbors quickly), instead of as one destination at a time as in LBC. Thus only one priority queue is needed instead of a minimum of k.

Experiments for kNN queries show a significant reduction in execution time, as well as the number of priority queue operations and priority queue size, for SWH vis-a-vis LBC. Since IER is superseded by LBC, no comparisons of SWH with IER are necessary, while comparisons of SWH with INE also show SWH to consistently outperform INE. In addition, we also show how to extend SWH to computing the *incremental distance semi-join (IDSJ)* query [9] where instead of just one query object, we have a set Q of query objects for which we obtain a significant improvement for the SWH variant over the LBC and INE variants in the time needed to compute their NNs in order of increasing network distance [28].

It is important to note that an alternative and more drastic approach to reducing the number of visited nodes in the traversal of the spatial network is to use precomputed distance information. For example, we can precompute the distances from each node in the network to a selected subset (called landmarks) [6], [13], and apply the triangle inequality to derive an upper bound and a lower bound on the network distance between any two nodes in the network. Another method called *distance oracle* [29], [30], [31] represents approximate network distances as tuples (X, Y, d) such that for each node $\boldsymbol{x}$ in X and $\boldsymbol{y}$ in Y, $|\textsc{Dist}(\boldsymbol{x}, \boldsymbol{y}) - d|$ is smaller than a predetermined error bound ϵ. Still other notable precomputation-based techniques include: *network Voronoi diagram* [15], [33], which precomputes the NN in a dataset $\mathcal{D}$ for each node in the network, and *shortest path quadtree* [26], [27], which relies on precomputation of $O(n^2)$ network distances, storing them using $O(n^{1.5})$ space, and allowing a shortest path search to be conducted using $O(h)$ lookup operations, where h denotes the

number of hops between the starting and the destination nodes.

All of these precomputation-based techniques significantly reduce the number of visited nodes in problems like shortest path and kNN search. We use the landmark method to demonstrate how our proposed method can make use of precomputed distance information, landmarks distances are easy to compute/store and are applicable to non-spatial networks. It is important to note that the precomputation-based techniques do incur the cost of an additional data structure to store the precomputed information and a computational effort to keep shortest path information up-to-date. Nevertheless, these costs are reasonable for the landmarks method where the number of landmarks is relatively small vis-a-vis the size of the network.

To summarize, the contributions of this paper are as follows: (i) Derivation of a heuristic function especially designed for the kNN query problem and a method to compute this heuristic function efficiently. (ii) Formulation of a novel spatial-network kNN algorithm (SWH), which does not result in visiting additional nodes in the spatial network and is main memory efficient. (iii) Extendsion of SWH to support the IDSJ query. (iv) Proofs of correctness and optimality of the algorithms. (v) Adaptation of SWH to utilize precomputed landmark distances. (vi) Performance evaluations using a real road network and experimental results showing that our proposed algorithm outperforms the two best existing competitors [4], [20] for both kNN and IDSJ queries.

The rest of this paper is organized as follows. Section II describes our proposed algorithm (SWH) for the kNN query, while Section III shows how to adapt it for the incremental distance semi-join (IDSJ) query. Section IV discusses the correctness and optimality of our proposed algorithms, while Section V presents an extension to support landmarks. Results of an experimental evaluation are reported in Section VI. Concluding remarks are drawn in Section VII.

II. Proposed Method

In this section, we present our single-wavefront heuristic (SWH) search algorithm for the kNN query (Definition 1).

Definition 1 (kNN query): Given a set $\mathcal{D}$ of locations and a query point $\boldsymbol{q}$, the kNN query finds a set $\mathcal{A}$ of locations such that (i) $\mathcal{A}$ is a subset of $\mathcal{D}$, (ii) $\mathcal{A}$ contains $\text{MIN}\{|\mathcal{D}|, k\}$ objects; (iii) for each $\boldsymbol{p}$ in $\mathcal{A}$ and $\boldsymbol{r}$ in $(\mathcal{D} \setminus \mathcal{A})$, $\text{DIST}(\boldsymbol{q}, \boldsymbol{p})$ is not greater than $\text{DIST}(\boldsymbol{q}, \boldsymbol{r})$, where $\text{DIST}(\boldsymbol{q}, \boldsymbol{p})$ is defined as the length of the shortest path from $\boldsymbol{q}$ to $\boldsymbol{p}$.

This section is organized as follows. Section II-A defines our heuristic function for kNN search. Section II-B describes the data structures used in the presentation of algorithms in this section. Section II-C presents a method to efficiently compute the heuristic function and presents our proposed algorithm which incorporates the heuristic computation method.

A. Heuristic Function

Our heuristic function is based on the observation that the problem of finding the NN in $\mathcal{D}$ with respect to $\boldsymbol{q}$ in a network is equivalent to computing the shortest path from $\boldsymbol{q}$ to the NN in the dataset $\mathcal{D}$. To compute the shortest path from $\boldsymbol{q}$ to a destination $\boldsymbol{p}$, for each network node $\boldsymbol{n}$ encountered by the search, an optimistic estimate of the remaining distance from $\boldsymbol{n}$ to the destination $\boldsymbol{p}$ is given by the heuristic function: $h(\boldsymbol{n}, \boldsymbol{p}) = ||\boldsymbol{n} - \boldsymbol{p}||$. By replacing the target $\boldsymbol{p}$ with a target location set $\mathcal{D}$, we compute an optimistic estimate of the distance from a network node $\boldsymbol{n}$ to the dataset $\mathcal{D}$ as the minimum Euclidean distance from $\boldsymbol{q}$ to all points in $\mathcal{D}$. That is, $h(\boldsymbol{n}, \mathcal{D}) = \text{MIN}\{||\boldsymbol{n} - \boldsymbol{p}|| : \boldsymbol{p} \in \mathcal{D}\}$.

The same concept can be extended to help find subsequent k NNs. Specifically, we can decompose the problem of finding the k NNs in $\mathcal{D}$ into finding the first NN $\boldsymbol{p_1}$ in $\mathcal{D}$, and finding the i-th NN $\boldsymbol{p_i}$ in $(\mathcal{D} \setminus \{\boldsymbol{p_1}, ..., \boldsymbol{p_{i-1}}\})$. In other words, we constantly update the dataset in which we search for the next result as NNs are incrementally discovered. Hence, when searching for the i-th NN, we have to remove the first $(i-1)$ NNs from $\mathcal{D}$. That is, $h(\boldsymbol{n}, \mathcal{D} \setminus \mathcal{A})$ is $\text{MIN}\{||\boldsymbol{n} - \boldsymbol{p}|| : \boldsymbol{p} \in \mathcal{D} \setminus \mathcal{A}\}$, where $\mathcal{A}$ denotes a list containing the first $(i-1)$ NNs.

B. Data Structure

We use a directed graph to represent a road network containing unidirectional components. As shown in Figure 1, a one-way street is represented as a unidirectional edge, a two-way street is represented as a bidirectional edge, and a dual carriage highway is represented as two opposing unidirectional edges. The figure also shows $\boldsymbol{n_3}$ with four immediate neighbors $\boldsymbol{n_1}$, $\boldsymbol{n_2}$, $\boldsymbol{n_4}$ and $\boldsymbol{n_5}$. Each of these neighbors can be *outgoing*, *incoming*, or both. For example, $\boldsymbol{n_1}$ is an outgoing neighbors of $\boldsymbol{n_3}$; $\boldsymbol{n_5}$ is an incoming neighbors of $\boldsymbol{n_3}$; $\boldsymbol{n_2}$ and $\boldsymbol{n_4}$ belong to both types. We use the notations "$\boldsymbol{n}$.OutgoingNodes()" and "$\boldsymbol{n}$.IncomingNodes()" to denote the outgoing-node set and the incoming-node set of $\boldsymbol{n}$, respectively.

For ease of exposition, data objects are treated as network nodes. That is, the dataset $\mathcal{D}$ is a subset of the set of nodes in the network. The same principle presented in this paper is still applicable to cases where data objects are treated separately.

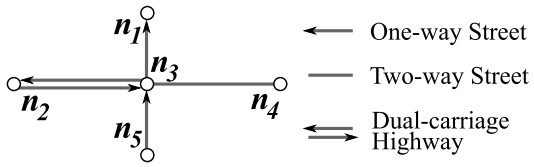


Fig. 1. Graph with 5 nodes, 1 bidirectional edge and 4 unidirectional edges.

We now describe the priority queue used to provide search order in our kNN algorithms (Algorithm 2). A priority queue is used to make sure that we always process a node with the smallest estimated distance first. Specifically, the structure organizes its entries so that the one with the smallest key value is always at the first/top element of the structure. Each entry consists of the following five attributes. (i) Distance d_h (the sorting key): an optimistic estimate of the total cost from $\boldsymbol{q}$ to the next NN for this priority queue entry. (ii) Distance d: a network distance from $\boldsymbol{q}$ to $\boldsymbol{n}$ (via $\boldsymbol{n_p}$). (iii) Node $\boldsymbol{n}$: the node to which this entry corresponds. (iv) Node $\boldsymbol{n_p}$: the previous node on the path from $\boldsymbol{q}$. (v) Object $\boldsymbol{x}$: the Euclidean NN of $\boldsymbol{n}$ in $(\mathcal{D} \setminus \mathcal{A})$. The distance d is computed as the sum of the (already obtained) network distance $\text{DIST}(\boldsymbol{q}, \boldsymbol{n_p})$ and the weight (e.g., length) of the edge that links $\boldsymbol{n_p}$ to $\boldsymbol{n}$. The

distance d_h is computed as the sum of the distance d and the estimated remaining distance $h(\boldsymbol{n}, \mathcal{D} \setminus \mathcal{A})$. Each priority queue entry is expressed as $(d_h, d, \boldsymbol{n}, \boldsymbol{n_p}, \boldsymbol{x})$ in Algorithm 2.

