International Journal of Grid and Distributed Computing
Vol. 9, No. 11 (2016), pp.187-198
http://dx.doi.org/10.14257/ijgdc.2016.9.11.16

# Efficient kNN Computation for Multiple Query Points on Road Network

Xian Tang

*School of Economics and Management, Yanshan University, Qinhuangdao, 066000, China*
*txianz@163.com*

***Abstract***

*Finding k nearest neighbours to one query point on road network is the typical kNN problem, and attracts lots of research interests in recent years with the development of mobile technologies. In this paper, we study a variant of the kNN problem, namely mkNN, which returns k objects for m query points. We firstly propose a baseline algorithm, which solves the mkNN problem in naive way. We then propose an optimized algorithm, which avoids the redundant computation of the baseline algorithm by organizing objects using index to firstly get candidate objects, such that to reduce the number of visited nodes of the given road network. Our third algorithm further improves the performance by using a tighter threshold to greatly reduce the number of visited nodes. We make detailed comparison between the three algorithms and show their benefits w.r.t. the mkNN problem.*

***Keywords**: kNN; Multiple Query Points; Road Network*

## 1. Introduction

With the development of mobile technologies, more and more mobile devices are equipped with global positioning system (GPS) chips, and users are more dependent on their mobile devices to access services they are searching for. Given a query point *q* on a road network *G*, a *k*NN query finds *k* nearest neighbors (*k*NN) to *q* [1, 2], or on the contrary, finds one nearest neighbor *v* for *k* query points, such that the sum of their distances to *v* is minimal [3-6], which is a hot research issue during the past years [1-13], due to its numerous applications in practice. For example, a tourist may want to search for *k* nearest hotels while walking in a city, a driver may want to find out *k* nearest gas stations during driving.

Compared with the typical *k*NN problem, we study the problem of *k*NN query with multiple query points, which finds *k* neighbors such that the sum of the distances of each neighbor to its nearest query point is minimal, and is called as *mk*NN problem. *mk*NN has many important applications. *E.g.*, a chain supermarkets in a city deliveries rewards to *k* star members from its *m* storehouses, a E-commerce company deliveries goods to a set of users from its warehouses distributed in different places, *etc*. We firstly propose a baseline algorithm to compute the *k*NN for each query point, namely *mk*NN, which uses Dijkstra algorithm to get the *k*NN for each query point, and then select the final *k*NN for all query points. Considering that *mk*NN suffers from redundant computation, we propose to use R-tree [14] to find *k*NN candidates based on the Euclidean distance, denoted as *mk*NN-O, then verify each of them using A* algorithm [10]. Finally, we make further optimization on *mk*NN-O by proposing a tighter upper bound to reduce the number of candidate nodes, such that to improve the overall performance.

## 2. Preliminaries and Related Work

### 2.1. Preliminaries

A road network can be modeled as a weighted graph $G = (V, E, W, O)$, where $V$ is the set of nodes representing a set of crossing, $E$ the set of edges denoting a set of roads between crossings, $W$ the set of weights on edges of $E$, each $w_e \in W$ denotes the length of a road $e \in E$, and $O$ is the set of to be searched objects in $G$, such as gas stations, users, and schools *etc*.

A query $Q = \{q1, q2, \ldots, qm\}$ is a set of points representing a set of locations. In a road network, they are given as a set of nodes of $V$. $d(q, o)$ is the length of the shortest path from query point $q$ to object $o$ in $G$, and $d_E(q, o)$ is the distance between $q$ and $o$ in Euclidean space. Hereafter, $d_E(q, o)$ is also called as Euclidean distance, and $d(q, o)$ is called as distance or road network distance if without ambiguity.

