Reverse Nearest Neighbors in Large Graphs

Man Lung Yiu, Dimitris Papadias, Nikos Mamoulis, and Yufei Tao

Abstract—A reverse nearest neighbor (RNN) query returns the data objects that have a query point as their nearest neighbor (NN). Although such queries have been studied quite extensively in Euclidean spaces, there is no previous work in the context of large graphs. In this paper, we provide a fundamental lemma, which can be used to prune the search space while traversing the graph in search for RNN. Based on it, we develop two RNN methods; an *eager* algorithm that attempts to prune network nodes as soon as they are visited and a *lazy* technique that prunes the search space when a data point is discovered. We study retrieval of an arbitrary number *k* of reverse nearest neighbors, investigate the benefits of materialization, cover several query types, and deal with cases where the queries and the data objects reside on nodes or edges of the graph. The proposed techniques are evaluated in various practical scenarios involving spatial maps, computer networks, and the DBLP coauthorship graph.

Index Terms— Query processing, spatial databases, graphs and networks.

1 Introduction

Given a multidimensional data set P and a point q, a (*monochromatic*) *reverse nearest neighbor* (RNN) query retrieves all the points $p \in P$ that have q as their nearest neighbor, i.e., $\mathrm{RNN}(q) = \{p \in P | \neg\exists p' \in P$ such that $d(p,p') < d(p,q)\}$, where d is a distance metric. Given two data sets P and Q and a point q, a *bichromatic* (b RNN) query retrieves all points $p \in P$ that are closer to q than to any point of Q, i.e., $b\mathrm{RNN}(q) = \{p \in P | \neg\exists q' \in Q$ such that $d(p,q') < d(p,q)\}$. The problem has received considerable attention in the last few years [9], [17], [1], [10], [14], [18] due to its relevance in several applications involving decision support, resource allocation, profile-based marketing, etc. However, all the existing work focuses exclusively on Euclidean spaces, whereas in several domains, the data are modeled as large disk-based graphs.

Let $G = (V, E, W)$ be an undirected weighted graph (we use the terms *graph* and *network* interchangeably), where V is the set of nodes and E is the set of edges. W associates each edge $n_i n_j$ with a positive real number $w(n_i n_j)$ (i.e., the weight). The interpretation of the weights is determined by the application domain, e.g., in road networks $w(n_i n_j)$ may denote the travel time or the driving distance of the road segment connecting n_i and n_j, whereas in peer-to-peer (P2P) systems, it may denote the network latency of the corresponding link, or simply be set to 1 (if the cost of a path is based on the number of hops).

The *network distance* $d(n_i, n_j)$ between two nodes $n_i, n_j \in V$ is defined as the minimum sum of weights of any path between them. We assume that the cost of traversing an edge is the same for both directions, i.e., $w(n_i n_j) = w(n_j n_i)$. Therefore, the network distance is symmetric and satisfies the inequality $d(n_i, n_j) \leq d(n_i, n_k) + d(n_k, n_j)$ (because $d(n_i, n_j)$ is the shortest distance between n_i and n_j), i.e., it is a metric. The network distance definition necessitates dedicated techniques for RNN processing, since the existing (i.e., Euclidean) methods are inapplicable.

Depending on the domain, the characteristics of the RNN problem differ. In the P2P context, each point (i.e., peer) $p \in P$ lies on a node $n \in V$, but some nodes may not contain relevant peers to a specific query. We call such networks *restricted*. For instance, in Fig. 1a, assume that a new user q interested in music enters the system. A (monochromatic) RNN query retrieves among the existing users also interested in music (p_1 to p_3), the ones for which q will become their new NN. On the other hand, nodes such as n_1 and n_2 are irrelevant to the user (e.g., they may represent peers with other types of content) and they are considered empty. Given the edge costs of Fig. 1a, $\mathrm{RNN}(q) = \{p_3\}$, e.g., q is beneficial to p_3 since it is its closest NN in terms of network cost and shares the same interests. In a collaborative environment, q would inform p_3 about its arrival, so that p_3 could address future requests directly to q, minimizing the network latency. Furthermore, the set $\mathrm{RNN}(q)$ reflects the potential workload of q; thus, by knowing this set, each peer could manage/control its available resources. Note that the NN of q (point p_1) is not an RNN (the NN of p_1 is p_2).

In *unrestricted* networks [12], [21], the query and the data points can reside anywhere on the edges of the graph. Fig. 1b shows an example of a bichromatic query in a road network, where points p_1 to p_5 stand for residential blocks and q_1, q_2 indicate restaurants. Nodes n_1 and n_2 are empty road junctions (i.e., they do not contain residential blocks or restaurants). Given several choices for the location of a new restaurant, the query $b\mathrm{RNN}(q)$ may be used to evaluate the benefit of q in terms of the customers that it may attract from rival restaurants based on proximity. Specifically, the set $b\mathrm{RNN}(q) = \{p_1, p_2, p_3\}$ represents the blocks that are closer to q than to any other competitor. Similarly,

- *M.L. Yiu and N. Mamoulis are with the Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong. E-mail: {mlyiu2, nikos}@cs.hku.hk.*
- *D. Papadias is with the Department of Computer Science, Hong Kong University of Science and Technology, Clearwater Bay, Hong Kong. E-mail: dimitris@cs.ust.hk.*
- *Y. Tao is with the Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. E-mail: taoyf@cityu.edu.hk.*

Manuscript received 15 Sept. 2004; revised 28 Apr. 2005; accepted 29 Aug. 2005; published online 17 Feb. 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0321-0904.

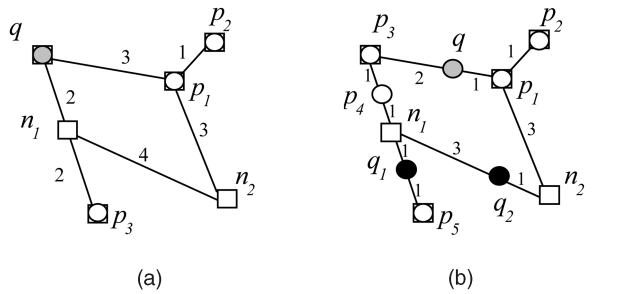

Fig. 1. Examples of RNN queries in graphs. (a) RNN in P2P system. (b) *b*RNN in road network.

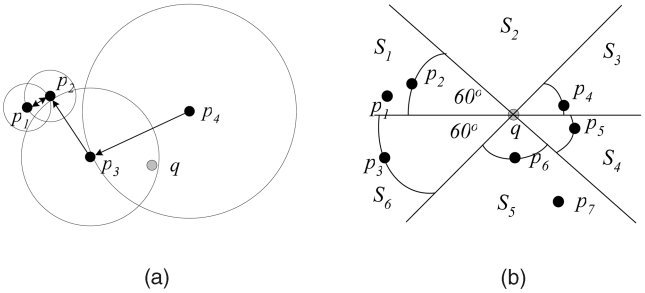

Fig. 2. Euclidean RNN algorithms. (a) Example of [9]. (b) Example of [13].

$b\mathrm{RNN}(q_1) = \{p_4, p_5\}$ and $b\mathrm{RNN}(q_2) = \emptyset$. Note that unrestricted networks can be transformed to restricted ones by adding a node for each data point that lies on a graph edge. We do not adopt this transformation since, as discussed in [12], the separation of the network component from the data points provides flexibility and facilitates updates. In particular, although the network can be considered static, the data points may change with high frequencies. Furthermore, new data sets (e.g., hotels) can be incorporated in the system as more information or services become available, without affecting the network storage scheme.

The above examples refer to single RNN retrieval. On the other hand, a (monochromatic) *reverse k nearest neighbor* (R*k*NN) query retrieves all the points $p \in P$ that have q as one of their k nearest neighbors, i.e., $\mathrm{R}k\mathrm{NN}(q) = \{p \in P | d(p,q) \leq d(p, p_k(p))$, where $p_k(p) \in P$ is the kth NN of $p\}$. For instance, assuming that in the P2P scenario each node propagates a query to four other peers (e.g., this value is used in Gnutella), the arrival of a new peer could trigger a R4NN query. In case of bichromatic queries:

$$b\mathrm{R}k\mathrm{NN}(q) = \{p \in P | d(p,q) \leq d(p, q_k(p)),$$

where $q_k(p) \in Q$ is the kth NN of p (only considering objects of Q)}, e.g., in Fig. 1b, $b\mathrm{R2NN}(q) = \{p_1, p_2, p_3, p_4\}$, $b\mathrm{R2NN}(q_1) = \{p_3, p_4, p_5\}$, and $b\mathrm{R2NN}(q_2) = \{p_1, p_2, p_5\}$.

In the sequel, we propose algorithms for processing both monochromatic and bichromatic RNN queries involving arbitrary values of k in restricted or unrestricted graphs. For our discussion, the term distance refers to the network distance defined using the edge weights. The rest of the paper is organized as follows: Section 2 overviews related work on RNN and graph algorithms. Section 3 introduces the architecture for graph storage and presents our basic methods. Section 4 focuses on optimizations and Section 5 discusses variations of RNN queries. Section 6 evaluates the proposed techniques in several practical scenarios involving spatial maps, computer networks, and the *DBLP* coauthorship graph. Section 7 concludes the paper.

2 Related Work

Section 2.1 surveys methods for RNN search in the Euclidean space. Section 2.2 presents related query processing techniques for large graphs.

2.1 RNN Search in the Euclidean Space

We start with methods for monochromatic RNN queries. The first algorithm [9] precomputes for each data point p its nearest neighbor $\mathrm{NN}(p)$. Then, it represents p as a *vicinity circle* $(p, dist(p, \mathrm{NN}(p)))$ centered at p with radius equal to the Euclidean distance between p and its NN. The MBRs of all circles are indexed by an R-tree, called the RNN-tree. Using the RNN-tree, the reverse nearest neighbors of q can be efficiently retrieved by a point location query, which returns all circles that contain q. Fig. 2a illustrates the concept using four data points, each associated with a vicinity circle. Since q falls in the circles of p_3 and p_4, the result of the query is $\mathrm{RNN}(q) = \{p_3, p_4\}$.

Because the RNN-tree is optimized for RNN, but not NN search, Korn and Muthukrishnan [9] use an additional (conventional) R-tree on the data points for computing nearest neighbors and the NN radii of the RNN-tree. In order to avoid the maintenance of two separate structures, Yang and Lin [20] combine the two indexes in the RdNN-tree. Similar to the RNN-tree, a leaf node of the RdNN-tree contains vicinity circles of data points. On the other hand, an intermediate node contains the MBR of the underlying points (not their vicinity circles), together with the maximum distance from every point in the subtree to its nearest neighbor.