C. Proposed kNN Algorithm

1) Heuristic computation: Computing $h(\boldsymbol{n}, \mathcal{D} \setminus \mathcal{A})$ is considered a challenge because it requires evaluation of the Euclidean NN [10], [22] of $\boldsymbol{n}$ in the set $(\mathcal{D} \setminus \mathcal{A})$. Due to the large number of nodes $\boldsymbol{n}$ for which we need to compute $h(\boldsymbol{n}, \mathcal{D} \setminus \mathcal{A})$, it is impractical to compute the heuristic value from scratch each time $\boldsymbol{n}$ is changed even if $(\mathcal{D} \setminus \mathcal{A})$ is indexed in a hierarchical structure [1], [7]. In this subsection, we derive a novel solution which enables sharing of NN results among nodes in proximity.

Specifically, our solution utilizes the best-first NN algorithm [10] to incrementally retrieve Euclidean NNs with respect to the query point $\boldsymbol{q}$. The Euclidean NN with respect to each node encountered by our search algorithm is computed from the Euclidean NNs of $\boldsymbol{q}$. As a result, evaluation of $h(\boldsymbol{n}, \mathcal{D} \setminus \mathcal{A})$ requires consideration of only a small number of Euclidean NNs with respect to $\boldsymbol{q}$ rather than iterating through every single point in $(\mathcal{D} \setminus \mathcal{A})$.

The intuition behind our technique is that we represent the set $\mathcal{D}$ using a smaller subset of objects and the "coverage" of this subset with respect to a reference point (which is the query point $\boldsymbol{q}$ in this case). Specifically, we use a Euclidean NN search to retrieve objects around $\boldsymbol{q}$ and insert them into a list $\mathcal{C}$ of Euclidean NNs with respect to $\boldsymbol{q}$. As a result, the coverage of this list is the disc centered at $\boldsymbol{q}$ that minimally encloses all objects in $\mathcal{C}$, i.e.,

$$\{\boldsymbol{v} : ||\boldsymbol{q} - \boldsymbol{v}|| \leq \text{MAX}\{||\boldsymbol{q} - \boldsymbol{p}|| : \boldsymbol{p} \in \mathcal{C}\}\}.$$

We call this disc the *known region* and we use r to denote its radius $\text{MAX}\{||\boldsymbol{q} - \boldsymbol{p}|| : \boldsymbol{p} \in \mathcal{C}\}$. The Euclidean NN in $(\mathcal{D} \setminus \mathcal{A})$ of any node $\boldsymbol{n}$ can be computed using a much smaller collection of objects in $(\mathcal{C} \setminus \mathcal{A})$ if the known region is large enough to guarantee the correctness of the result. Otherwise, we have to expand the known region by retrieving more Euclidean NNs with respect to $\boldsymbol{q}$. The details of this evaluation process is given as follows.

In order to compute $h(\boldsymbol{n}, \mathcal{D} \setminus \mathcal{A})$, we find the nearest object $\boldsymbol{x}$ in $(\mathcal{C} \setminus \mathcal{A})$ and obtain the Euclidean distance $||\boldsymbol{n} - \boldsymbol{x}||$. Next, we check if it is possible to have any object $\boldsymbol{y}$ outside the known region that could invalidate $\boldsymbol{x}$ as the object which minimizes the Euclidean distance from $\boldsymbol{n}$. That is, $||\boldsymbol{n} - \boldsymbol{y}|| < ||\boldsymbol{n} - \boldsymbol{x}||$. We call this process a *reliability check*. To perform this reliability check, we adopt the concept of *safe region with respect to a data object* [18]. That is, given a known region with a center of $\boldsymbol{q}$ and a radius of r, we can guarantee that $\boldsymbol{n}$ is closer to $\boldsymbol{x}$ than any object outside the known region if $\boldsymbol{n}$ is inside the safe region

$$S(\boldsymbol{q}, \boldsymbol{x}, r) = \{\boldsymbol{v} : ||\boldsymbol{v} - \boldsymbol{x}|| + ||\boldsymbol{q} - \boldsymbol{v}|| \leq r\}. \quad (1)$$

If this check fails, then we have to keep expanding the known region to increase the r value and to take more objects into consideration until this condition is satisfied. Formalization of the described process is given by Algorithm 1.

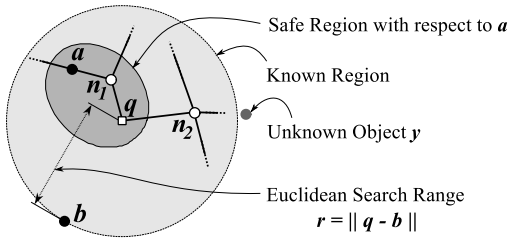


Fig. 2. Safe region with respect to Object $\boldsymbol{a}$.

For a better illustration of this concept, we provide an example in Figure 2. The figure shows a query point $\boldsymbol{q}$ adjacent to two network nodes $\boldsymbol{n_1}$ and $\boldsymbol{n_2}$. In this example, we wish to compute $h(\boldsymbol{n_1}, \mathcal{D} \setminus \mathcal{A})$ and $h(\boldsymbol{n_2}, \mathcal{D} \setminus \mathcal{A})$ to determine the order in which these nodes are accessed. Assume that $\mathcal{D}$ contains $\{\boldsymbol{a}, \boldsymbol{b}, ..., \boldsymbol{z}\}$ and the answer set $\mathcal{A}$ is currently empty. The figure shows that two objects $\boldsymbol{a}$ and $\boldsymbol{b}$ are retrieved as the first two Euclidean NNs with respect to $\boldsymbol{q}$. Hence, the list $\mathcal{C}$ is given as $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$. The known region is shown as the gray disc centered at $\boldsymbol{q}$ with a radius r of $||\boldsymbol{q} - \boldsymbol{b}||$.

Algorithm 1: EstDist$(\boldsymbol{n}, \boldsymbol{q}, \mathcal{C}, r, \mathcal{A})$

input : (i) Node $\boldsymbol{n}$, (ii) Query point $\boldsymbol{q}$, (iii) List $\mathcal{C}$ of Euclidean NNs, (iv) Scope r of the Euclidean NN Search, (v) Set $\mathcal{A}$ of network NNs obtained so far.
modified input: $\mathcal{C}$, r
output : Pair $(||\boldsymbol{n} - \boldsymbol{x}||, \boldsymbol{x})$ where $||\boldsymbol{n} - \boldsymbol{x}||$ is equal to $\text{MIN}\{||\boldsymbol{n} - \boldsymbol{p}|| : \boldsymbol{p} \in \mathcal{C}\}$.
environment : Dataset $\mathcal{D}$

```
1  Object x ← Null;
2  Distance d_min ← ∞;
3  for each p in (C \ A) do
4      Distance d ← ||n − p||;
5      if d < d_min then
6          x ← p;
7          d_min ← d;
8  while d_min > r − ||q − n|| do
9      (p, r) ← NextEuclideanNN(q, D);
10     Insert(p, C);
11     Distance d ← ||n − p||;
12     if d < d_min then
13         x ← p;
14         d_min ← d;
15 return (d_min, x);
```

Evaluation of $h(\boldsymbol{n_1}, \mathcal{D} \setminus \mathcal{A})$ is conducted as follows. First, we iterate through the objects in $(\mathcal{C} \setminus \mathcal{A})$ to find the minimum distance and the nearest object $(d_{min}, \boldsymbol{x})$ (Lines 3 to 7) with respect to $\boldsymbol{n_1}$. As a result, we obtain $(||\boldsymbol{n_1} - \boldsymbol{a}||, \boldsymbol{a})$ in this case. Second, we check the reliability of $\boldsymbol{a}$, i.e., if it is possible to have any unknown object closer to $\boldsymbol{n_1}$ than $\boldsymbol{a}$ (Line 8). This is done by testing if $\boldsymbol{n_1}$ is inside the safe region $S(\boldsymbol{q}, \boldsymbol{a}, r)$, which is depicted as the gray elliptic region in the figure. As can be seen, $\boldsymbol{n_1}$ is in fact inside $S(\boldsymbol{q}, \boldsymbol{a}, r)$, and hence $h(\boldsymbol{n_1}, \mathcal{D} \setminus \mathcal{A})$ is equal to $||\boldsymbol{n_1} - \boldsymbol{a}||$.

To calculate $h(\boldsymbol{n_2}, \mathcal{D} \setminus \mathcal{A})$, we follow the same steps and first determine that $\boldsymbol{a}$ is the closest object among objects in $\mathcal{C}$ after the for loop (Lines 3 to 7). Next, we check if $\boldsymbol{a}$ is

reliable with respect to $\boldsymbol{n_2}$ (Line 8). As can be seen, $\boldsymbol{n_2}$ is outside $S(\boldsymbol{q}, \boldsymbol{a}, r)$. This condition implies that we need more information in order to evaluate $h(\boldsymbol{n_2}, \mathcal{D} \setminus \mathcal{A})$. This is because, $\boldsymbol{n_2}$ could be closer to an unknown object $\boldsymbol{y}$ (depicted as a gray dot outside the known region) than $\boldsymbol{a}$. Hence, we need to retrieve more Euclidean NNs with respect to $\boldsymbol{q}$ to expand the search range r, and to take more objects into consideration until the condition at Line 8 is broken.

2) Algorithm description: Our proposed algorithm (Algorithm 2) traverses the graph/network in best-first order according to the heuristic values produced by Algorithm 1 (Lines 11 and 22), and is described as follows. The algorithm accepts a dataset $\mathcal{D}$, a query point $\boldsymbol{q}$ and the desired number of kNN results. The initialization (Lines 1 to 7) include the following steps: creating a priority queue *PQ*, inserting the initial priority queue entry into *PQ*, creating a hash table *VisitedNodes* to store the shortest path information from $\boldsymbol{q}$ for each visited node, and creating an empty set $\mathcal{A}$ to store k NNs, create a list of NN candidates and insert the first Euclidean NN of $\boldsymbol{q}$ as the first candidate. The heuristic best-first search is conducted in the control loop (Lines 8 to 23). In the beginning of each iteration, the head entry $(d_h, d, \boldsymbol{n}, \boldsymbol{n_p}, \boldsymbol{x})$ is retrieved from *PQ*. The rest of the loop consists of the following parts: (i) heuristic update and validation (Lines 10 to 13), (ii) labeling and result check (Lines 15 to 19), (iii) expansion of neighboring nodes (Lines 20 to 23). To provide a clearer illustration of how Algorithm 2 operates, we provide a sample execution using the example in Figure 3. Table I shows the values of the key variables for each iteration of the control loop (Lines 8 to 23). After the initialization process (Lines 1 to 4), the priority queue *PQ* contains a single entry $(-, 0, \boldsymbol{q}, -, -)$ created from $\boldsymbol{q}$.