**Definition 1. (The *k*NN Problem)** Given a query point $q$ and query objects $O$, find $k$ objects $D = \{o1, o2, \ldots, ok\}$ satisfying that $d(q, o_1) \leq d(q, o_2) \leq \ldots \leq d(q, o_k)$, such that $\forall o \in D, o' \in O \setminus D, d(q, o) \leq d(q, o')$

**Definition 2. (The *mk*NN Problem)** Given a query $Q = \{q1, q2, \ldots, qm\}$ and query objects $O$, find $k$ objects $D$ from $O$, such that $\forall o \in D$; $o' \in O \setminus D$, there exists $q \in Q$, $d(q, o) \leq d(q, o') \wedge d(q, o) \leq d(q', o)$, where $q' \in Q \setminus \{q\}$.

### 2.2. Related Work

Generally speaking, existing algorithms on *k*NN computation can be classified into two categories: (C1) finding $k$ nearest objects for one query point [1, 2], and (C2) finding one object for multiple query points, such that the sum of the distances from each query point to the object is minimal [3-6]. Both kinds of algorithms are also extended with the restriction of query keywords [7-9], where the core operation is how to find the nearest neighbor $o$ for a given query point $q$.

INE [11] uses Dijkstra to visit each object $o \in O$ in ascending order w.r.t. the distance $d(q, o)$. Let $o_k$ be the $k^{th}$ nearest object of $q$, the benefit of INE lies in that it does not need to visit any object $o' \in O$, such that $d(q, o_k) \leq d(q, o')$. However, it is inefficient due to that to find the $k$ objects, it needs to visit all nodes of $G$ with distances less than $d(q, o_k)$.

IER [11] improves INE by firstly finding a candidate *k*NN $D1 = \{o_1^1, o_2^1, \ldots, o_k^1\}$ based on Euclidean distance [13], then computes the exact distance $d(q, o_i^1)$ for each $o_i^1 \in D1$. Without loss of generality, assume that $d(q, o_1^1) \leq d(q, o_2^1) \leq \ldots \leq d(q, o_k^1)$, IER uses $d(q, o_k^1)$ as the radius to find a set of objects, such that the Euclidean distance of each object to $q$ is not greater than $d(q, o_k^1)$. It then computes the exact distance between $q$ to each object on road network, and finds the second *k*NN candidate $D2 = \{o_1^2, o_2^2, \ldots, o_k^2\}$ satisfying $d(q, o_1^2) \leq d(q, o_2^2) \leq \ldots \leq d(q, o_k^2)$. As $d(q, o_k^2) \leq d(q, o_k^1)$, in the next iteration, it uses $d(q, o_k^2)$ as the radius to find a smaller set of objects, and then updates the *k*NN candidate $Dj$. It stops when $Dj = Dj\text{-}1$ and returns $Dj$ as the *k*NN result. Note that IER uses A* algorithm [10] to get the exact distance on road network for $q$ and an object $o$. Even though IER avoids visiting large number of nodes as INE does, it suffers from repeatedly visiting the same set of nodes when calling the A* algorithm to update *k*NN candidate.

[12] Proposed a G-Tree index to accelerate *k*NN computation. The G-Tree index is generated by recursively decomposing the given road network $G$ into a set of

smaller sub-graphs. G-Tree is efficient for *k*NN computation, but suffers from maintaining a huge index.

## 3. The Baseline Algorithm

Given a query $Q = \{q1, q2, \ldots, qm\}$, the baseline algorithm for *mk*NN, namely mkNN as shown by Algorithm 1, uses a heap $C$ to buffer $k$ candidates, where each candidate $c_i \in C$ has two variables, $c_i.o$ corresponds to an object, and $c_i.d$ is the road network distance between a query point and $c_i.o$. As shown by Algorithm 1, mkNN computes *k*NN of each query point $q$ by calling Procedure getkNN () in line 3 and updates the candidate set $C$ during the processing. The $k$ candidates in $C$ are sorted by their road network distance. After processing all query points, the final *k*NN for the $m$ query points is returned in line 4.