Techniques (e.g., [9], [20]) that rely on preprocessing cannot deal efficiently with updates. Stanoi et al. [13] eliminate the need for precomputing all NNs by utilizing some interesting properties of RNN retrieval. Consider Fig. 2b, which divides the space around a query q into six equal regions S_1 to S_6. Let p be the NN of q in some region S_i; it can be proven that: 1) either $p \in \mathrm{RNN}(q)$ or 2) there is no RNN of q in S_i. For instance, in Fig. 2b, the NN of q in S_1 is point p_2. However, the NN of p_2 is p_1. Consequently, there is no RNN of q in S_1 and we do not need to search further in this region. The same is true for S_2 (no data points), S_3, S_4 (p_4, p_5 are NNs of each other), and S_6 (the NN of p_3 is p_1). The actual result is $\mathrm{RNN}(q) = \{p_6\}$. Based on the above property [13] adopts a two-step processing method. First, six constrained NN queries retrieve the nearest neighbors of q in regions S_1 to S_6. These points constitute the *candidate* result. Then, at a second step, a NN query is applied to find the NN p' of each candidate p. If $dist(p,q) < dist(p,p')$, p belongs to the actual result; otherwise, it is a false hit.

Assuming that the data points are indexed by an R-tree, Tao et al. [18] also adopt a two-step framework that retrieves a set of candidate R*k*NNs and then removes the

false hits. The two steps are combined into a single traversal of the tree. In particular, the algorithm first retrieves a set of potential candidates in ascending order of their distance to the query point q. Nodes that cannot contain candidates are pruned by a *half-plane* strategy and are inserted (without being visited) in a refinement set S_{rfn}. At the second step, the entries of S_{rfn} are used to eliminate false hits. Singh et al. [14] propose another multistep method for monochromatic R*k*NN retrieval in high dimensionality. The algorithm may incur false misses, i.e., it may fail to discover some RNN that is far from the query point. Benetis et al. [1] study *continuous* RNN, where P contains linearly moving objects with fixed velocities, and the goal is to retrieve all RNNs of q for a future interval.

Regarding bichromatic queries, the methods of [9], [20] can be applied directly by setting the vicinity circle of each point $p_i \in P$ using its distance from its NN $q_i \in Q$. However, they still have the problem of expensive updates. Stanoi et al. [17] propose an improved technique, which first computes the Voronoi cell of the query point (based on the objects of Q) and then retrieves the data points that fall in the cell (using a range query on the R-tree of data points). Assuming that the data arrive in the form of streams, Korn et al. [10] report aggregate results over the RNNs of a set of query points.

All the above techniques are inapplicable to graphs for several reasons. First, most techniques rely on indexes (e.g., R-trees) which can only handle Cartesian spaces. Furthermore, properties such as the one exploited by [13] do not hold in networks, i.e., the maximum number of RNNs is not constrained by the dimensionality (i.e., six in two dimensions). Similarly, the half-plane pruning technique of [18] or the Voronoi cells of [17] are specific to Euclidean distance. Finally, the above approaches target special instances of the problem (e.g., monochromatic versus bichromatic queries) and are usually limited to retrieval of the single RNN.

Since the network distance is a metric, an alternative solution could rely on indexes for *general* metric spaces (e.g., [19], [3]). However, such indexes do not capture the *connectivity* of nodes, which can be utilized to speed-up search compared to simply using the triangular inequality. In this paper, we use the network connectivity to solve RNN queries on large graphs.

2.2 Graph Algorithms

Our problem is related to shortest path computation in graphs. Given a source n_i and a destination node n_j, Dijkstra's algorithm [4] expands the network from n_i until n_j is reached. A priority queue H organizes the neighbors of the nodes found so far, so that intermediate nodes are visited in increasing order of their distances from n_i. A* search (e.g., see [15]) speeds up shortest path computation by using the Euclidean distance bounds to guide the search. Materialization techniques that precompute and store distances between nodes can be applied to retrieve the shortest path distance in constant time. However, the high storage cost renders *full* materialization (of distances for all node pairs) infeasible for most practical networks. For instance, given a graph of $V = 100$K nodes, we need to store $|V|(|V|-1)/2 \cong 5 \times 10^9$ distances. HiTi [7] and HEPV [6] avoid the extreme space requirements by *partial* materialization of a subset of the distances.

Recently, there is an increasing interest for query processing in spatial networks. Papadias et al. [12] propose a storage scheme for large graphs and algorithms for nearest neighbors, range search, and distance joins. Their methods combine connectivity and location information about data objects (indexed by R-trees) to guide search. Kolahdouzan and Shahabi [8] use the concept of network Voronoi cells and materialization to speed-up query processing. Jensen et al. [5] discuss nearest-neighbor queries for points moving in a network. Shekhar and Yoo [16] find all the nearest neighbors along a given route. Yiu and Mamoulis [21] study clustering problems in spatial networks.

Most related to our techniques is the concept of network expansion as performed by Dijkstra's algorithm and some of the methods for spatial networks (e.g., [12], [8], [21]). However, there is also an important difference. Whereas in spatial networks, the Euclidean distance constitutes a bound of the network distance, used to prune the search space (e.g., in the A* algorithm or the *Euclidean restriction* framework of [12]), in our case the Euclidean distance may be undefined (e.g., in P2P applications), or (if it is defined) it may not provide a bound for the network distance (e.g., in road networks where edge costs correspond to travel time). For generality, we do not exploit the Euclidean distance in the proposed algorithms.

3 Preliminaries and Basic Algorithms

Section 3.1 briefly describes the storage scheme for the network and a lemma for early search termination. Sections 3.2 and 3.3 present two algorithms for R*k*NN search, based on *eager* and *lazy* evaluation, respectively. For simplicity, we focus on monochromatic queries in restricted networks. The extensions to all other cases are elaborated in Section 5.

3.1 Architecture and Pruning Method

We use a file of adjacency lists to represent the graph. The adjacency list of node n keeps the neighboring nodes of n together with the weights of the corresponding edges. In order to minimize the I/O cost in the presence of a buffer, a disk page stores lists of neighboring nodes, grouped together using the method of [2]. Furthermore, we build an index on node id; for each node id in the index, there is a pointer to the corresponding list and the data point that it contains (if any). Fig. 3b shows the storage scheme for the network of Fig. 3a assuming that the adjacency lists of $\{n_1, n_4, n_7\}$, $\{n_2, n_5\}$, and $\{n_3, n_6\}$ are stored in the same disk pages.

Given a source node n and an integer k, a nearest-neighbor query retrieves the k nearest data points of n in terms of network distance. NN search can be efficiently processed by the above storage scheme and an algorithm similar to Dijkstra that utilizes a heap H to expand the network around n. In particular, the entry of n is retrieved using the index on node id. If n contains a point p and $k = 1$, p is added to the result and the search terminates. Otherwise, the nodes in the adjacency list of n are inserted to H. Subsequently, nodes are deheaped, their potential points are added to the result, and their lists are fetched. The process is repeated until k neighbors are found.

A simple method for retrieving the RNN set is to traverse the network from q, and for each point $p \in P$ encountered

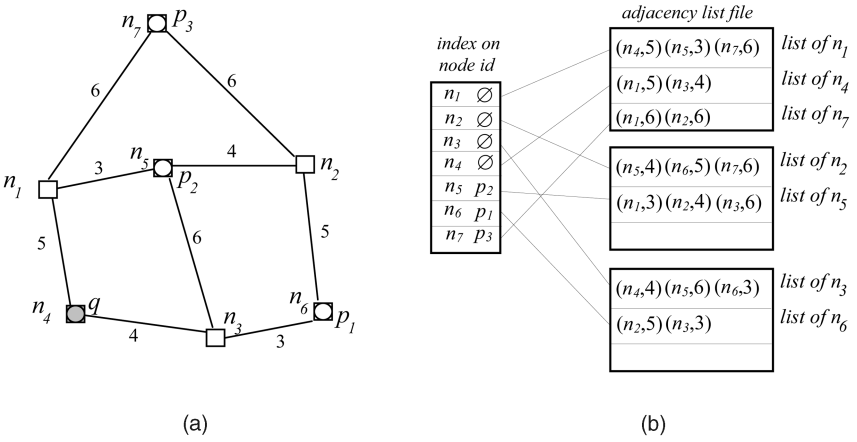

Fig. 3. Running example. (a) Network. (b) Storage scheme.

issue a nearest-neighbor query; $p \in \mathrm{RNN}(q)$, if $d(p,q)$ is no greater than the distance between p and its NN. The problem is that the set $\mathrm{RNN}(q)$ does not have a fixed size and may contain points which are far from q. Thus, the above approach would have to visit all data points. The proposed algorithms minimize the extent of network traversal by utilizing the following lemma.

Lemma 1. *Let q be a query point, n a graph node, and p a data point satisfying $d(q,n) > d(p,n)$. For any point $p' \neq p$ whose shortest path to q passes through n, $d(q,p') > d(p,p')$, i.e., $p' \notin \mathrm{RNN}(q)$.*

Proof.

$$d(q,p') = d(q,n) + d(n,p') > d(p,n) + d(n,p') \geq d(p,p'). \square$$

For instance, in Fig. 3a, $d(q,n_3) = 4 > d(p_1,n_3) = 3$. Thus, any data point (other than p_1) whose shortest path to q passes from n_3 cannot be a RNN of q because it is closer to p_1 (than q). On the other hand, $p_1 \in \mathrm{RNN}(q)$ if there is not other data points within distance $d(p_1,q)$ from p_1 (which is true in this example).

Our algorithms exploit two alternatives of NN search, which we call *range*-NN and *verification* queries. Specifically, a $\text{range-NN}(n,k,e)$ retrieves the k nearest data points with (network) distance smaller than e from n, if such k points exist. Otherwise, it returns a smaller number (possibly 0) of NNs. For instance, in Fig. 3a the range-NN query with parameters $n = n_4$, $k = 1$, and $e = 7$ has no results because the NN p_1 of n_4 has distance $d(p_1,n_4) = 7 \geq e$.