In the first iteration $(i = 1)$, the priority queue entry corresponding to $\boldsymbol{q}$ is retrieved from *PQ*. At this point, we expand the search to the adjacent nodes ($\boldsymbol{n_2}$ and $\boldsymbol{n_6}$) of $\boldsymbol{q}$ by creating priority queue entries for them. To provide the order in which these two entries are processed, we need to compute the heuristic value for each of them. The heuristic value d_h for $\boldsymbol{n_2}$ is evaluated as the sum of (i) the distance d from $\boldsymbol{q}$ so far (which is still 0 in this case), (ii) the length of the edge that connects $\boldsymbol{q}$ to $\boldsymbol{n_2}$, and (iii) the euclidean distance between $\boldsymbol{n_2}$ and $\boldsymbol{b}$, which is nearest object in $\{\boldsymbol{a}$,$\boldsymbol{b}$,$\boldsymbol{c}$,$\boldsymbol{d}\}$ from $\boldsymbol{b}$. This yields a distance of 2, $(0+1+1)$ units. By following the same process, we obtain the key value of $\boldsymbol{n_6}$ as 4, $(0+2+2)$ units. New priority queue entries with these key values are inserted into *PQ*.

In the second iteration $(i = 2)$, $\boldsymbol{n_2}$ is identified as the node with the smallest key. We now consider the adjacent nodes that have never been visited before, i.e., $\boldsymbol{n_1}$ and $\boldsymbol{b}$. Using the same process described in the first iteration, the key value d_h of $\boldsymbol{b}$ and $\boldsymbol{n_1}$ are obtained as 2, $(1+1+0)$, and 7.16, $(1+3+3.16)$, units. Entries with corresponding nodes and key values are then inserted into *PQ*. The same process continues until two NNs are discovered in the fifth iteration as shown in Table I.

Table II provide the details on how heuristic values are computed using Algorithm 1 in the context of the running example. The iteration number i in the first column corresponds to that of Table I. We now consider the first iteration where we compute estimates for $\boldsymbol{n_2}$ and $\boldsymbol{n_6}$. To compute an estimate for $\boldsymbol{n_2}$, we find the object in $\mathcal{C}$ which minimizes the Euclidean distance from $\boldsymbol{n_2}$, which is $\boldsymbol{b}$ in this case. We now check if $\boldsymbol{n_2}$ is in the safe region $S(\boldsymbol{q}, \boldsymbol{b}, r)$ by checking the condition

$$||\boldsymbol{n_2} - \boldsymbol{b}|| + ||\boldsymbol{q} - \boldsymbol{n_2}|| \leq r.$$

This condition fails because we have the distance of (1+1) units for $(||\boldsymbol{n_2} - \boldsymbol{b}|| + ||\boldsymbol{q} - \boldsymbol{n_2}||)$ and the r value of 1.41 units. Consequently, we retrieve the next Euclidean NN of $\boldsymbol{q}$, i.e., $\boldsymbol{a}$. After considering the new object $\boldsymbol{a}$, the minimum distance is unchanged but the r value becomes 2.83 units. As a result, we obtain $(||\boldsymbol{n_2} - \boldsymbol{b}|| = 1, \boldsymbol{b})$ as the estimate and the nearest object for $\boldsymbol{n_2}$, respectively.

Algorithm 2: SWH-kNN($\mathcal{D}$, $\boldsymbol{q}$, k)

input : Dataset $\mathcal{D}$, Query Point $\boldsymbol{q}$, Number k of NNs
output : Set $\mathcal{A}$ of k NNs in $\mathcal{D}$ with respect to $\boldsymbol{q}$
environment: Dataset $\mathcal{D}$ (also input)

```
1  Initialize Priority Queue PQ;
2  Insert(PQEntry (0, 0, q, −, −), PQ);
3  HashTable VisitedNodes ← Create an empty hash table;
4  Set A ← Create an empty set;
5  List C ← Create an empty list;
6  (Object p, Distance r) ← FirstEuclideanNN(q, D);
7  Insert(p, C);
8  while PQ is not empty do
9      PQEntry (d_h, d, n, n_p, x) ← DequeueHead(PQ);
10     if Object x is in A and PQ is not empty then
11         (Distance d_e, Object x) ← EstDist(n, q, C, r, A);
12         if d + d_e > TopKeyOf(PQ) then
13             Insert(PQEntry (d + d_e, d, n, n_p, x), PQ);
14     else if Node n is not in VisitedNodes then
15         Insert(Pair (IdOf(n), (d, n_p)), VisitedNodes);
16         if n is a member of D then
17             Insert(n, A);
18             if |A| is equal to k or |A| is equal to |D| then
19                 return A
20         for each n_a in OutgoingNodes(n) and n_a is not in
           VisitedNodes do
21             Distance d_a ← d+ WeightOf(EDGE(n_a,n));
22             (Distance d_e, Object x) ←
               EstDist(n_a, q, C, r, A);
23             Insert(PQEntry (d_a + d_e, d_a, n_a, n, x), PQ);
24 return A;
```

The next step is to repeat the same process to compute $||\boldsymbol{n_6} - \boldsymbol{a}||$. After iterating through the list $\mathcal{C}$, we obtain $\boldsymbol{a}$ and the distance $||\boldsymbol{n_6} - \boldsymbol{a}||$ of 2 units as our tentative results. We now check if $\boldsymbol{a}$ is reliable with respect to $\boldsymbol{n_6}$ using the condition $||\boldsymbol{n_6} - \boldsymbol{a}|| + ||\boldsymbol{q} - \boldsymbol{n_6}|| \leq r$.

Since we obtain (2+2) units on the left hand side and 2.83 units on the other, we need to expand the known region. The next Euclidean NN of $\boldsymbol{q}$ is $\boldsymbol{c}$ which has the Euclidean distance r from $\boldsymbol{q}$ of 5.10 units. As we can see, $\boldsymbol{a}$ is still the nearest object in $\mathcal{C}$ for $\boldsymbol{n_6}$. Hence, the pair $(||\boldsymbol{n_6} - \boldsymbol{a}||, \boldsymbol{a})$ is returned from the function call $\text{EstDist}(\boldsymbol{n_2}, \boldsymbol{q}, \mathcal{C}, r, \mathcal{A})$. The remaining calculation steps are given in Table II.

TABLE I

EXAMPLE RUN OF ALGORITHM 2 WITH $\mathcal{D}$ OF $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}\}$, AND k OF 2.

i	PQ Entry	$\mathcal{A}$	Inserted Entries	Priority Queue PQ
1	$(-, 0, \boldsymbol{q}, -, -)$	$\langle\rangle$	$(2, 1, \boldsymbol{n_2}, \boldsymbol{q}, \boldsymbol{b})$, $(4, 2, \boldsymbol{n_6}, \boldsymbol{q}, \boldsymbol{a})$	$\langle(2, 1, \boldsymbol{n_2}, \boldsymbol{q}, \boldsymbol{b}), (4, 2, \boldsymbol{n_6}, \boldsymbol{q}, \boldsymbol{a})\rangle$
2	$(2, 1, \boldsymbol{n_2}, \boldsymbol{q}, \boldsymbol{b})$	$\langle\rangle$	$(2, 2, \boldsymbol{b}, \boldsymbol{n_2}, \boldsymbol{b})$,$(7.16, 4, \boldsymbol{n_1}, \boldsymbol{n_2}, \boldsymbol{a})$	$\langle(2, 2, \boldsymbol{b}, \boldsymbol{n_2}, \boldsymbol{b}), (4, 2, \boldsymbol{n_6}, \boldsymbol{q}, \boldsymbol{a}), (7.16, 4, \boldsymbol{n_1}, \boldsymbol{n_2}, \boldsymbol{a})\rangle$
3	$(2, 2, \boldsymbol{b}, \boldsymbol{n_2}, \boldsymbol{b})$	$\langle\boldsymbol{b}\rangle$	$(6, 4, \boldsymbol{n_3}, \boldsymbol{b}, \boldsymbol{c})$	$\langle(4, 2, \boldsymbol{n_6}, \boldsymbol{q}, \boldsymbol{a}), (6, 4, \boldsymbol{n_3}, \boldsymbol{b}, \boldsymbol{c}), (7.16, 4, \boldsymbol{n_1}, \boldsymbol{n_2}, \boldsymbol{a})\rangle$
4	$(4, 2, \boldsymbol{n_6}, \boldsymbol{q})$	$\langle\boldsymbol{b}\rangle$	$(4, 4, \boldsymbol{a}, \boldsymbol{n_6}, \boldsymbol{a})$, $(8.60, 5, \boldsymbol{n_7}, \boldsymbol{n_6}, \boldsymbol{c})$	$\langle(4, 4, \boldsymbol{a}, \boldsymbol{n_6}, \boldsymbol{a}), (6, 4, \boldsymbol{n_3}, \boldsymbol{b}, \boldsymbol{c}), (7.16, 4, \boldsymbol{n_1}, \boldsymbol{n_2}, \boldsymbol{a}), (8.60, 5, \boldsymbol{n_7}, \boldsymbol{n_6}, \boldsymbol{c})\rangle$
5	$(4, 4, \boldsymbol{a}, \boldsymbol{n_6}, \boldsymbol{a})$	$\langle\boldsymbol{b}, \boldsymbol{a}\rangle$	-	-

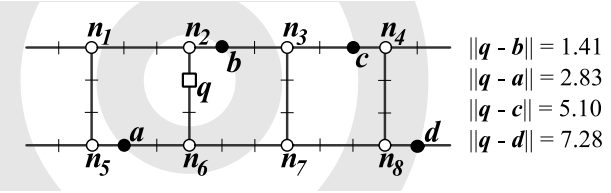


Fig. 3. Evaluation of $h(\boldsymbol{n}, \mathcal{D} \setminus \mathcal{A})$ using ESTDIST().