Procedure getkNN() works as follows. It uses a priority queue $S$ to maintain a set of nodes that are not expanded yet. Before processing, it inserts the current query point $q$ into $S$ with distance equals 0 (line 6). During the processing, if $S$ contains nodes that are not expanded, it removes a node, i.e., $c$, from $S$ (line 8). Here node $c.o$ is represented by $c = <c.o, c.d>$. If the distance $c.d \geq c_k.d$, we directly terminate the processing of $q$. Otherwise, if $c.o$ is not visited before, we check whether $c.o$ is an object in line 11. If the answer is true, we update the candidate set $C$ using $c.o$ and increase the number of candidates by 1 in line 13, and terminate the processing in line 14 if $count = k$. After that, we insert $c.o$'s adjacency nodes that are not visited before into $S$ if their distance is less than $c_k.d$ (lines 15-19).

**Example 1.** Let $k = 3$, consider the query $Q = \{q_1, q_2, q_3\}$ on the road network $G$ in Figure. 1, where the set of objects is $O = \{o_1, o_2, o_6\}$. Algorithm 1 firstly processes $q_1$. As the distance of $q_1$'s adjacency nodes to $q_1$ is less than $c_k.d = \infty$, $q_1$'s adjacency nodes, i.e., $v_2$ and $v_3$, are inserted into $S$. After that, as $d(q_1, v2) = 3 < d(q_1, v3) = 5$, $v_2$ is firstly removed from $S$ and marked as visited, then its adjacency nodes, i.e., $v_1$ and $v_4$, are inserted into $S$. After processing $v_3$, we find the first object $o_1$ and use it to update the candidate set $C$, then $C = \{< o_1, 9 >\}$. The following processing is similar. After processing $q_1$, we have $C = \{<o_1, 9>, <o_2, 10>, <o_3, 11>\}$. The next to be processed query point is $q_2$, and we find that the nearest object of $q_2$ is $o_4$ and $d(q_2, o_4) = 5$, thus we use it to update $C$, then $C = \{<o_4, 5>, <o_1, 9>, <o_2, 10>\}$. Similarly, as the distance from the second nearest object of $q_2$, i.e., $o_6$, is $d(q_2,o_6) = 6 < c_3.k = 10$, $C$ is updated again and becomes $C = \{<o_4, 5>, <o_6, 6>, <o_1. 9>\}$. In the same way, after processing all query points, we have $C = \{<o_4, 5>, <o_6, 6>, <o_1, 7>\}$ and return the three objects as the final results.

```
Algorithm 1: mkNN(Q = {q_1, q_2, …, q_m})
1    $C \leftarrow \{c_1, c_2, \ldots, c_k\}$, where each $c_i = <o, d>$, $c_i.d = \infty$
2    for each ($q_i \in Q$) do
3        getkNN($q_i$)
4    return $C$
     Procedure getkNN($q$)
5    $count \leftarrow 0$; visit[$v_1 \ldots v_n$]$\leftarrow$FALSE
6    Enqueue($S$,<$q$, 0>)
7    while (isEmpty($S$)=FALSE) do
8        $c \leftarrow$Dequeue($S$)
9        if ($c.d \geq c_k.d$) then return
10       if (visit[$c.o$]=FALSE) then
11           if ($c.o \in O$) then
12               update($C$, $c.o$)
13               $count \leftarrow count + 1$
```

```
14            if (count = k) then return
15        for each (adjacency node v of c.o) do
16            if (visit[v] = FALSE) then
17                d←c.d + w(c.o, v)
18                if (d < c_k.d) then
19                    Enqueue(S,<v,d>)
20        visit[c.o]←TRUE
```