Given two points p and q, a *verification* query $\mathrm{verify}(p,k,q)$ checks whether q is among the kNNs of a data point p by applying a range-NN query around the node that contains p. $\mathrm{verify}(p,k,q)$ is in fact equivalent to $\text{range-NN}(p,k,d(p,q))$, i.e., search terminates as soon as q is encountered (the maximum range e in this case is implied by the distance $d(p,q)$).

3.2 *Eager* Algorithm

Eager traverses the network around q (in a way similar to Dijkstra's algorithm), using Lemma 1 to eliminate nodes that may not lead to RNN results. For simplicity, we first assume $k = 1$ and then deal with arbitrary values of k. The algorithm initializes a heap H by inserting the (source) node containing the query point q. When a node n is deheaped, *eager* applies Lemma 1 in order to determine whether the expansion should proceed. In particular, it first retrieves the NN of n by performing a range-NN query $(n,1,d(n,q))$. If no data point is discovered within distance $d(n,q)$ from n, the algorithm en-heaps the adjacent nodes of n. If there is a point p, such that $d(n,q) > d(n,p)$, the expansion does not proceed further because (according to Lemma 1) n cannot lead to a RNN of q. In this case, however, we need to verify if $p \in \mathrm{RNN}(q)$ because Lemma 1 is only true for points $p' \neq p$. Thus, *eager* issues a $\mathrm{verify}(p,1,q)$ query. If $q = \mathrm{NN}(p)$, p is added to the result. Furthermore, p is marked as *verified* in order not to be expanded, if it is found again in the future through another node.

As an example consider the RNN query of Fig. 3 initiated at node n_4, which is inserted into $H =< n_4, 0 >$. Then, n_4 is deheaped and its adjacent nodes are added to $H =< n_3, 4 >, < n_1, 5 >$. The subsequent removal (from H) of n_3 triggers a range-NN$(n_3,1,4)$ around n_3 for finding potential data points closer than the query. In this case, $d(n_3,p_1) = 3 < d(n_3,q)$, meaning (by Lemma 1) that we do not search farther, i.e., the adjacent nodes of n_3 are not inserted to H. Nevertheless, we have to check if $p_1 \in \mathrm{RNN}(q)$ by verify$(p_1,1,q)$. Since $q = \mathrm{NN}(p_1)$, p_1 is an actual result. Next, n_1 is deheaped, its NN p_2 is discovered (by range-NN$(n_1,1,5)$) and search terminates because $d(n_1,q) = 5 > d(n_1,p_2) = 3$. The final step simply verifies p_2 as an actual result because $q = \mathrm{NN}(p_2)$.

Fig. 4 illustrates a pseudocode of *eager* for an arbitrary order-k of RNNs. The main difference with respect to single RNN retrieval is that we can stop the expansion of the deheaped node n, only if k data points are found within distance $d(q,n)$ from n. *Eager* verifies all data points encountered by the range-NN query. Specifically, a point $p \in \mathrm{R}k\mathrm{NN}(q)$, iff q is discovered by $\mathrm{verify}(p,k,q)$; otherwise, $p \notin \mathrm{R}k\mathrm{NN}(q)$. Note that for each point p, we only know $d(p,n) + d(n,q)$, which is an upper bound for $d(p,q)$ (i.e., if the shortest path from p to q does not pass through n, then $d(p,q) < d(p,n) + d(n,q)$). This does not affect the correctness of the verification phase since the search will anyway terminate as soon as q is discovered.

```
Algorithm eager(q,k)
1. insert < n(q),0> into H // n(q) is the node containing q
2. while (not-empty(H))
3.   <n, d(n,q)>:=de-heap(H)
4.   if n has not been visited before
5.       mark n as visited
6.       kNN(n):= range-NN(n,k,d(n,q))
7.       for each point p in kNN(n)
8.           if p has not been verified before
9.               mark p as verified
10.              kNN(p):= verify(p, k, q)
11.              if q discovered by verification, add p to RkNN(q)
12.      if |kNN(n)|<k //fewer than k points found around n
13.          for each adjacent non-visited node n_i of n
14.              insert <n_i, d(n,q)+ w(n_i,n)> to H
15. return RkNN(q)
```

Fig. 4. *Eager* algorithm.

3.3 *Lazy* Algorithm

Although *eager* minimizes the number of nodes inserted into the heap, it may perform numerous local network expansions for 1) retrieving the nearest point p of a deheaped node and 2) for verifying whether $p \in \mathrm{RNN}(q)$. The *lazy* algorithm delays pruning until a point is visited.

In particular, assuming single RNN retrieval, when the deheaped node n does not contain a point, *lazy* simply inserts its adjacent nodes into H. If n contains a point p, the expansion stops since every subsequent node is closer to p than to q. *Lazy* issues a verification query to check whether p is a result. Verification queries are exploited to prune the search space. Specifically, let n' be a node visited by the verification phase of a data point p. If n' has not been visited by the expansion around the query, we know that $d(n', p) < d(n', q)$ since nodes are visited in ascending order of their distances and the verification query has maximum range $d(p, q)$ (i.e., $d(n', p) < d(p, q) \leq d(n', q)$). Therefore, (by Lemma 1) n' cannot lead to any RNNs of q. On the other hand, if n' has already been visited by the expansion around q, we compare the distances $d(n', p)$ and $d(n', q)$. If $d(n', p) < d(n', q)$, nodes inserted into H by the expansion of n' are eliminated.

An in-memory hash table is maintained in order to facilitate pruning for visited nodes. When a node n is deheaped its id is stored in the table, together with its distance from q and pointers to all the heap entries that were inserted during its processing. Thus, if n is invalidated in the future by a verification query, its adjacent nodes are removed from the heap by following the pointers. Furthermore, a table entry is deleted if it does not have any pointers to heap entries. Thus, the total number of pointers in the hash table equals the heap cardinality.

Fig. 5 illustrates an example where the first data point (p_1) is discovered when node n_4 is deheaped (nodes are numbered according to their distance from q). *Lazy* checks whether $p_1 \in \mathrm{RNN}(q)$ by a query $\mathrm{verify}(p_1, 1, q)$ that visits nodes n_5, n_2, and n_3 before determining that $q = \mathrm{NN}(p_1)$. The verification of p_1 will encounter n_5; thus, when n_5 is deheaped later it will be immediately discarded. Fig. 6 illustrates the contents of the heap and the hash table for the example of Fig. 5. n_7 (inserted during the processing of n_2) cannot lead to a RNN since $d(n_2, p_1) < d(n_2, q)$. Similarly, n_6 (inserted during the processing of n_3) does not need to be expanded because $d(n_3, p_1) < d(n_3, q)$ and *lazy* terminates with $\mathrm{RNN}(q) = \{p_1\}$.

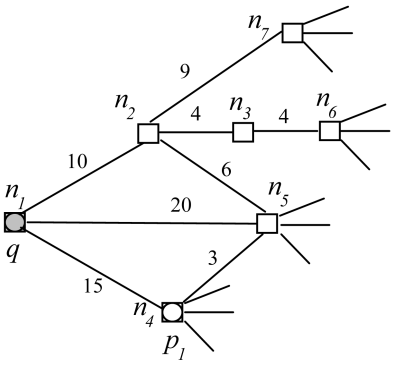

Fig. 5. Example of single RNN by *lazy*.

The retrieval of R*k*NNs is more complex because *lazy* may have to expand nodes that contain data points. In this case, we keep a *count* of the number of times that a node n is encountered during the verification phase. As shown in Fig. 7, if $count(n) \geq k$, n is not visited by the RNN query (line 4) because it is closer to at least k data points than q. Furthermore, if during a verification query the count of a visited node n_i becomes k (line 11) and $d(n_i, p) < d(n_i, q)$, the entries that were inserted into H during the processing of n_i are removed (using the hash table). This process is not repeated if $count(n_i)$ subsequently exceeds k since it would be redundant. Unvisited nodes, whose counter reaches k, are not removed from the heap since they will be captured when visited.

4 Optimizations

In this section, we discuss some optimizations to speed up performance. Section 4.1 proposes a materialization-based

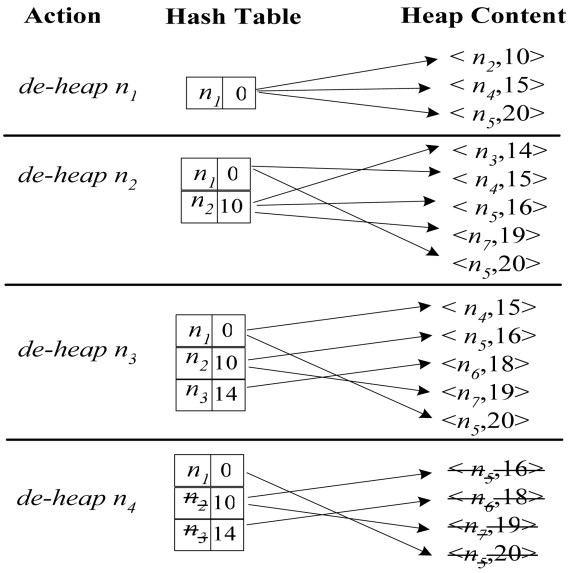

Fig. 6. Hash table and heap contents.

```
Algorithm lazy(q,k)
1. insert <n(q), 0> into H // n(q) is the node containing q
2. while (not-empty(H))
3.   <n, d(n,q)>:=de-heap(H)
4.   if n has not been visited before and count(n)<k
5.       mark n as visited
6.       if n contains a data point p
7.          kNN(p):= verify(p, k, q)
8.          if q discovered by verification, add p to RkNN(q)
9.          for each node n_i encountered by the verification
10.             count(n_i)++
11.             if count(n_i)=k, n_i is visited and d(n_i,p)<d(n_i,q)
12.                 remove from H all entries inserted by n_i
13.      if count(n)<k //n may be expanded even if it contains p
14.         for each adjacent non-visited node n_i of n
15.            if count(n_i)<k, insert <n_i, d(n,q)+ w(n_in)> to H
16. return RkNN(q)
```

Fig. 7. *Lazy* algorithm.

method for *eager*, and Section 4.2 an extended pruning technique for *lazy*.

4.1 Materialization for *eager*

Eager can take advantage of materialized information to avoid local expansions incurred by range-NN and verification queries. As discussed in Section 2.2, full materialization of the graph requires the storage of $|V|\,(|V|-1)/2$ distances. In order to avoid the quadratic space overhead, we follow an alternative approach that materializes[1] the *K*NNs of each node n, where K is the maximum number of RNNs to be requested by any query. Practical values of K are fairly small (e.g., 20 for 2D space in [18] and 50 for high-dimensional spaces in [14]).