TABLE II

EXAMPLE RUN OF ALGORITHM 1.

i	$\boldsymbol{n_a}$	$\mathcal{A}$	Input $(\mathcal{C}, r)$	Modified $(\mathcal{C}, r)$	d_e	$\boldsymbol{x}$
1	$\boldsymbol{n_2}$	$\langle\rangle$	$(\langle\boldsymbol{b}\rangle, 1.41)$	$(\langle\boldsymbol{b}, \boldsymbol{a}\rangle, 2.83)$	1	$\boldsymbol{b}$
	$\boldsymbol{n_6}$	$\langle\rangle$	$(\langle\boldsymbol{b}, \boldsymbol{a}\rangle, 2.83)$	$(\langle\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{c}\rangle, 5.10)$	2	$\boldsymbol{a}$
2	$\boldsymbol{b}$	$\langle\rangle$	$(\langle\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{c}\rangle, 5.10)$	$(\langle\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{c}\rangle, 5.10)$	0	$\boldsymbol{b}$
	$\boldsymbol{n_1}$	$\langle\rangle$	$(\langle\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{c}\rangle, 5.10)$	$(\langle\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{d}\rangle, 7.28)$	3.16	$\boldsymbol{a}$
3	$\boldsymbol{n_3}$	$\langle\boldsymbol{b}\rangle$	$(\langle\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{d}\rangle, 7.28)$	$(\langle\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{d}\rangle, 7.28)$	2	$\boldsymbol{c}$
4	$\boldsymbol{a}$	$\langle\boldsymbol{b}\rangle$	$(\langle\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{d}\rangle, 7.28)$	$(\langle\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{d}\rangle, 7.28)$	0	$\boldsymbol{a}$
	$\boldsymbol{n_7}$	$\langle\boldsymbol{b}\rangle$	$(\langle\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{d}\rangle, 7.28)$	$(\langle\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{d}\rangle, 7.28)$	3.60	$\boldsymbol{c}$
5	-	$\langle\boldsymbol{b}, \boldsymbol{a}\rangle$	-	-	-	-

We can see how we avoid invoking the Euclidean NN query for each network node $\boldsymbol{n}$ visited by exploiting spatial locality of references of the Euclidean NN query. Specifically, the NN of each node $\boldsymbol{n}$ is computed by identifying the nearest object in a list $\mathcal{C}$ which is smaller than the dataset $\mathcal{D}$. This benefit becomes more obvious in a setting with a larger dataset $\mathcal{D}$.

III. INCREMENTAL DISTANCE SEMI JOIN

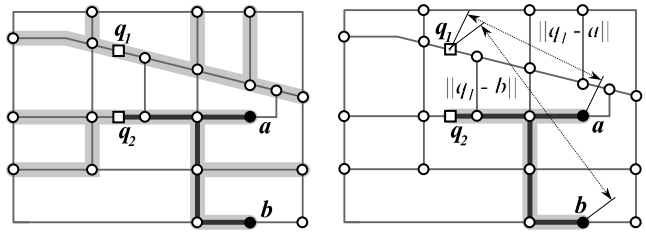
We have shown in the previous section how our proposed technique, SWH, is used in a single query point setting. In this section, we consider a generalized case of multiple query locations. That is, given a set Q of query points, find k points $\boldsymbol{p}$ in $\mathcal{D}$ with smallest minimum distances $\text{MIN}\{\text{DIST}(\boldsymbol{q}, \boldsymbol{p}) : \boldsymbol{q} \in Q\}$. This query type is known as *the distance semi join query*, which can be defined formally as follows.

Definition 2 (Distance Semi-join): Given two sets Q and $\mathcal{D}$ of locations, the *distance semi-join* of Q and $\mathcal{D}$ is a set of pairs $(\boldsymbol{q}, \boldsymbol{p})$ such that (i) $\boldsymbol{q}$ is a member of Q, (ii) $\boldsymbol{p}$ is a member of $\mathcal{D}$, and (iii) $\boldsymbol{p}$ is the NN of $\boldsymbol{q}$.

The *incremental distance semi-join (IDSJ)* query [9], [34], [36] incrementally retrieves these pairs $(\boldsymbol{q}, \boldsymbol{p})$ in ascending order of the distance $\text{DIST}(\boldsymbol{q}, \boldsymbol{p})$. Based on this definition, we can consider IDSJ as the incremental NN query with a query object of Q, a dataset of $\mathcal{D}$ and the distance from the query object Q to a location $\boldsymbol{p}$ in $\mathcal{D}$ of $\text{MIN}\{\text{DIST}(\boldsymbol{q}, \boldsymbol{p}) : \boldsymbol{q} \in Q\}$.

An example query scenario can be given as follows. Given a logistics company with a set W of warehouses and a set C of customers' locations, the logistics company wishes to find k customers nearest to any given warehouse in W to give them discount offers. In this scenario, the IDSJ query can be used to incrementally find the k locations c in C of the customers with the smallest distances from the nearest warehouse $\boldsymbol{w}$ in W.

Two examples of how an IDSJ query can be processed are given in Figure 4. In the first example (Figure 4(a)), we can perform a distance scan and build shortest path trees [19] with respect to the query set $\{\boldsymbol{q_1}, \boldsymbol{q_2}\}$. In this way, each object $\boldsymbol{p}$ in $\{\boldsymbol{a}, \boldsymbol{b}\}$ is associated with its nearest query point $\boldsymbol{q}$ in $\{\boldsymbol{q_1}, \boldsymbol{q_2}\}$. Similar to the uninformed kNN search, INE-kNN, this method performs distance scans with respect to each query point in the query set Q. In order to retrieve a pair $(\boldsymbol{q_i}, \boldsymbol{p_k})$, the search traverses all nodes $\boldsymbol{n}$ such that the minimum distance $\text{MIN}\{\text{DIST}(\boldsymbol{q}, \boldsymbol{p}) : \boldsymbol{q} \in Q\}$ is smaller than $\text{DIST}(\boldsymbol{q_i}, \boldsymbol{p_k})$. As shown in Figure 4(a), we need to explore nodes around $\boldsymbol{q_1}$ although none of the data objects is associated with $\boldsymbol{q_1}$.



(a) Uninformed Search (b) Informed (Heuristic) Search

Fig. 4. Illustration of distance semi-join of $\{\boldsymbol{a}, \boldsymbol{b}\}$ and $\{\boldsymbol{q_1}, \boldsymbol{q_2}\}$ with resultant pairs of $(\boldsymbol{q_2}, \boldsymbol{a})$ and $(\boldsymbol{q_2}, \boldsymbol{b})$.

Figure 4(b) shows that we can tremendously reduce the search space by using the Euclidean distance as a lower bound. In this example, we see that Euclidean distances $||\boldsymbol{q_1} - \boldsymbol{a}||$ and $||\boldsymbol{q_1} - \boldsymbol{b}||$ makes $\boldsymbol{q_1}$ highly undesirable in comparison to $\boldsymbol{q_2}$. As a result, we favor expansion around $\boldsymbol{q_2}$ and in this case, the results $\langle(\boldsymbol{q_2}, \boldsymbol{a}), (\boldsymbol{q_2}, \boldsymbol{b})\rangle$ can be produced without having to consider nodes around $\boldsymbol{q_1}$ at all. Our experimental results demonstrate the efficiency of our IDSJ extension to the single-wavefront concept in terms of the graph traversal cost and memory consumption.

IV. ANALYSIS

A. Correctness and optimality

Dechter and Pearl [3] show that the correctness and optimality of A* search depends on the *consistency* of the heuristic function, i.e., whether the function satisfies the triangle inequality. Since our proposed algorithm (Algorithm 2) relies on the same best-first heuristic search principle as A* search, we prove that $h(\boldsymbol{n}, \mathcal{D})$ is a consistent and admissible heuristic function as follows.

Lemma 1 (Consistency and Admissibility): The heuristic function $h(\boldsymbol{n}, \mathcal{D})$ is consistent and admissible. That is,

(i) **consistent:** $h(\boldsymbol{n}, \mathcal{D}) \leq \text{DIST}(\boldsymbol{n}, \boldsymbol{n'}) + h(\boldsymbol{n'}, \mathcal{D})$; and

(ii) **admissible:** $h(\boldsymbol{n}, \mathcal{D}) \leq \text{MIN}\{\text{DIST}(\boldsymbol{n}, \boldsymbol{p}) : \boldsymbol{p} \in \mathcal{D}\}$.

Proof: Let $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ be objects in $\mathcal{D}$ which minimize the Euclidean distance from $\boldsymbol{n}$ and $\boldsymbol{n'}$, respectively. That is

$$||\boldsymbol{n} - \boldsymbol{p_1}|| = h(\boldsymbol{n}, \mathcal{D}) \quad \textbf{and} \quad ||\boldsymbol{n'} - \boldsymbol{p_2}|| = h(\boldsymbol{n'}, \mathcal{D})$$

According to the triangle inequality,

$$||\boldsymbol{n} - \boldsymbol{p_2}|| \leq ||\boldsymbol{n} - \boldsymbol{n'}|| + ||\boldsymbol{n'} - \boldsymbol{p_2}||.$$

Since $\boldsymbol{p_1}$ has the smallest Euclidean distance from $\boldsymbol{n}$ and the Euclidean distance is a lower bound of the network distance, we can replace $||\boldsymbol{n} - \boldsymbol{p_2}||$ by $||\boldsymbol{n} - \boldsymbol{p_1}||$ and $||\boldsymbol{n} - \boldsymbol{n'}||$ by $\text{DIST}(\boldsymbol{n}, \boldsymbol{n'})$, respectively. As a result, we have

$$||\boldsymbol{n} - \boldsymbol{p_1}|| \leq \text{DIST}(\boldsymbol{n}, \boldsymbol{n'}) + ||\boldsymbol{n'} - \boldsymbol{p_2}||,$$

which is equivalent to the consistency condition given in the definition. Hence, $h(\boldsymbol{n}, \mathcal{D})$ is consistent.