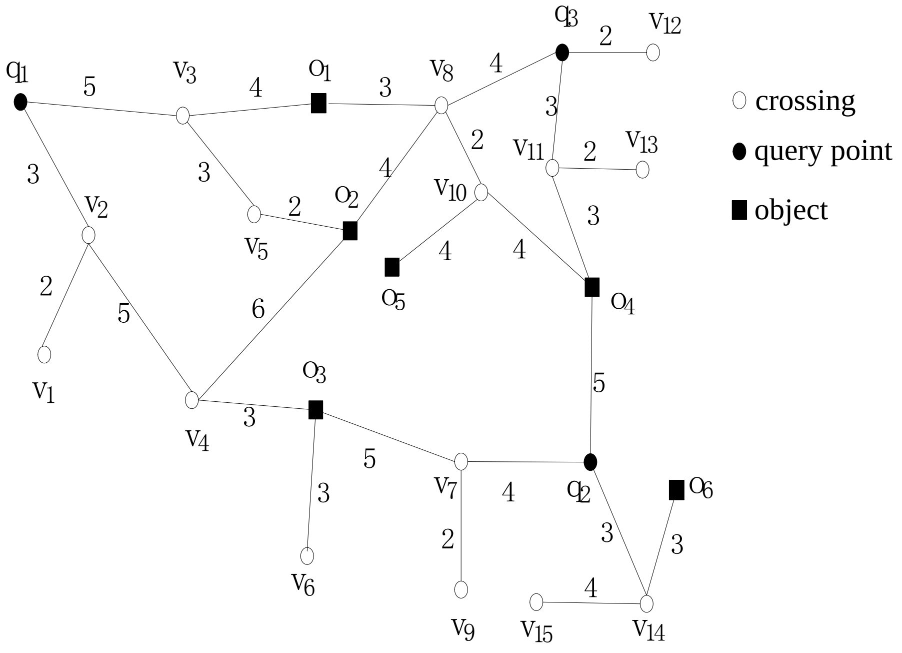


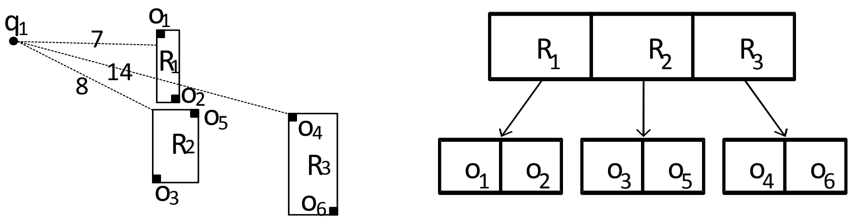
**Figure 1. A Sample Road Network G**

## 4. The Optimized Algorithms

As shown by Algorithm 1, the blindness of *mk*NN lies in that it needs to visit all nodes of $V$ with distances less than $d(q, c_k.d)$, therefore when applied in practice, it may be inefficient. Considering this problem, we propose in this section two optimized algorithms to improve the overall performance.

### 4.1. The mkNN-O Algorithm

The intuition behind mkNN-O is: the distance of two nodes in Euclidean space is the lower bound of that on road network, i.e., $d(q, o) \geq d_E(q, o)$, and an object near to $q$ in Euclidean space has higher probability that it is also near to $q$ on road network. Moreover, as R-Tree index [14] is typically used to efficiently find *k*NN in Euclidean space, we use it to get *k*NN candidates of a query point without actually visiting nodes of road network. Figure. 2 (b) shows the R-Tree index of objects in the road network in Figure. 1, where each internal node represents a set of rectangular regions, and each leaf node is an object.



(a) Regions of objects in G of Figure. 1 (b) The R-Tree index R

**Figure 2. Illustration of the R-Tree index**

Based on the above observation, the basic idea of the optimized algorithm is: process query points one by one. For each query point, get its *k*NN in Euclidean space using R-Tree index, then compute their road network distances using A* algorithm to update the candidate set. During the processing, if the next nearest object has Euclidean distance larger than the road network of the $k^{th}$ candidate, we directly terminate the processing.

As shown by Algorithm 2, the difference between the optimized algorithm, i.e., *mk*NN-*O*, and *mk*NN lies in that it calls getkNN-O () to find *k*NN for each query point. Lines 1, 2 and 4 are same as that of mkNN and are therefore omitted to save space.