A naïve solution for computing the K closest data points of all nodes, would simply apply a *K*NN query for each node. Instead, we use the algorithm of Fig. 8 that expands the network only once. *All*-NN initializes a single heap H that stores the nodes containing the data points together with their distance (initially 0) from the corresponding point. When a node n is deheaped, if all its *K*NNs have been found or n has been visited by the same point, we ignore it. Otherwise, we update the K NN set of n and en-heap the entries for its unvisited neighbor nodes (by the same point). Each completed *K*NN list is flushed to the disk. The process terminates when H becomes empty. The worst-case complexity is $\mathrm{O}(K \cdot |E| \cdot \log(K \cdot |E|))$, where $|E|$ is the number of edges in the graph, because an edge may be inserted in the heap at most K times and each heap operation has logarithmic cost. The space overhead for materialization is $\mathrm{O}(K \cdot |V|)$.

Fig. 9 illustrates an example for materializing the single NN ($K = 1$) of each network node in Fig. 3, where $P = \{p_1, p_2, p_3\}$. All-NN(1) initializes the heap to $H =< n_6, p_1, 0 >, < n_5, p_2, 0 >, < n_7, p_3, 0 >$. Next, it deheaps $< n_6, p_1, 0 >$, writes $\mathrm{NN}(n_6) = p_1$, and en-heaps the entries $< n_3, p_1, d(n_6, p_1) + w(n_6 n_3) >$ and

$$< n_2, p_1, d(n_6, p_1) + w(n_6 n_2) > .$$

1. Each entry in the materialized list $K\mathrm{NN}(n)$ for node n is a pair $< p_i, d(p_i, n) >$, where p_i, $1 \le i \le K$, is the ith-NN of n.

```
Algorithm all-NN(K)
1. for each node n containing a point p ∈ P
2.       insert <n,p,0> into H
3. while (not-empty(H))
4.   <n, p, d(p,n)>:=de-heap(H)
5.   if |KNN(n)|<K and p ∉ KNN(n)
6.       add <p,d(p,n)> to KNN(n)
7.       for each adjacent node n_i of n
8.          if |KNN(n_i)|<K and p ∉ KNN(n_i)
9.                insert <n_i, p, d(n,p)+ w(n_in)> to H
10.      if |KNN(n)|=K write KNN(n) to the disk
11. return
```

Fig. 8. All-NN algorithm.

Similarly, after the deheaping of $< n_5, p_2, 0 >$ and $< n_7, p_3, 0 >$, the nearest-neighbor lists are updated to $\mathrm{NN}(n_5) = p_2$, $\mathrm{NN}(n_7) = p_3$, and

$$\begin{aligned} H = &< n_3, p_1, d(n_6, p_1) + w(n_6 n_3) >, \\ &< n_1, p_2, d(n_5, p_2) + w(n_5 n_1) >, \\ &< n_2, p_2, d(n_5, p_2) + w(n_5 n_2) >, \\ &< n_2, p_1, d(n_6, p_1) + w(n_6 n_2) >, \\ &< n_3, p_2, d(n_5, p_2) + w(n_5 n_3) > . \end{aligned}$$

The subsequent processing: 1) of $< n_3,\ p_1,\ d(n_6, p_1) + w(n_6 n_3) >$ will lead to $\mathrm{NN}(n_3) = p_1$, 2) of $< n_1, p_2, d(n_5, p_2) + w(n_5 n_1) >$ to $\mathrm{NN}(n_1) = p_2$ and 3) of $< n_2, p_2, d(n_5, p_2) + w(n_5 n_2) >$ to $\mathrm{NN}(n_2) = p_2$. The algorithm terminates after processing n_4 since all the NN lists are completed.

Eager-M (M stands for materialization) utilizes the stored information to avoid network expansions as follows: When a node n is deheaped, instead of applying range-NN$(n, k, d(n, q))$, it retrieves its *k*NNs directly. Furthermore, the verification of a data point $p \in k\mathrm{NN}(n)$ must compare $d(p, q)$ with $d(p, p_k(n'))$, where $p_k(n')$ is the kth NN of the node n' that contains p. However, when p is discovered, we only have $d(q, n) + d(n, p)$, which is an upper bound for $d(p, q)$. Thus, if $d(q, n) + d(n, p) \le d(p, p_k(n'))$, then $p \in \mathrm{R}k\mathrm{NN}(q)$. Otherwise, *eager*-M performs a query $\mathrm{verify}(p, k, q)$.

Now, it remains to clarify the maintenance of materialized information in the presence of object updates. An insertion of a new data point p is handled by a variation of all-NN that initializes H to $< n, p, 0 >$, where n is the node containing p. Suppose that we insert the point p_4 at node n_4 in Fig. 9. First, we en-heap the entry $< n_4, p_4, 0 >$. The materialized NN of n_4 is updated to $\mathrm{NN}(n_4) = p_4$ and the entries $< n_3, p_4, 4 >, < n_1, p_4, 5 >$ are inserted into H. Then, $< n_3, p_4, 4 >$ is deheaped; since $d(n_3, p_4) \ge 3$ (i.e., the distance between n_3 and its existing NN p_1), the materialized NN of n_3 is not updated and the neighbor nodes are not en-heaped. A similar situation happens when $< n_1, p_4, 5 >$ is deheaped (because $d(n_1, p_4) \ge d(n_1, p_2)$) and the algorithm terminates.

On the other hand, deletion is more complex because we have to modify all influenced nodes, i.e., the nodes for which the deleted point is a *K*NN. This is essentially a R*K*NN query; the difference with respect to the algorithms of Section 3, is that 1) now we are looking for the RNN nodes, as opposed to the RNN data points (i.e., we also consider nodes that do not contain data points), and 2) we

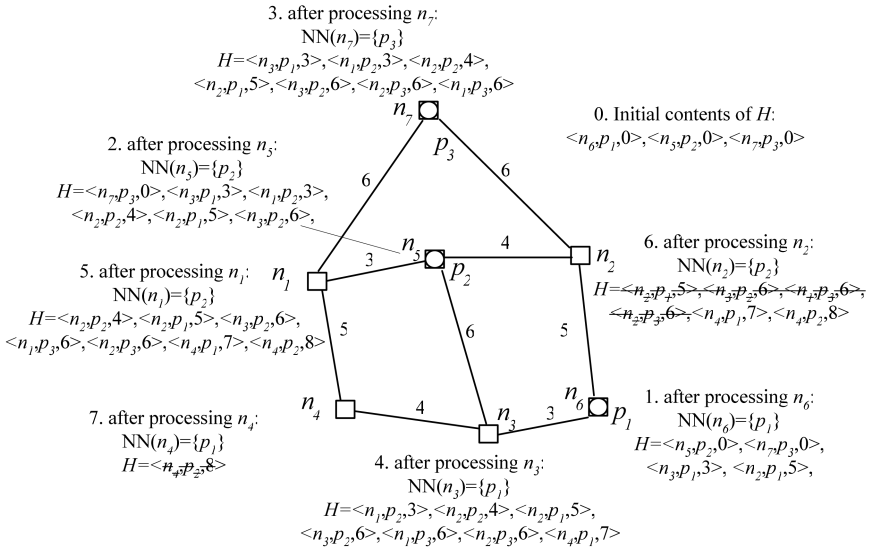

Fig. 9. All-NN example.

need to update the affected nodes by retrieving their new NNs. The deletion algorithm (shown in Fig. 10) applies two steps. The first step expands the network around p, removing p from the materialized lists of all nodes n for which $p \in K\text{NN}(n)$. The expansion stops at some *border* nodes, whose *K*NNs do not change. Let n be a border node; then, n has at least one affected neighbor n', such that $|K\text{NN}(n')| < K$ (since p was removed from $K\text{NN}(n')$). For each n', the algorithm will insert K entries of the form $< n', p_i, d(n, p_i) + w(n'n) >$ to a second heap H', where p_i, $1 \leq i \leq K$, is the ith NN of n. Then, the second step will complete the *K*NN list of all affected nodes by using the contents of H' to perform another expansion. The intuition is that the new NN (to replace p) of n' is one of the NNs of the adjacent border nodes.

Fig. 11 illustrates the deletion of p_1 in the example of Fig. 3, assuming that $K = 1$. Point p_1 is the NN of n_6, n_3, and n_4. The border nodes are n_2, n_5, and n_1 since they are neighbors of the affected nodes (and their NNs do not change). During the first step of deletion, the third node deheaped from H is $n_2(\text{NN}(n_2) = p_2)$, which causes the insertion of entry $< n_6, p_2, d(n_2, p_2) + w(n_2n_6) >$ to H'. Similarly, the processing of border nodes n_5 and n_1 will add entries $< n_3, p_2, d(n_5, p_2) + w(n_5n_3) >$ and $< n_4, p_2, d(n_1, p_2) + w(n_1n_4) >$, respectively, and the first step terminates. Note that the expansion does not proceed after a border node is visited; e.g., in this case, the processing of n_5 will not cause the insertion of $< n_1, 3 >$ in H (observe that there is only one entry $< n_1, 12 >$ in H). The second step computes the NN sets of the affected nodes in the order that they are deheaped from H', i.e., the processing of n_3 will lead to $\text{NN}(n_3) = p_2$ and the insertion of

$$< n_1, p_2, d(n_3, p_2) + w(n_3n_1) >,$$
$$< n_4, p_2, d(n_3, p_2) + w(n_3n_4) >$$

to H'. The final result will update: $\text{NN}(n_4) = p_2$ and $\text{NN}(n_6) = p_2$.

Each insertion and deletion is expected to affect a small percentage of the nodes. Thus, the memory requirements of the update algorithms are relatively small. On the other hand, for some large problems, the heap size for all-NN may exceed the available main memory. In this case, we can divide P into partitions and apply an incremental batch insertion (a variation of all-NN) for each partition. Finally, note that materialization is not beneficial for *lazy* because 1) it does not apply range-NN queries (therefore, it cannot benefit as much as *eager*) and 2) the expansion during the verification queries constitutes an essential pruning component of the algorithm. In the next section, we discuss an alternative optimization technique for *lazy*.

```
Algorithm deletion-M(p,K)
1. insert <n(p), 0> into H // n(p) is the node containing p
2. while (not-empty(H)) //first step
3.   <n, d(n,q)>:=de-heap(H)
4.   if n has not been visited before
5.       mark n as visited
6.       if p ∈ KNN(n)
7.           remove p from KNN(n)
8.           for each adjacent non-visited node n_i of n
9.                insert <n_i, d(n,p)+ w(n_in)> to H
10.      else // p ∉ KNN(n); i.e., n is a border node
11.          for each p_i∈ KNN(n)
12.               for each adjacent node n' of n with |KNN(n')|<K
13.                    insert <n',p_i, d(n,p_i)+w(n' n)> to H'
14. end while //empty(H)
15. while (not-empty(H')) // second step
16.       <n', p', d'>:=de-heap(H' )
17.   if | KNN(n') |<K
18.      add <p', d'> to KNN(n')
19.      for each adjacent node n_i of n' with |KNN(n_i)|<K
20.           insert <n_i,p', d'+w(n'n_i)> to H'
21. end while //empty(H')
```

Fig. 10. Object deletion for materialized *K*NN.