Now, let us consider a special case where $\text{DIST}(\boldsymbol{n}, \boldsymbol{n'})$ is $\text{MIN}\{\text{DIST}(\boldsymbol{n}, \boldsymbol{p}) : \boldsymbol{p} \in \mathcal{D}\}$. We have

$$h(\boldsymbol{n}, \mathcal{D}) \leq \text{MIN}\{\text{DIST}(\boldsymbol{n}, \boldsymbol{p}) : \boldsymbol{p} \in \mathcal{D}\} + 0.$$

Hence, the consistency ensures that $h(\boldsymbol{n}, \mathcal{D})$ is admissible. That is, it cannot overestimate the network distance from $\boldsymbol{n}$ to the nearest object in $\mathcal{D}$ [3], [21]. ■

Since we keep updating the target $(\mathcal{D} \setminus \mathcal{A})$, we must ensure that the heuristic function is still admissible after a change in $(\mathcal{D} \setminus \mathcal{A})$. We use the principle of *incremental/adaptive heuristic* search [14], [35] and show in Lemma 2 that an estimate computed for a previous target can never overestimate the distance from $\boldsymbol{n}$ to the current target.

Lemma 2 (Incremental Admissibility): Let $\boldsymbol{p_1}$ denote the network NN of $\boldsymbol{q}$ in $\mathcal{D}$ and $\boldsymbol{p_2}$ denote the network NN of $\boldsymbol{q}$ in $(\mathcal{D} \setminus \{\boldsymbol{p_1}\})$. For any node $\boldsymbol{n}$ in the spatial network, $h(\boldsymbol{n}, \mathcal{D}) \leq \text{DIST}(\boldsymbol{n}, \boldsymbol{p_2})$. That is, the estimate $h(\boldsymbol{n}, \mathcal{D})$ for the first NN $\boldsymbol{p_1}$ is also optimistic with respect to the second NN $\boldsymbol{p_2}$.

Proof: Since $\boldsymbol{p_2}$ is a member of $\mathcal{D}$ and the Euclidean distance is a lower bound of the network distance,

$$h(\boldsymbol{n}, \mathcal{D}) \leq ||\boldsymbol{n} - \boldsymbol{p_2}|| \leq \text{DIST}(\boldsymbol{n}, \boldsymbol{p_2}).$$

As a result, $h(\boldsymbol{n}, \mathcal{D})$ is also smaller than or equal to the network distance $\text{DIST}(\boldsymbol{n}, \boldsymbol{p_2})$. Hence, $h()$ is admissible throughout the incremental search process. ■

We now provide a proof of correctness of Algorithm 2.

Theorem 1: SWH-kNN($\mathcal{D}$, $\boldsymbol{q}$, k), Algorithm 2, produces correct kNN results.

Proof: Since the traversal order of Algorithm 2 is given by the heuristic function $h()$, the correctness of the algorithm relies on the following two properties of the heuristic function.

(i) The first property is the admissibility of the heuristic function $h()$ as given by Lemma 1. This property ensures that $h()$ always gives an optimistic estimate, and hence the algorithm cannot overlook any node $\boldsymbol{n}$ that may form the shortest path from $\boldsymbol{q}$ to the nearest object in $(\mathcal{D} \setminus \mathcal{A})$.

(ii) The second property is the admissibility of the heuristic function after an update as given by Lemma 2. This property ensures that updates to the heuristic function cannot cause Algorithm 2 to overlook the next nearest object.

These two properties ensure that the best-first search cannot overlook the k NNs and the shortest paths from $\boldsymbol{q}$ to them. ■

Next, we show that the SWH-kNN is optimal in terms of the number of visited nodes.

Theorem 2: Given a dataset $\mathcal{D}$ and a query point $\boldsymbol{q}$, let N be the set of nodes visited by SWH-kNN in order to find the first NN in $\mathcal{D}$ with respect to $\boldsymbol{q}$. The set of nodes visited by any other admissible algorithm which uses the same heuristic function as SWH-kNN is a superset of N.

Proof: Since using SWH-kNN to search for the first kNN is equivalent to using the A* search algorithm to find the shortest path from $\boldsymbol{q}$ to the nearest object in $\mathcal{D}$, we apply the proof for the optimality of A* search given by Dechter and Pearl [3]. Specifically, they proved that if the heuristic function $h()$ is consistent, then A* search using $h()$ always visits a subset of nodes visited by another admissible algorithm which use the same heuristic $h()$.

Since we proved in Lemma 1 that the heuristic function used by SWH-kNN is consistent, we can conclude that SWH-kNN is optimistic with respect to all other admissible search algorithms using the same heuristic to search for the NN. ■

Now, as we treat the problem of finding the 2-nd NN in $\mathcal{D}$ as one of finding the first NN in $(\mathcal{D} \setminus \{\boldsymbol{p_1}\})$ (where $\boldsymbol{p_1}$ denotes the first NN in $\mathcal{D}$) and finding subsequent NNs in incremental steps, the same principle is also applicable to the optimality of finding the k NNs. The same principle can also be extended to the IDSJ query.

Note that our algorithms are only optimal with respect to admissible algorithms which use the same heuristic function. That is, one may achieve a lower graph access cost by (i) trading off admissibility for speed, i.e., allowing the search to miss the best solution and to return an approximate solution only; (ii) improving on the accuracy of the heuristic function $h(\boldsymbol{n}, \mathcal{D})$.

B. Comparison with LBC and INE

In this subsection, we compare our proposed algorithm SWH with LBC and INE. As discussed in the introduction, LBC makes use of multiple priority queues, where each priority queue handles a shortest path computation from $\boldsymbol{q}$ to a kNN candidate object $\boldsymbol{p}$ from a set of possible k NNs in a dataset $\mathcal{D}$. Specifically, in each priority queue, the search order of nodes $\boldsymbol{n}$ is determined by the distance key calculated as $(\text{DIST}(\boldsymbol{q}, \boldsymbol{n}) + ||\boldsymbol{n} - \boldsymbol{p}||)$. To maintain a uniform search order across different priority queues, each insertion and deletion operation is repeated in each priority queue. As a result, each priority queue contains the same set of nodes but are organized differently according to its associated kNN candidate $\boldsymbol{p}$.

The overall search order is determined by comparing the minimum distance keys of these priority queues and select the entry with the overall minimum. Essentially, at each expansion, we explore the node $\boldsymbol{n}_{\text{min}}$ with the minimum distance

$$\text{MIN}\{\text{MIN}\{\text{DIST}(\boldsymbol{q}, \boldsymbol{n}) + ||\boldsymbol{n} - \boldsymbol{p}|| : \boldsymbol{n} \in N_r\} : \boldsymbol{p} \in \mathcal{D}\}, \quad (2)$$

where N_r denotes the set of unexplored nodes and $\mathcal{D}$ denotes the dataset in which we want to find the next NN.

Let us now show how SWH achieves the same search order without the need for multiple priority queues. For a graph traversal, SWH maintains a single priority queue in which nodes $\boldsymbol{n}$ are organized according to the sum ($\text{DIST}(\boldsymbol{q}, \boldsymbol{n}) + \text{MIN}\{||\boldsymbol{n} - \boldsymbol{p}|| : \boldsymbol{p} \in \mathcal{D}\}$). At each expansion, we therefore explore the node $\boldsymbol{n}_{\text{min}}$ with the minimum distance

$$\text{MIN}\{\text{DIST}(\boldsymbol{q}, \boldsymbol{n}) + \text{MIN}\{||\boldsymbol{n} - \boldsymbol{p}|| : \boldsymbol{p} \in \mathcal{D}\} : \boldsymbol{n} \in N_r\}, \quad (3)$$

As we can see, the two expressions (2 and 3) are equivalent. Since both algorithms LBC and SWH share the same starting condition (i.e., the query point $\boldsymbol{q}$), we can conclude that they both share the same search order.

Since LBC and SWH share the same search order, the difference in their performances has to be determined by the cost to maintain their search order. At each time a node $\boldsymbol{n}$ is visited, LBC has to perform a deletion operation of $\boldsymbol{n}$ and insertion operations of nodes $\boldsymbol{n_a}$ adjacent to $\boldsymbol{n}$ for each of the priority queues. SWH, on the other hand, maintains only one priority queue. Therefore, the same deletion and insertion operations are conducted once each. However, for each insertion of an adjacent node $\boldsymbol{n_a}$, the computation of the heuristic value $h(\boldsymbol{n_a}, \mathcal{D})$ requires consideration of multiple objects in $\mathcal{D}$. Algorithm 1 shows that we can exploit the spatial locality of reference and compute $h(\boldsymbol{n_a}, \mathcal{D})$ by considering only a small subset of objects around the query point $\boldsymbol{q}$. Experimental results show that this cost is much cheaper than maintenance of multiple priority queues.

We now compare INE with SWH. We can consider INE as a heuristic search algorithm where the heuristic function $h()$ only returns a lower bound of 0. Since SWH always produce a tighter lower bound, we can conclude that SWH accesses a smaller number of nodes. The difference in the performances of the two algorithms depends on whether the search space reduction from using a heuristic function (Algorithm 1) outweighs its computational cost. To gain a better insight into the performance diferences between the three algorithms, we present experimental studies in Section VI.

V. Extension to Support Landmarks

So far we have considered the Euclidean heuristic function and presented an efficient method to compute a heuristic estimate $h(\boldsymbol{n}, \mathcal{D})$ for the distance from a node $\boldsymbol{n}$ to the nearest data object in $\mathcal{D}$. This enables us to perform a kNN search using a single heuristic search operation instead of using multiple instances of A* search like the existing methods.