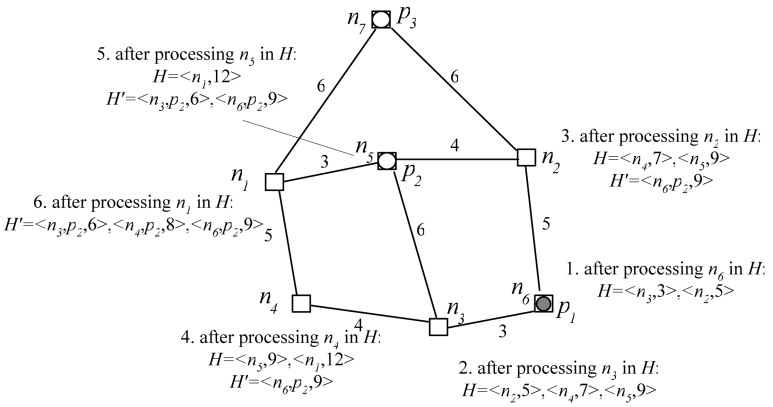

Fig. 11. Deletion of p_1—first step.

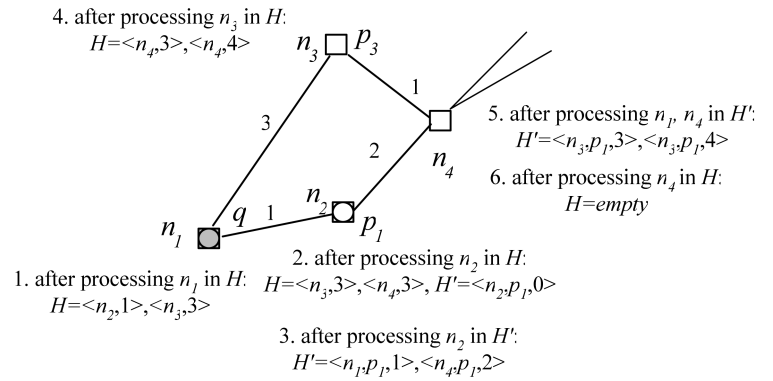

Fig. 12. *Lazy-EP* example.

4.2 Extended Pruning for *lazy*

Although *lazy* takes advantage of verification to reduce the search space, it may still traverse a large part of the network. Consider, for instance, the R1NN query of Fig. 12, where n_2 (containing point p_1) is processed after the source node n_1. The verification of p_1 will not prune any node. Thus, after the deheaping of n_3, *lazy* will continue with n_4 possibly reaching numerous nodes before termination. However, the expansion is useless because n_4 is closer to p_1 than q and, therefore, according to Lemma 1, it cannot lead to any results.

The *lazy-EP* algorithm (*EP* stands for extended pruning) avoids such cases by expanding the network in parallel using two heaps H and H'. The heap H' applies the pruning effect of the discovered points. In the above example, when the entry $< n_2, 1 >$ is processed in the conventional heap H, a new entry $< n_2, p_1, 0 >$ is inserted in H'. Then, expansion proceeds with the heap H' whenever its top distance is less than the last deheaped distance of H. Continuing the example, $< n_2, p_1, 0 >$ is deheaped and $< n_1, p_1, 1 >$, $< n_4, p_1, 2 >$ are inserted into H'. Since the last deheaped distance from H is 1, we now switch back to H. The entry $< n_3, 3 >$ is deheaped and $< n_4, 4 >$ is inserted into H. The expansion proceeds with H' and the entries $< n_1, p_1, 1 >$ and $< n_4, p_1, 2 >$ are deheaped. The processing of $< n_4, p_1, 2 >$ marks node n_4 using the distance $d(p_1, n_4)$. After that, $< n_4, 4 >$ is deheaped from H; because n_4 is closer to p_1 than q, the adjacent unvisited nodes of n_4 are not inserted to H and the algorithm terminates.

Fig. 13 illustrates *lazy-EP* for arbitrary values of k. Whereas *lazy* uses verification to update counters and prune the search space, *lazy-EP* eliminates nodes during the expansion of H'. In addition, *lazy-EP* maintains the kNN of each node found so far in order to avoid visiting the same point again during the expansion of H'. The expansion in H is pruned (by Lemma 1) if $d(n, q) > d(n, p_k(n))$, where $p_k(n)$ is the kth-neighbor of n (if $|k\text{NN}(n)| < k$, we consider that $d(n, p_k(n)) = \infty$).

5 Variants of RNN Queries

Section 5.1 studies bichromatic and continuous RNN retrieval. Section 5.2 extends our techniques to unrestricted networks with data points lying on the edges.

5.1 Bichromatic and Continuous RNN Queries

Given two data sets P and Q and a point q, the output of a *bichromatic* RkNN query is

$$b\text{R}k\text{NN}(q) = \{p \in P | d(p, q) \leq d(p, q_k(p)),$$

where $q_k(p) \in Q$ is the kth NN of p only considering objects of $Q\}$. The processing of a bichromatic query can be reduced to the monochromatic case, where the data set is Q (instead of P) and the result contains all nodes n (instead of data points) for which $q \in k\text{NN}(n)$. A final step simply retrieves the points $p \in P$ contained in these nodes. Specifically, Lemma 1 now requires that for a node n to be pruned it should be closer to k points of Q than q. In both *eager* and lazy expansion stops at nodes pruned by Lemma 1: 1) for *eager*, it is not important if the node n under consideration contains a point and 2) for lazy, n is examined if it contains a point $q_i \in Q$. Furthermore, the

```
Algorithm lazy-EP(q,k)
1. insert <n(q), 0> into H // n(q) is the node containing q
2. while (not-empty(H))
3.   <n, d(n,q)>:=de-heap(H)
4.   if n has not been visited before and d(n,q) ≤ d(n , p_k(n))
       // p_k(n) is the kth-NN of n found so far
5.     mark n as visited
6.     if n contains a data point p
7.       insert <n,p,0> to H'
8.       while ( H'.top.dist < d(n,q) ) // extended pruning
9.         < n',p',d'> :=de-heap (H')
10.          if  d' < d(n' ,p_k(n') ) and (p',d') ∉ kNN(n')
11.            update kNN(n') by (p',d')
12.            if n' visited and d(n' ,p_k(n'))<d(n',q)
13.              remove from H all entries inserted by n'
14.            for each adjacent node n_y of n'
15.              if d' + w(n'n_y) < d(n_y ,p_k(n_y) )
16.                insert <n_y, p', d' + w(n'n_y)> to H'
17.     end-while
18.     kNN(p):= verify(p, k, q)  // do not update count here
19.     if q discovered by verification, add p to RkNN(q)
20.   for each adjacent non-visited node n_i of n
21.     if d(n,q)+ w(n_in) ≤ d(n_i ,p_k(n_i))
22.       insert <n_i, d(n,q)+ w(n_in)> to H
23. return RkNN(q)
```

Fig. 13. *lazy-EP* algorithm.

optimizations of Section 4 can also be applied. For *eager-M*, we simply materialize the set $K\mathrm{NN}(n) \subseteq Q$ for each node n, whereas *lazy-EP* remains the same but only considers Q during the parallel expansions.

The concept of continuous RNN queries, as defined for objects moving in the Euclidean space [1], is not applicable to graphs, because objects do not follow linear movement. Instead, in our case, a *continuous* query retrieves the RkNNs for each node $n_i \in V$ on a predefined route $r =< n_1, n_2, \ldots, n_r >$, where $n_i n_{i+1} \in E$, i.e., $c\mathrm{R}k\mathrm{NN}(r) = \cup_{n_i \in r} \mathrm{R}k\mathrm{NN}(n_i)$. The distance between r and a network node n is defined as:

$$d(r, n) = min\{d(n_i, n) | n_i \in r\}.$$

Eager and *lazy* (as well as their optimizations) can be applied directly using the new distance definition $d(r, n)$. For example, $eager(r, k)$ initially inserts $< n_1, 0 >, \ldots, < n_r, 0 >$ into H. When a node n is deheaped for the first time, its distance corresponds to $d(r, n)$ and n is marked as visited. The range-NN$(n, k, d(r, n))$ retrieves the NNs of n within the range $d(r, n)$. If the query returns k data points, n is pruned. Moreover, the maximum verification range of a point p is $d(r, p)$, i.e., $p \in c\mathrm{R}k\mathrm{NN}(r)$, if the first node of the route is met before k other points are found. The same modification applies to *lazy*.

5.2 RNN Queries in Unrestricted Networks

In an unrestricted network, the position of a point p lying on the edge $n_i n_j$ can be expressed by the triplet $< n_i, n_j, pos >$, where $pos \in [0, w(n_i n_j)]$ is the distance of p from node n_i and $w(n_i n_j)$ is the weight (cost) of the edge. In order to avoid ambiguity, we assume a lexicographic ordering of nodes so that p is assigned to the edge $n_i n_j$, $i < j$. The *direct distance* between a point p at $< n_i, n_j, pos >$ and node n_i is $d_L(p, n_i) = pos$. The direct distance between p and n_j is $d_L(p, n_j) = w(n_i n_j)$-pos. Let p and p' be two points at $< n_i, n_j, pos >$ and $< n'_i, n'_j, pos' >$, respectively. If $n_i = n'_i$ and $n_j = n'_j$ (i.e., p and p' lie on the same edge), the direct distance $d_L(p, p')$ between p and p' is defined as $|pos\text{-}pos'|$; otherwise, it is ∞. The *network distance* $d(p, p')$ is $min_{x \in \{i,j\}, y \in \{i',j'\}}(d_L(p, n_x) + d(n_x, n_y) + d_L(n_y, p'))$, if p and p' lie on different edges; otherwise, $d(p, p')$ is the minimum of the previous quantity and $d_L(p, p')$. The distance $d(p, p')$ is symmetric and satisfies the triangular inequality.

For the following example, we use the network of Fig. 14a. Since the data points do not necessarily lie on nodes, the storage scheme of Fig. 3b does not apply. Instead, we store the data points in a separate file, pointed by the edges on which they reside (see Fig. 14b), so that when an edge (e.g., $n_2 n_6$) is processed we can efficiently retrieve all the points (e.g., p_1) on the edge. Although only $d_L(p_1, n_2) = 4$ is stored, $d_L(p_1, n_6)$ (= 1) can be computed as $w(n_2 n_6)$-$d_L(p_1, n_2)$.

Lemma 1 does not presume that the query or data points lie on nodes; therefore, it is still applicable. The general framework of *eager* also remains the same as presented in Section 3. The difference is in the basic algorithms for retrieval and verification of the NNs of a node. We focus on range-NN, since verification is similar. In case that the

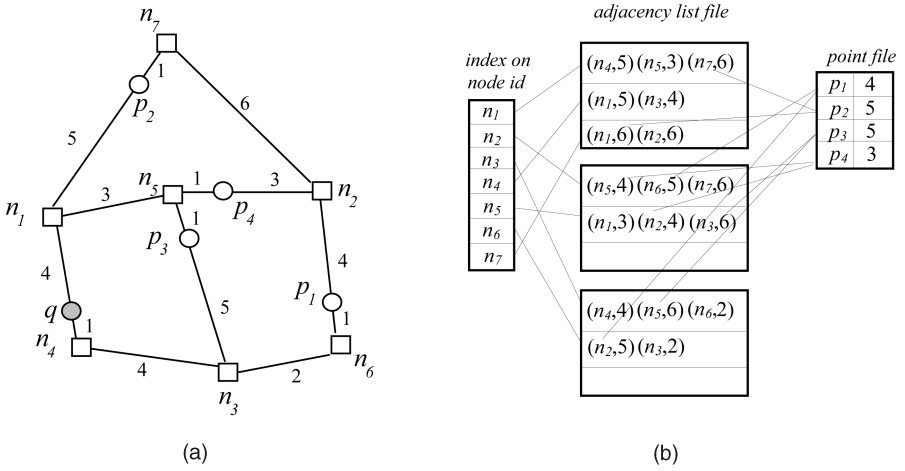

Fig. 14. Example with points lying on edges. (a) Network. (b) Storage scheme.

TABLE 1
Cost of Ad Hoc Queries ($DBLP, k = 1$)

Condition	*eager*-I/O	*lazy*-I/O	*eager*-CPU	*lazy*-CPU
0 *SIGMOD*	64	74	0.053	0.060
1 *SIGMOD*	73	77	0.077	0.057
2 *SIGMOD*	86	88	0.201	0.091

TABLE 2
Cost versus Density $D(DBLP, k = 1)$

Density	*eager*-I/O	*lazy*-I/O	*eager*-CPU	*lazy*-CPU
0.01	87	88	2.176	0.124
0.02	85	87	1.316	0.110
0.05	82	85	0.338	0.099
0.1	78	82	0.104	0.085

query or the data points lie on the edges, the range-NN algorithm discussed in Section 3.1 is inapplicable because 1) data points are not necessarily visited according to their distance from the source node, and 2) the same data point may be found twice (with two different distances). Consider, for instance, the query q in Fig. 14a and point p_3 on the edge n_3n_5. When n_3 is processed, we can compute an upper bound for the distance between q and p_3 as: $d(q, n_3) + d_L(n_3, p_3) = 10$. The subsequent processing of n_5 will provide another upper bound $d(q, n_5) + d_L(n_5, p_3) = 8$. The actual distance $d(q, p_3) = 8$ is the minimum of the two bounds.

The *unrestricted-range*-NN algorithm takes the above observations into account for finding the kNNs of node n within a range e (if k such points exist). When a node n_i is deheaped, each point p on its adjacent edges is inserted back to H, provided that $d(q, n_i) + d_L(n_i, p) < e$. Thus, points are deheaped (and inserted to $k\text{NN}(n)$) in ascending order of their distance from n. Multiple insertions of the same point visited via different paths are avoided. The search terminates when the number of discovered points equals k, the range e is exceeded, or the heap becomes empty.

Verification queries follow the same approach. By substituting the original with the unrestricted functions, we can obtain a version of *eager* for unrestricted networks. Similarly, *Eager-M* requires minor modifications of the all-NN and the update algorithms. Note that although the kNNs of a point p (lying on the edge n_in_j) are not stored, they can be computed efficiently using $k\text{NN}(n_i)$, $k\text{NN}(n_j)$, $d_L(n_i, p)$, and $d_L(n_j, p)$.

On the other hand, recall that *lazy* initiates the pruning process during the deheaping of nodes that contain data points. For unrestricted networks, the pruning occurs during the processing of edges. Assuming, for example, R1NN retrieval, when a node n is deheaped, we check the adjacent edges. If the edge nn' (where is n' is an unvisited node) contains a data point p, then n' is not en-heaped because it is closer p to than q. The algorithms can also be applied for bichromatic and continuous queries in unrestricted networks with straightforward modifications. In the next sections, we evaluate the proposed techniques under various settings.

6 Experimental Evaluation

Let $|V|$ be the node cardinality and $|P|$ be the data cardinality; the ratio $|P|/|V|$ corresponds to the data *density* D. In general, high density leads to low processing cost since it limits the extent of expansions. For instance, assuming a restricted network, in the extreme case that $D = 1$ (every node contains a point), the output of a R1NN query contains only the point that resides on the query node. In order to provide meaningful results, we set the maximum value of density to 0.1 in all cases. The diagrams display the average cost of workloads containing 50 queries. Each query is randomly chosen from the set of data points, so that the queries follow the data distribution. For all experiments, we use a Pentium 4 CPU (2.3GHz) with page size set to 4Kb and an LRU buffer of 1Mb (256 pages). Section 6.1 focuses on restricted networks and Section 6.2 on unrestricted ones.

6.1 Experiments with Restricted Networks

The first set of experiments uses the coauthorship graph of *DBLP* (database.cs.ualberta.ca/coauthorship/) [11]. Each node corresponds to an author that has published in *SIGMOD*, *VLDB*, *ICDE*, or *ACM PODS*. Two authors are linked by an edge if their names appear together in some paper. All the edge weights are set to 1 such that the length of the shortest path between two authors models their *degree of separation*. The graph is "cleaned" to form a connected network with 4,260 nodes and 13,199 edges.

Given an author q, an *ad hoc query* retrieves the RNN nodes of q, where 1) the distance corresponds to the degree of separation and 2) the result contains only authors satisfying an input condition (e.g., they should have exactly two *SIGMOD* papers). Table 1 shows the cost of *eager* and *lazy* for three query conditions (the CPU-cost is measured in

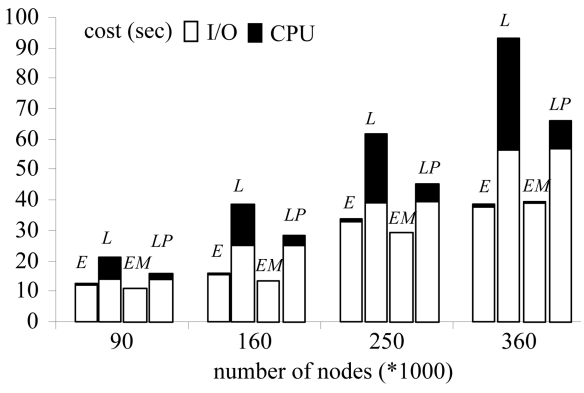

Fig. 15. Cost versus $|V|$ ($BRITE, D = 0.01, k = 1$).

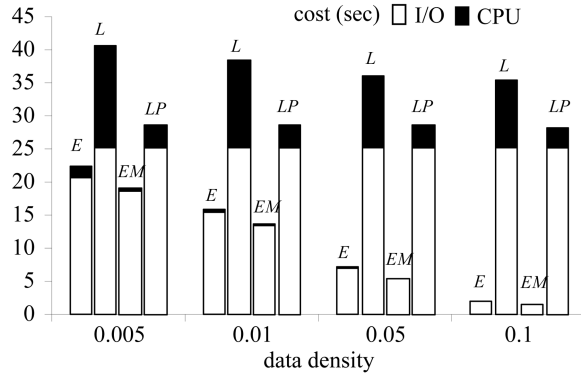

Fig. 16 Cost versus D ($BRITE, |V| = 160\text{K}, k = 1$).

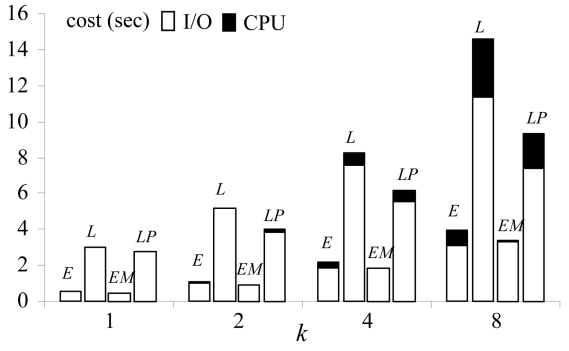

Fig. 18. Cost versus $k(SF, D = 0.01)$.

seconds). We do not include *eager-M* because materialization is not possible for ad hoc queries (i.e., the set of NNs of a node depends on the condition) and *lazy-EP* because its cost is similar to *lazy*.

The number of *SIGMOD* papers determines the selectivity of the condition. In particular, most authors have 0 papers and the selectivity increases with the number of papers leading to larger expansion around the query point, which explains the rising cost of the algorithms. *Eager* is slightly better than *lazy* in terms of I/O cost but worse in CPU time (especially for the most selective condition) because of the range-NN queries, which may visit the same node multiple times. Although, due to locality, these queries are likely to access pages already in the buffer, the overhead is reflected in the CPU-time.

Next, we randomly select "interesting" nodes (i.e., that correspond to data points) and test the performance of the algorithms on the resulting network. Table 2 shows the cost of *eager* and *lazy* as a function of the data density. As expected, the cost decreases as D increases. Although the algorithms incur very similar I/O cost, *eager* is much more CPU-intensive and the difference exceeds an order of magnitude for the low density values. This agrees with Table 1 because low density corresponds to high selectivity.