In this section, we show that the same concept is still applicable when heuristic estimates are calculated using precomputed distances, which is commonly used to derive a heuristic function when the Euclidean heuristic function is inaccurate or does not apply. A simple way to make use of precomputed distance to derive distance estimates is to precompute the distances to/from every node in the network from/to a small subset L of nodes called landmarks. A heuristic search algorithm can make use of these precomputed distances to compute a distance lower bound and upper bound for any two nodes $\boldsymbol{x}$ and $\boldsymbol{y}$ using triangle inequality. For example, for any node $\boldsymbol{l}$ in L, the distance $\text{DIST}(\boldsymbol{x}, \boldsymbol{y})$ is guaranteed to be smaller than or equal to $(\text{DIST}(\boldsymbol{x}, \boldsymbol{l}) + \text{DIST}(\boldsymbol{l}, \boldsymbol{y}))$. Following the same principle, a lower bound of $\text{DIST}(\boldsymbol{x}, \boldsymbol{y})$ can be given as $(\text{DIST}(\boldsymbol{x}, \boldsymbol{l}) - \text{DIST}(\boldsymbol{y}, \boldsymbol{l}))$ or $(\text{DIST}(\boldsymbol{l}, \boldsymbol{y}) - \text{DIST}(\boldsymbol{l}, \boldsymbol{x}))$. In other words, it is guaranteed that $\text{DIST}(\boldsymbol{x}, \boldsymbol{y})$ has to be greater than the maximum of the two lower bounds, i.e.,

$$\text{MAX}\{\text{DIST}(\boldsymbol{x}, \boldsymbol{l}) - \text{DIST}(\boldsymbol{y}, \boldsymbol{l}), \text{DIST}(\boldsymbol{l}, \boldsymbol{y}) - \text{DIST}(\boldsymbol{l}, \boldsymbol{x})\},$$

which is denoted as $\text{LMDIST}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{l})$ for conciseness. The overall lower bound across different landmarks in L is given as $\text{MAX}\{\text{LMDIST}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{l}) : \boldsymbol{l} \in L\}$. Since landmarks distances are easy to compute/store and are applicable to non-spatial networks, we use the landmark estimator to demonstrate how our proposed method can make use of precomputed distances.

In Algorithm 1, we show that we can avoid considering all data objects in $\mathcal{D}$ every time we need to compute a heuristic value $h(\boldsymbol{n}, \mathcal{D})$ for a node $\boldsymbol{n}$ by incrementally retrieving data objects with respect to the query point $\boldsymbol{q}$ and applying triangle inequality to prune objects that are faraway from $\boldsymbol{n}$. We apply the same principle to prune data objects that will result with large landmark estimates in order to generate a small list $\mathcal{C}$ of NN candidates. Note that if we do not assume a spatial data structure like the R-Tree, then we need to compute the landmark estimate for every object in $\mathcal{D}$ with respect to $\boldsymbol{q}$, so that we can prune data objects $\boldsymbol{p}$ using the distance from $\boldsymbol{q}$ to $\boldsymbol{n}$ and the distance lower bound from $\boldsymbol{q}$ to $\boldsymbol{p}$. This distance computation for the entire dataset happens only once in the beginning and the cost is amortized as the search progresses.

We are now ready to compute an overall estimate (lower bound) from a given list $\mathcal{C}$ of NN candidates. Recall that the lower bound of the distance from $\boldsymbol{q}$ to the nearest object in $\mathcal{C}$ is the minimum lower bound across different objects in $\mathcal{C}$. When the same principle is applied to landmark distances, the lower bound of the distance from $\boldsymbol{q}$ to the nearest object in $\mathcal{C}$ is given as $\text{MIN}\{\text{MAX}\{\text{LMDIST}(\boldsymbol{q}, \boldsymbol{p}, \boldsymbol{l}) : \boldsymbol{l} \in L\} : \boldsymbol{p} \in \mathcal{C}\}$.

Since we are interested in computing the overall minimum instead of the lower bound for each object $\boldsymbol{p}$, we can avoid considering all $(|\mathcal{C}| \times |L|)$ possible object-landmark combinations. Specifically, we can apply the alpha-beta pruning principle to speed up the calculation. For example, let D_{min} denote the minimum distance computed so far as we traverse the list $\mathcal{C}$. Assume that we are considering object $\boldsymbol{p}$ and find that $\text{LMDIST}(\boldsymbol{q}, \boldsymbol{p}, \boldsymbol{l})$ is greater than D_{min}. This rules out $\boldsymbol{p}$ as the object that can provide the overall minimum and hence $\boldsymbol{p}$ can be skipped, since further consideration of $\boldsymbol{p}$ cannot yield any distance smaller than D_{min}.

Furthermore, we can accommodate the pruning process by carefully organizing the order in which data objects and landmarks are considered. Ideally, we would prefer to consider objects $\boldsymbol{p}$ in $\mathcal{C}$ that yields small values of $\text{MAX}\{\text{LMDIST}(\boldsymbol{q}, \boldsymbol{p}, \boldsymbol{l}) : \boldsymbol{l} \in L\}$ first to improve the pruning power of D_{min}. Similarly, when considering each object $\boldsymbol{p}$ we also want to process landmarks l that yields large values of $\text{LMDIST}(\boldsymbol{q}, \boldsymbol{p}, \boldsymbol{l})$ first to increase the chance of finding one that is greater than D_{min}. As future work, we plan to investigate different ordering methods to speed up computations of heuristic values.

We can see that our search method which considers data objects around the query point as a single target enables us to optimize the computation of lower bounds across different data objects instead of considering all $(|\mathcal{C}| \times |L|)$ possible object-landmark combinations. Since LBC utilizes multiple instances of A* search, applying landmark distances to the algorithm can be done by directly replacing the Euclidean heuristic function with the landmark-based heuristic. Our experimental studies show that this optimization tremendously reduces the number of object-landmark combinations in comparison to LBC, where computations of heuristic values are conducted on the object-by-object basis.

VI. Experimental Studies

This section contains results of a performance comparison of our proposed algorithms and their competitors. Experiments were conducted on an Intel i7-2720QM @ 2.20 GHz with 8GB RAM. All algorithms were implemented in Java with the javac compiler version 1.6.0_22 and OpenJDK Runtime Environment (IcedTea6 1.10.2).

TABLE III
Experimental parameters.

Parameter	Default	Min	Max
Number k of resultant objects	5	5	25
Node-to-object ratio r	1,000	100	10,000
Number m of IDSJ query points	10	10	50

Table IV shows road networks of five cities extracted from `openstreetmap.org`. Note that we use only city road networks instead of state or country road networks, because the kNN query is typically used to search for nearby objects within a city. As shown in Table III, we studied the effects of the following parameters on INE-kNN, LBC-kNN and SWH-kNN as well as on their IDSJ counterparts. The descriptions of these parameters are as follows.

- *Number k of resultant objects:* The value of k indicates the number of resultant objects requested by a user. The k value ranged from 5 to 25 objects. This range of k values is typical of the number of results reported by GPS navigation devices when searching for nearby objects.
- *Object sparseness s:* The value of s indicates the number of nodes divided by the number of data objects. For example, the default s value of 1000 indicates that there is one object of interest (e.g., a bank or a gas station) for every 1000 network nodes. We choose a high object sparsity for our default to reflect the fact that when a user searches for a particular type of nearby locations the number of objects that match the user's interest is likely to be low. For example, there are 185 USPS locations around the DC metropolitan area, 8 DMV locations in Washington, DC; while the network contains 414,712 nodes.
- *Number m of IDSJ query points:* The value of m indicates the number of query points for an IDSJ query. The use of an m value between 10 and 50 locations conforms with the warehouses-and-customers scenario discussed in Section III. This parameter is not relevant for kNN queries.

We used the following measures in the experiments. (i) *Graph traversal cost:* the number of nodes visited by an algorithm. (ii) *Execution time:* the time to complete a query. (iii) *Priority queue cost (PQ Ops):* the number of priority queue operations. (iv) *Priority queue size (PQ Size):* the maximum number of entries in the priority queue(s). Each result is reported as the average of measurements from 200 independent queries.

A. Overview of the Experimental Results

In this section, we show how the three algorithms perform in different road networks, which have different sizes. The characteristics of these networks are given in Table IV. We use the default values of the parameters given in Table III.

The results are displayed in Table V. The first and second columns correspond to the network and the algorithm, respectively. The next 4 columns contain kNN results in 4 cost measures. The remaining 4 columns contain IDSJ results. Each cell shows an absolute result and a result relative to the corresponding SWH result. As can be seen from the table, changes in the network size have no effect on the graph traversal cost of any algorithm. This is because we fix the sparseness s of objects, which means that a given algorithm has to traverse a similar number of network nodes in order to find k NNs even though the network size changes. Other cost measures are also unaffected due to the same reason.

For ease of comparison, Figure 5 shows a histogram of relative results for the Washington, DC network in the four cost measures. The figure shows that, for the kNN query, we obtain similar relative costs (2.18 to 2.51) of INE with respect to SWH across for all cost measures. This shows that the reduction in the graph traversal cost of SWH is reflected in the other three cost measures. Let us now consider LBC, which has the same graph traversal cost as SWH, but has a greater number of priority queue operations than SWH (5.57 times). This is because each visited node is involved in multiple instances of A* search. In terms of the priority queue size, LBC's multiple priority queues have a total size that is 7.45 times greater than SWH, which requires only a single priority queue to traverse the network. This means that SWH requires a smaller memory space to operate than LBC.

Finally, SWH is the best performer in terms of the execution time. Averaging the results from the five road networks, SWH is 2.5 times faster than INE and 3.5 times faster than LBC for kNN. For IDSJ, SWH is 5 times faster than INE and 4 times faster than LBC. These results conform with the fact that SWH dominates INE and LBC in terms of the traversal cost, number of priority queue operations and priority queue size for both query types.

We have shown that SWH consistently outperforms INE and LBC for the default values of the parameters shown in Table III. In the next subsections, we study the effects of the parameters described in Table III using the Washington, DC road network for kNN queries and IDSJ queries. Notice that in order to be able to visualize the ranges of differences for the parameters for the different algorithms (e.g., small for some, and large for others), we had to use log scales which, at times,

TABLE IV

CHARACTERISTICS OF THE FIVE ROAD NETWORKS.