The second set of experiments simulates a P2P scenario. We use *BRITE* (www.cs.bu.edu/brite/) to generate graph topologies with average *degree* (number of edges adjacent to each node) equal to 4. Data points (i.e., peers) are located at random network nodes. Graphs generated by *BRITE* contain arbitrary connections between nodes and, consequently, the number of nodes visited during expansions increases exponentially and converges fast to the node cardinality $|V|$ since eventually all the nodes are reached with a relatively small number of hops. We call this phenomenon *exponential expansion*.

Fig. 15 illustrates the overall cost of *eager*, *eager-M*, *lazy* and *lazy-EP* as a function of $|V|$ ranging from 90K to 360K nodes ($D = 0.01, k = 1$). Abbreviations of the algorithms (E for *eager*, L for *lazy*, EM for *eager-M*, and LP for *lazy-EP*) are shown on the top of each column. The cost is measured in seconds, after charging 10ms for each random I/O (a common value used in the literature [12]). The variations of *eager* perform well because they prune the search space early, thus reducing the effect of exponential expansion. On the other hand, the pruning strategy of *lazy* fails completely because, as explained in the example of Fig. 12, *lazy* may expand some nodes that can be pruned and, subsequently, access most of the network. The extended pruning strategy of *lazy-EP* pays-off only in terms of CPU time because it leads to fewer verification queries (when compared to *lazy*).

Fig. 16 evaluates the algorithms as a function of the density in a network with $|V| = 160\text{K}$ nodes. Similar to Fig. 15, *lazy* and *lazy-EP* visit most of the network (independently of the density value) because of the exponential expansion effect. The performance of *eager* and *eager-M* improves significantly for large values of D, since each node is surrounded by more data points (and, therefore, it can be pruned by the range-NN queries). The irregularity of the network leads to high standard deviations of the I/O costs, which range from 25 percent to 67 percent of the mean values shown in the figure.

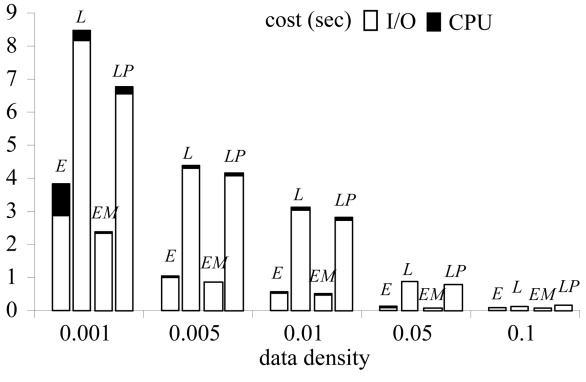

Fig. 17. Cost versus $D(SF, k = 1)$.

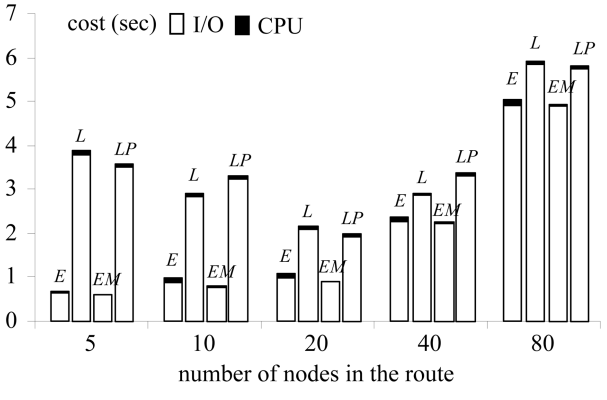

Fig. 19. Cost of continuous queries versus route size (SF, $D = 0.01, k = 1$).

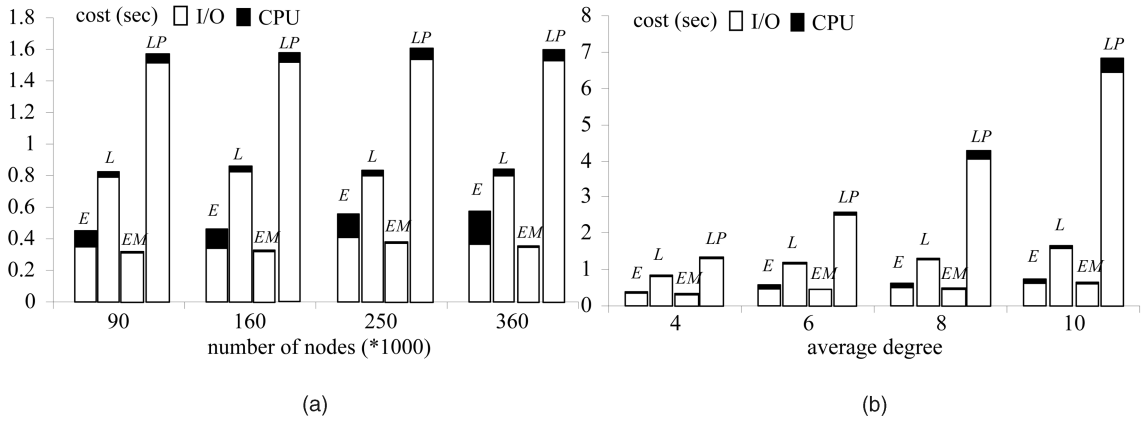

Fig. 20. *Grid maps* ($D = 0.01, k = 1$). (a) Cost versus $|V|$ (degree = 4). (b) Cost versus degree ($|V| = 160\text{K}$).

6.2 Experiments with Unrestricted Networks

For unrestricted networks we use spatial graphs. In such (planar) graphs, adjacent nodes are likely to be near in space (therefore, they do not incur exponential expansion). The first network is generated from the San Francisco (*SF*) map (www.maproom.psu.edu/dcw) after extracting its largest connected component. The number of nodes and edges of the cleaned network are 174,956 and 223,001, respectively. The coordinates of the nodes are normalized in the range $[0, 10000]^2$ and edge weights are set to the Euclidean distance of the connected nodes. Fig. 17 evaluates performance versus the data density. The density is again defined as $D = |P|/|V|$, but the data points are distributed randomly on the edges. Similar to the *DBLP* graph, *eager* is better than *lazy* in terms of I/O, but worse in CPU-time. The extended pruning technique improves the performance of *lazy* at $D \leq 0.01$, because the parallel expansion of *lazy-EP* eliminates many nodes which cannot be pruned *lazy* at low density values. Nevertheless, the performance of *lazy* improves significantly for high density (as opposed to the case of exponential expansion—see Fig. 16). Materialization is beneficial for *eager*, and *eager-M* has the lowest I/O and CPU cost. The standard deviation of the I/O costs ranges from 42 percent to 79 percent of the mean values (shown).

Fig. 18 illustrates the cost of the algorithms as a function of the number of retrieved RNNs. As expected, the performance deteriorates with k. This is most obvious for *lazy* because of the diminishing pruning effect of the verification queries. On the other hand, *lazy-EP* scales better since all discovered points are utilized for pruning. *Eager* and *Eager-M* are again the best choices. The I/O cost of *Eager-M* for accessing the materialized NNs increases with k, and exceeds that of *eager* for $k = 8$.

Fig. 19 evaluates the algorithms on continuous[2] RNN queries as a function of the route size (i.e., the number of nodes in the route). Each route is a random walk without repeated nodes. The cost of *eager* and *eager-M* grows linearly with the route size. Interestingly, the cost of *lazy* variants first decreases because a longer route leads to earlier discovery of points (close to the path), limiting the scope of verification ranges. After the length of the path exceeds 20 nodes, the cost increases because a longer route leads to more RNNs and, thus, more network expansions.

The next set of experiments applies synthetic *grid* networks [7], [5] to test the behavior of the algorithms for different node cardinalities and average degrees. The standard grid map has an average degree of 4. To generate maps with higher degree, new edges are randomly added between nearby nodes. As shown in Fig. 20a, $|V|$ does not have a serious effect on the cost because the expansion of all algorithms terminates around the query point (i.e., it does not depend on the size of the network). Fig. 20b investigates the effect of the average degree. In general, the performance deteriorates with the degree since the processing of each node must consider a larger number of adjacent nodes. *Lazy EP* scales worse than the other algorithms, because of the additional expansions incurred by the second heap H'.

As discussed in the beginning of the section, all previous experiments were performed with an LRU buffer of 1Mb (256 pages). Fig. 21 evaluates the effect of the buffer size on the performance. Due to the log scale, the CPU cost is not visible because it is dominated by the I/O cost. At buffer size = 0, every network node access incurs a page fault. The cost of *eager* in this case is much higher than that of *lazy* due to the range-NN queries. However, since these queries may visit the same node multiple times, even a small buffer improves the performance of *eager* considerably. After the

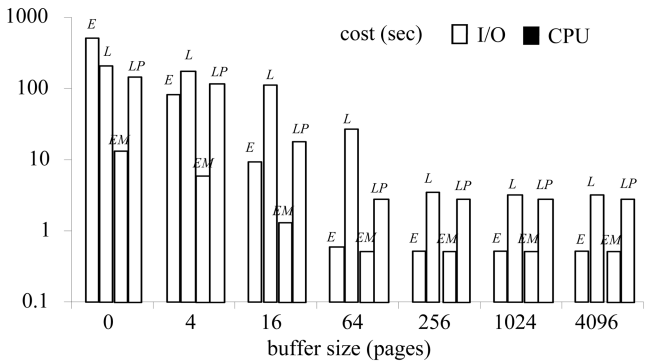

Fig. 21. Cost versus buffer size ($SF, D = 0.01, k = 1$).

2. We do not evaluate bichromatic search since, as discussed in Section 5.1, the cost of bichromayic query is similar to that of a monochromatic one (on the same network) assuming that the set of data points is Q.

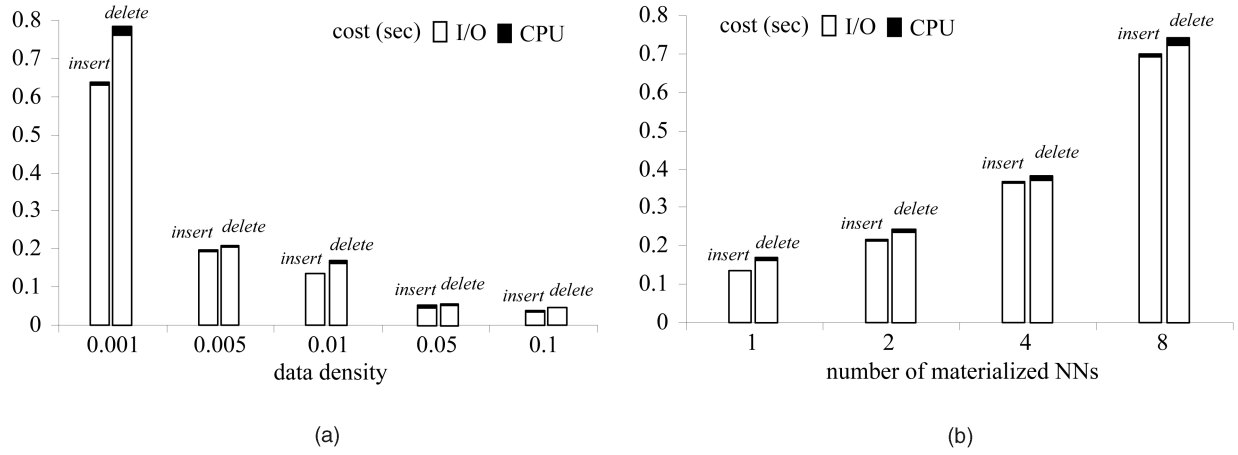

Fig. 22. Cost of updates. (a) Cost versus $D(SF, K = 1)$. (b) Cost versus $K(SF, D = 0.01)$.

buffer size reaches 64, the cost of *eager* stabilizes, indicating that 64 pages are sufficient for keeping all the visited nodes in memory. On the other hand, the performance of *lazy* stabilizes after the buffer reaches 256 pages, implying a more extensive search in the network. This experiment verifies that *eager* visits a (much) smaller set of nodes than *lazy*, but possibly many times, which explains its higher CPU overhead in the majority of experiments not involving networks with exponential expansion. Nevertheless, since the I/O cost is the dominant factor, *eager* is, in general, preferable.

In summary, the problem characteristics have a significant effect on the behavior of the algorithms. *Lazy* has lower CPU overhead than *eager* (up to two orders of magnitude for selective queries in the *DBLP* coauthorship graph), but it has higher I/O cost and it is unacceptable for networks that incur exponential expansion. *Lazy-EP* provides improvements over the basic algorithm, but in some cases, its parallel expansion may increase the I/O cost. *Eager* has a more balanced behavior and in most settings it is preferable to both lazy and *lazy-EP*. The best choice for all scenarios in which it is applicable (i.e., excluding ad hoc queries) is *eager-M*. However, it requires the materialization and maintenance of the NN points of all nodes.

The last set of experiments uses the San Francisco network to evaluate the cost of maintaining this information in the presence of object insertions and deletions. The inserted points follow the distribution of the nodes, whereas the deleted points are chosen randomly from the existing data points. Fig. 22a shows the cost of insertions and deletions as a function of the density. Deletions incur higher cost than insertions since, as discussed in Section 4.1, their expansion incurs two steps: identification of the affected nodes and then retrieval of the new NNs. Fig. 22b illustrates the cost as a function of the number K of stored NNs. As expected, the I/O overhead increases with K. Nevertheless, even for $K = 8$ each operation takes less that 0.8 seconds, and given that updates are not frequent in several practical applications, materialization is a feasible option.

7 Conclusion

This paper constitutes the first work that deals with RNN queries in large networks. We propose several techniques that can be used for a variety of queries in different types of graphs. Our evaluation covers applications including spatial maps, computer networks, and coauthorship graphs. The best and most robust algorithm (*eager-M*) relies on materialized information. In cases where the set of interesting data points is not known in advance (and, therefore, materialization is not possible), the choice of the processing method depends on the problem characteristics; the *eager* algorithm minimizes the I/O cost and is, in general, the method of choice, but it can be more CPU-intensive than *lazy* for certain networks (e.g., *DBLP* coauthorship graph).

The proposed techniques assume undirected graphs. An interesting direction for future work concerns their extensions to directed networks (e.g., spatial maps with one-way streets). In this case, the neighborhood relation is asymmetric, complicating query processing. We also intend to develop models for estimating 1) the cost of algorithms and 2) the selectivity of RNN queries in large graphs. Such models are useful both for selecting the best processing method given the problem characteristics, and optimizing complex spatial queries involving several operators. This is a particularly challenging problem, given that it has not been solved even for the apparently simpler (and more established) Euclidean version of RNN search.

Acknowledgments

Man Lung Yiu and Nikos Mamoulis were supported by grant HKU 7149/03E from Hong Kong RGC. Dimitris Papadias was supported by grant HKUST 6178/04E from Hong Kong RGC. Yufei Tao was supported by SRG Grant no 7001843 from the City University of Hong Kong.

References

[1] R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis, "Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects," *Proc. Int'l Database Eng. and Applications Symp.*, 2002.

[2] E. Chan and N. Zhang, "Finding Shortest Paths in Large Network Systems," *Proc. Ninth ACM Int'l Symp. Advances in Geographic Information Systems,* 2001.
[3] E. Chavez, G. Navarro, R. Baeza-Yates, and J. Marroquin, "Searching in Metric Spaces," *ACM Computing Surveys,* vol. 33, no. 3, pp. 273-321, 2001.
[4] E. Dijkstra, "A Note on Two Problems in Connection with Graphs," *Numeriche Mathematik,* vol. 1, pp. 269-271, 1959.
[5] C. Jensen, J. Kolar, T. Pedersen, and I. Timko, "Nearest Neighbor Queries in Road Networks," *Proc. 11th ACM Int'l Symp. Advances in Geographic Information Systems,* 2003.
[6] N. Jing, Y. Huang, and E. Rundensteiner, "Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its Performance Evaluation," *IEEE Trans. Knowledge and Data Eng.,* vol. 10, no. 3, pp. 409-432, 1998.
[7] S. Jung and S. Pramanik, "HiTi Graph Model of Topographical Roadmaps in Navigation Systems," *Proc. Int'l Conf. Data Eng.,* 1996.
[8] M. Kolahdouzan and C. Shahabi, "Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases," *Proc. 30th Int'l Conf. Very Large Data Bases,* 2004.
[9] F. Korn and S. Muthukrishnan, "Influence Sets Based on Reverse Nearest Neighbor Queries," *Proc. SIGMOD,* 2000.
[10] F. Korn, S. Muthukrishnan, and D. Srivastava, "Reverse Nearest Neighbor Aggregates Over Data Streams," *Proc. 28th Int'l Conf. Very Large Data Bases,* 2002.
[11] M. Nascimento, J. Sander, and J. Pound, "Analysis of Sigmod's Co-Authorship Graph," *SIGMOD Record,* vol. 32, no. 3, pp. 8-10, 2003.
[12] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query Processing in Spatial Network Databases," *Proc. 28th Int'l Conf. Very Large Data Bases,* 2003.
[13] I. Stanoi, D. Agrawal, and A. Abbadi, "Reverse Nearest Neighbor Queries for Dynamic Databases," *Proc. SIGMOD Workshop Research Issues in Data Mining and Knowledge Discovery,* 2000.
[14] A. Singh, H. Ferhatosmanoglu, and A. Tosun, "High Dimensional Reverse Nearest Neighbor Queries," *Proc. Conf. Information and Knowledge Management (CIKM),* 2003.
[15] S. Shekhar, A. Kohli, and M. Coyle, "Path Computation Algorithms for Advanced Traveler Information System," *Proc. Int'l Conf. Data Eng.,* 1993.
[16] S. Shekhar and J. Yoo, "Processing In-Route Nearest Neighbor Queries: A Comparison of Alternative Approaches," *Proc. 11th ACM Int'l Symp. Advances in Geographic Information Systems,* 2003.
[17] I. Stanoi, M. Riedewald, D. Agrawal, and A. Abbadi, "Discovery of Influence Sets in Frequently Updated Databases," *Proc. 28th Int'l Conf. Very Large Data Bases,* 2001.
[18] Y. Tao, D. Papadias, and X. Lian, "Reverse kNN Search in Arbitrary Dimensionality," *Proc. 30th Int'l Conf. Very Large Data Bases,* 2004.
[19] C. Traina, A. Traina, B. Seeger, and C. Faloutsos, "Slim-Trees: High Performance Metric Trees Minimizing Overlap between Nodes," *Proc. Seventh Int'l Conf. Extending Database Technology,* 2000.
[20] C. Yang and K. Lin, "An Index Structure for Efficient Reverse Nearest Neighbor Queries," *Proc. Int'l Conf. Data Eng.,* 2001.
[21] M. Yiu and N. Mamoulis, "Clustering Objects on a Spatial Network," *Proc. SIGMOD,* 2004.

Man Lung Yiu received the bachelor's degree in computer engineering from the University of Hong Kong, China, in 2002. He is currently a PhD candidate in the Department of Computer Science at the University of Hong Kong. His research interests include databases and data mining.

Dimitris Papadias is an associate professor in the Computer Science Department, Hong Kong University of Science and Technology (HKUST). Before joining HKUST in 1997, he worked and studied at the German National Research Center for Information Technology (GMD), the National Center for Geographic Information and Analysis (NCGIA, Maine), the University of California at San Diego, the Technical University of Vienna, the National Technical University of Athens, Queen's University (Canada), and University of Patras (Greece). He has published extensively and been involved in the program committees of all major database conferences, including SIGMOD, VLDB, and ICDE.

Nikos Mamoulis received a diploma in computer engineering and informatics in 1995 from the University of Patras, Greece, and the PhD degree in computer science in 2000 from the Hong Kong University of Science and Technology. Since September 2001, he has been an assistant professor in the Department of Computer Science, University of Hong Kong. In the past, he has worked as a research and development engineer at the Computer Technology Institute, Patras, Greece, and as a postdoctoral researcher at the Centrum voor Wiskunde en Informatica (CWI), the Netherlands. His research interests include spatial, spatio-temporal, multimedia, object-oriented, semistructured databases, and constraint satisfaction problems.

Yufei Tao received the diploma from the South China University of Technology in August 1999, and the PhD degree from the Hong Kong University of Science and Technology in July 2002, both in computer science. After that, he spent a year as a visiting scientist at the Carnegie Mellon University. Currently, he is an assistant professor in the Department of Computer Science, City University of Hong Kong. Dr. Tao is the winner of the Hong Kong Young Scientist Award 2002 from the Hong Kong Institution of Science. His research includes query algorithms and optimization in temporal, spatial, and spatio-temporal databases.