Road Network	Number of Nodes	Number of Edges	Latitudes	Longitudes	Total length	Motorway	One-way
Austin, TX	174,851	191,354	30.03°N - 30.53°N	97.49°W - 97.94°W	10,785 km	769 km	2,181 km
Phoenix, AZ	352,803	398,362	33.22°N - 33.74°N	111.56°W - 112.38°W	26,643 km	1,316 km	2,108 km
Washington, DC	414,712	442,287	38.70°N - 39.12°N	76.75°W - 77.26°W	12,370 km	874 km	2,145 km
Chicago, IL	807,387	918,574	41.39°N - 42.35°N	87.02°W - 88.56°W	58,497 km	2,686 km	5,071 km
Los Angeles, CA	1,267,729	1,401,641	33.34°N - 34.42°N	117.08°W - 118.70°W	80,520 km	5,403 km	7,859 km

TABLE V

EXPERIMENTAL RESULTS ON DIFFERENT ROAD NETWORKS WHERE EACH CELL CONTAINS AN ABSOLUTE RESULT AND A RESULT RELATIVE TO SWH IN THE FORMAT "`absolute (relative-to-SWH)`". NOTE THAT WE OBTAIN SIMILAR RESULTS ACROSS THE FIVE NETWORKS WITH DIFFERENT SIZES.

	Method	kNN Query				IDSJ Query			
		Traversal	Time	PQ Ops	PQ Size	Traversal	Time	PQ Ops	PQ Size
Austin, TX	INE	5,173 *(2.65)*	5.88 *(2.63)*	10,428 *(2.34)*	258 *(2.24)*	5,145 *(5.06)*	6.13 *(4.57)*	11,357 *(4.83)*	594 *(5.08)*
(174,851 nodes)	LBC	1,951 *(1.00)*	8.29 *(3.70)*	27,300 *(6.13)*	966 *(8.33)*	1,017 *(1.00)*	4.60 *(3.43)*	14,325 *(6.10)*	1,081 *(9.23)*
	SWH	1,951 *(1.00)*	2.24 *(1.00)*	4,453 *(1.00)*	116 *(1.00)*	1,017 *(1.00)*	1.34 *(1.00)*	2,349 *(1.00)*	117 *(1.00)*
Phoenix, AZ	INE	4,906 *(2.57)*	5.99 *(2.56)*	11,240 *(2.45)*	342 *(2.31)*	4,998 *(6.35)*	6.19 *(5.90)*	11,447 *(6.11)*	819 *(6.88)*
(352,803 nodes)	LBC	1,905 *(1.00)*	8.12 *(3.47)*	22,203 *(4.84)*	962 *(6.50)*	786 *(1.00)*	3.89 *(3.70)*	11,489 *(6.13)*	1,102 *(9.26)*
	SWH	1,905 *(1.00)*	2.34 *(1.00)*	4,589 *(1.00)*	148 *(1.00)*	786 *(1.00)*	1.05 *(1.00)*	1,873 *(1.00)*	119 *(1.00)*
Washington, DC	INE	5,111 *(2.51)*	5.70 *(2.18)*	10,848 *(2.42)*	224 *(2.36)*	5,116 *(6.36)*	6.10 *(5.64)*	11,096 *(6.02)*	559 *(5.64)*
(414,712 nodes)	LBC	2,031 *(1.00)*	7.39 *(3.18)*	24,947 *(5.57)*	708 *(7.45)*	804 *(1.00)*	4.41 *(4.08)*	13,915 *(7.55)*	1,064 *(10.74)*
	SWH	2,031 *(1.00)*	2.32 *(1.00)*	4,480 *(1.00)*	95 *(1.00)*	804 *(1.00)*	1.08 *(1.00)*	1,843 *(1.00)*	99 *(1.00)*
Chicago, IL	INE	5,042 *(2.75)*	5.80 *(2.78)*	11,541 *(2.61)*	313 *(2.32)*	5,252 *(6.74)*	6.08 *(6.02)*	12,134 *(6.50)*	793 *(6.50)*
(807,387 nodes)	LBC	1,833 *(1.00)*	6.99 *(3.34)*	20,279 *(4.58)*	793 *(5.87)*	779 *(1.00)*	4.30 *(3.99)*	12,691 *(6.80)*	1,144 *(9.38)*
	SWH	1,833 *(1.00)*	2.09 *(1.00)*	4,426 *(1.00)*	135 *(1.00)*	779 *(1.00)*	1.01 *(1.00)*	1,867 *(1.00)*	122 *(1.00)*
Los Angeles, CA	INE	5,202 *(2.42)*	6.04 *(2.35)*	11,577 *(2.33)*	302 *(2.25)*	5,147 *(4.87)*	5.83 *(4.45)*	11,545 *(6.18)*	704 *(5.77)*
(1,267,729 nodes)	LBC	2,154 *(1.00)*	10.38 *(4.03)*	37,274 *(7.49)*	1057 *(7.89)*	1,056 *(1.00)*	5.37 *(4.10)*	17,156 *(9.19)*	1,193 *(9.78)*
	SWH	2,154 *(1.00)*	2.57 *(1.00)*	4,977 *(1.00)*	134 *(1.00)*	1,056 *(1.00)*	1.31 *(1.00)*	2,446 *(1.00)*	115 *(1.00)*

result in a reduction in the ability to discern the differences. For this reason, we often resort to also provide graphs in terms of relative performance. The overall trends, though, are clear from looking at the figures.

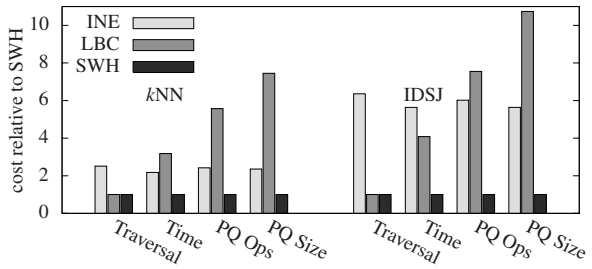


Fig. 5. Relative results for the Washington, DC network.

B. Number k of results

In this experiment, we varied the value of k between 5 and 25. Since LBC and SWH yield the same graph traversal cost, we focus on the in-memory processing cost components and report results for the total response time, number of priority queue operations, and priority queue size in Figures 6 and 7.

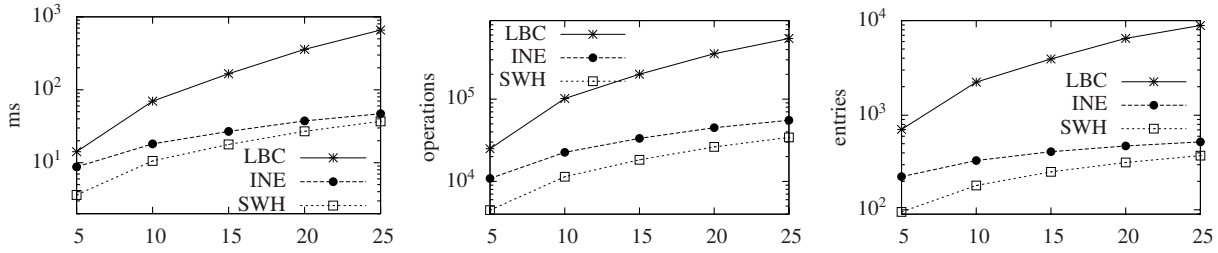
In terms of the execution time, Figure 6(a) shows that SWH consistently performs better than INE and LBC. Figure 6(b), SWH has the smallest priority queue cost in comparison to INE and LBC. The figure also shows that as k increases, the difference between the priority queue cost of INE and those of LBC and SWH becomes smaller. This is because as k increases, we have more objects around the query point making searching for k NNs less directional. Figure 6(c), shows that the results in terms of the priority queue size also conforms with the previous two cost measures. Figure 7 shows that the effect of k on the IDSJ algorithms is similar to that on the kNN algorithms.

Although SWH has a similar performance as INE for larger numbers k of NNs, in an incremental query processing environment, a user issues a query and results are displayed as they become available. The fact that SWH can produce the first 5 NNs 2.5 times faster than INE means that the user spends less time waiting for the initial results, while subsequent NNs can be incrementally reported as the user considers those currently available. This property can be useful for slower devices such as mobile phones and GPS navigators.

It is also important to note that in a location-based application, a user searching for nearby objects usually specifies which type of objects they wish to find as search criteria. These search criteria can then be used to *pre-prune* candidate objects when retrieving Euclidean NNs (Line 9 of Algorithm 1). As a result, we can filter out irrelevant objects in advance thereby discounting the need to find a large number k of results for post-pruning.

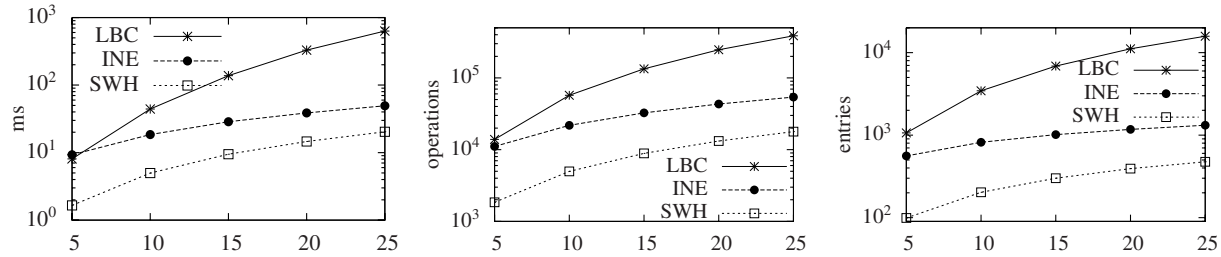
C. Object sparseness s

We used the maximum and minimum object sparseness s values of 100 and 10,000 nodes per object, respectively. For kNN queries, Figure 8(a) shows that INE is significantly faster than LBC for low values of s. In all cases, SWH is much faster than LBC. Figure 8(b) shows that as s increases, the priority queue costs of the three algorithms also increase. This is because a greater sparseness s means that we have more network nodes to consider in order to retrieve same number k



(a) Execution Time vs k (b) Priority Queue Cost vs k (c) Priority Queue Cost vs k

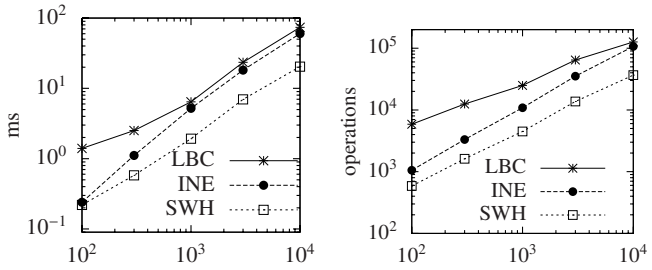
Fig. 6. The effect of k on kNN algorithms.



(a) Execution Time vs k (b) Priority Queue Cost vs k (c) Priority Queue Size vs k

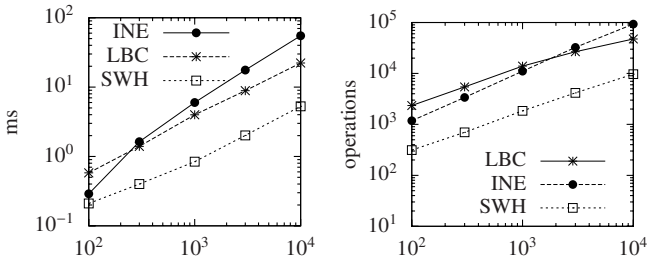
Fig. 7. The effect of k on IDSJ algorithms

of nearest objects. Figure 9 shows that the effects of s on the IDSJ algorithms are the same as that of the kNN algorithms.



(a) Execution Time vs s (b) Priority Queue Cost vs s

Fig. 8. The effect of s on kNN algorithms

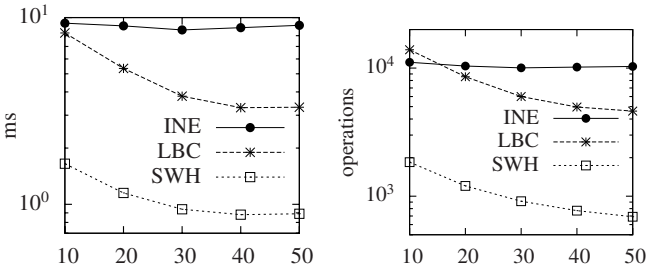


(a) Execution Time vs s (b) Priority Queue Cost vs s

Fig. 9. The effect of s on IDSJ algorithms.

D. Number m of query points (IDSJ only)

Figure 10(a) shows that the execution time of LBC is significantly greater than that of SWH. Figure 10(b) shows that as the number m of query points increases, the priority queue costs of LBC and SWH decrease. This effect can be explained by the following two reasons. First, a greater number m of query points increases the tendency that a data object would be found close to the query point. Second, LBC and SWH consider only query points that may return IDSJ results (as illustrated in Figure 4). This cost-saving benefit is accentuated as the number m of query points increases. Specifically, the Euclidean distance is used as a lower bound to prune those query points that are far away from any data object. This cost-saving benefit is absent in INE since the network expansion is done in an uninformed manner.

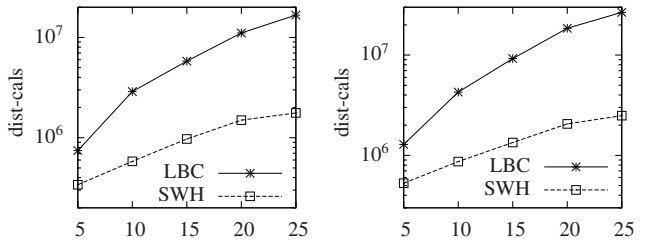


(a) Execution Time vs m (b) Priority Queue Cost vs m

Fig. 10. The effect m on IDSJ algorithms.

E. Preliminary Experiments on the Landmark Variants

In this section, we report preliminary results of our experiments on the landmark variants of SWH and LBC (described in Section V). Note that INE is omitted from this experiment since it is not a heuristic search algorithm. To compare the performance of SWH and LBC, we count the number of landmark distance calculations incurred by each algorithm during an entire search process. Figures 11(a) and 11(b) shows experimental results on the Washington, DC road network with 16 and 32 landmarks respectively. The shows that SWH achieves a much lower distance calculation cost than LBC for both cases. Please note that Figures 11(a) and 11(b) yield similar results. Although a greater number of landmarks prunes the search space and hence needs less time, it requires more distance calculations which counteracts the reduction in search per landmark.



(a) Number of Distance Calculations vs k (16 Landmarks) (b) Number of Distance Calculations vs k (32 Landmarks)

Fig. 11. The effect k on NN algorithms (with landmarks).

VII. Concluding Remarks

We proposed an incremental kNN algorithm called the single-wavefront heuristic search (SWH) kNN algorithm, which we showed to be optimal in terms of the graph traversal cost, i.e., the number of visited nodes. We compared SWH with the current state-of-the-art algorithm LBC, which is also optimal in terms of the graph traversal cost. Although both methods yield the same graph traversal cost, in a setting where the execution time is dominated by the in-memory processing cost (e.g., in an MMDBS [5], [16]), SWH has a much lower execution time than LBC. Note that LBC is outperformed by an uninformed-search algorithm INE in such a setting.

Experimental results show that SWH possesses both (i) the single-wavefront benefit found in INE, and (ii) the heuristic search benefit found in LBC. Specifically, as the value of k increases, the performance of LBC which makes use of multiple priority queues drastically degrades in comparison to both INE and SWH which use only one priority queue to traverse the network. In a multiple query point setting (IDSJ), on the other hand, LBC and SWH make use of the heuristic function, which allows them to consider only query points that have data objects nearby, while INE has to consider all query points indiscriminately. In terms of the object sparseness s, we can see that both INE and SWH significantly outperform LBC when the object sparseness is low (i.e., high object density). As s increases, the performance of INE with respect to SWH degrades drastically, while LBC becomes more competitive althugh still not as good as either of INE or SWH.

Preliminary results on the landmark variant of SWH demonstrate a potential of using our method to process kNN queries in a non-spatial graph. Future work is to improve the efficiency of our search process when using landmarks.

Acknowledgments. This work was supported in part by the NSF under Grants IIS-08-12377, CCF-08-30618, IIS-09-48548, IIS-10-18475, and IIS-12-19023, and by Google Research.

References

[1] N. Beckmann and B. Seeger. A revised R*-tree in comparison with related index structures. In *SIGMOD*, pages 799–812, 2009.

[2] S. K. Begley, Z. He, and Y.-P. P. Chen. Mcjoin: a memory-constrained join for column-store main-memory databases. In *SIGMOD Conference*, pages 121–132, 2012.

[3] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *J. ACM*, 32(3):505–536, 1985.

[4] K. Deng, X. Zhou, H. T. Shen, S. W. Sadiq, and X. Li. Instance optimal query processing in spatial networks. *VLDB J.*, 18(3):675–693, 2009.

[5] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng.*, 4(6):509–516, 1992.

[6] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *SODA*, pages 156–165, 2005.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[8] P. E. Heart, N. J. Nilsson, and B. Raphael. Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *SIGART Bull.*, (37):28–29, 1972.

[9] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, pages 237–248, 1998.

[10] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[11] C. S. Jensen, J. Kolárvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *GIS*, pages 1–8, 2003.

[12] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[13] J. M. Kleinberg, A. Slivkins, and T. Wexler. Triangulation and embedding using small sets of beacons. *J. ACM*, 56(6), 2009.

[14] S. Koenig, M. Likhachev, Y. Liu, and D. Furcy. Incremental heuristic search in AI. *AI Magazine*, 25(2):99–112, 2004.

[15] M. R. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB*, pages 840–851, 2004.

[16] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *VLDB*, pages 294–303, 1986.

[17] S. Nutanong, E. H. Jacox, and H. Samet. An incremental hausdorff distance calculation algorithm. *PVLDB*, 4(8):506–517, 2011.

[18] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. The V*-Diagram: a query-dependent approach to moving kNN queries. *PVLDB*, 1(1):1095–1106, 2008.

[19] A. Okabe, T. Satoh, T. Furuta, A. Suzuki, and K. Okano. Generalized network Voronoi diagrams: Concepts, computational methods, and applications. *International Journal of Geographical Information Science*, 22(9):965–994, 2008.

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

[21] J. Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley series in artificial intelligence. Addison-Wesley Pub. Co., 1984.

[22] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

[23] H. Samet. Distance transform for images represented by quadtrees. *IEEE Trans. Pattern Anal. Mach. Intell.*, 4(3):298–303, 1982.

[24] H. Samet, H. Alborzi, F. Brabec, C. Esperança, G. R. Hjaltason, F. Morgan, and E. Tanin. Use of the SAND spatial browser for digital government applications. *Commun. ACM*, 46(1):63–66, Jan. 2003.

[25] H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber. A geographic information system using quadtrees. *Pattern Recognition*, 17(6):647–656, November/December 1984.

[26] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, 2008.

[27] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *GIS*, pages 200–209, 2005.

[28] J. Sankaranarayanan, H. Alborzi, and H. Samet. Distance join queries on spatial networks. In *GIS*, pages 211–218, 2006.

[29] J. Sankaranarayanan and H. Samet. Distance oracles for spatial networks. In *ICDE*, pages 652–663, 2009.

[30] J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks. *IEEE Trans. Knowl. Data Eng.*, 22(8):1158–1175, 2010.

[31] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2(1):1210–1221, 2009.

[32] C. A. Shaffer, H. Samet, and R. C. Nelson. QUILT: a geographic information system based on quadtrees. *IJGIS*, 4(2):103–131, April–June 1990.

[33] M. Sharifzadeh and C. Shahabi. VoR-tree: R-trees with Voronoi diagrams for efficient processing of spatial nearest neighbor queries. *PVLDB*, 3(1):1231–1242, 2010.

[34] H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In *SIGMOD*, pages 343–354, 2000.

[35] X. Sun, S. Koenig, and W. Yeoh. Generalized adaptive A*. In *AAMAS (1)*, pages 469–476, 2008.

[36] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An efficient method for KNN join processing. In *VLDB*, pages 756–767, 2004.