GetkNN-O () works as follows. It takes each rectangular region as an object, and pushes all regions of the R-Tree *R*'s root node into *S* in lines 5-6. After that, in each iteration, it pops out an entry *c* from *S* in line 8, then processes it in three steps: (1) if the popped region has an Euclidean distance greater than $c_k.d$ (line 9), it means that the road network of all objects to *q* is greater than $c_k.d$, then we directly terminate the processing. Note that all objects in *S* are not needed to be processed further due to that S is a priority queue, and the first popped entry has smallest Euclidean distance to *q* than that of other entries in S; (2) if *c.e* is a leaf node of *R* (lines 10-13), we find the road network distance of *c.e* to *q* using the A* algorithm, and update the candidate set if $d(q, c.e) < c_k.d$; (3) if *c.e* is an internal node of *R* (lines 15-17), we push child nodes of *c.e* into *S* if their Euclidean distances are smaller than $c_k.d$.

Compared with getkNN(), getkNN-O() does not need to visit all nodes of *G* within distance smaller than $c_k.d$, due to the assistance of R-Tree index and A* algorithm.

```
Algorithm 2: mkNN-O(Q = {q_1, q_2, …, q_m})
3            getkNN-O(q_i)
Procedure getkNN-O(q)
5        for each (entry e ∈ root(R)) do
6            Enqueue(S, <e, d_E(q, e) >)
7        while (isEmpty(S)=FALSE) do
8            c←Dequeue(S)
9            if (c.d ≥ c_k.d) then return
10           if (c.e is a leaf node of R) then
11               d←A*(q, c.e)
12               if (d < c_k.d) then
13                   update(C, c.e)
14           Else
15               for each (child node e' of c.e) do
16                   if (d_E(q, e') < c_k.d) then
17                       Enqueue(S, <e', d_E(q, e') >)
```

**Example 2.** Consider the same query of Example 1 again. The Euclidean distance of $q_1$ to each region is shown in Figure. 2 (a). We firstly process $q_1$, and push $R_1$ to $R_3$ into *S* in line 6 with Euclidean distances 7, 8 and 14, respectively. In the first iteration, $c$ =<$R_1$, 7> is popped out from $S$. As $R_1$ is not a leaf node, its child nodes, i.e., $o_1$ and $o_2$, are pushed into $S$ with Euclidean distances 8 and 9, respectively. Then $c$ =<$R_2$, 8 > is popped out from $S$, and then its child nodes, i.e., $o_3$ and $o_5$, are pushed into $S$ with Euclidean distances 9 and 10, respectively. After that, $c$ =<$o_1$, 8> is popped out from $S$. Since $o_1$ is a leaf node of $R$, we compute its road network distance $d$ ($q$, $o_1$) = 9 using A* algorithm, and update the candidate set to $C$ = {<$o_1$, 9>}. Similarly, after $c$ =<$o_2$, 9 > is popped out from $S$, we update the candidate set to $C$ = {<$o_1$, 9>, <$o_2$, 10>}. After $c$ =<$o_3$, 9> is popped out and processed, the candidate set becomes $C$ = {<$o_1$, 9>, <$o_2$, 10>, <$o_3$, 11>}, and the final *k*NN of $q_1$ is $C$ = {<$o_1$, 9>, <$o_2$, 10>, <$o_3$, 11>}. Then we process $q_2$ and $q_3$ as $q_1$, the difference is that $c_k.d$ = 11 now, rather than 1 before processing $q_1$, we omit the detailed description.

## 4.2. The mkNN-O+ Algorithm

Even though mkNN-O does not need to visit all nodes within road network distance less than the $k^{th}$ candidate object, it needs to call the A* algorithm to get the road network distance between $q$ and each object with Euclidean distance less than the $k^{th}$ candidate object, which still may result in inefficiency due to repeatedly calling of the A* algorithm to visit nodes in the road network, the reason lies in that mkNN-O takes road network distance of the $k^{th}$ candidate object as the threshold.

Considering the above problem, we propose the second optimized algorithm, namely mkNN-O+, which computes *k*NN of $m$ query points in $k$ iterations. In the $j^{th}$ iteration, it takes the road network distance of the $j^{th}$ candidate object as the threshold, such that to reduce the number of calling of A* algorithm and the cost of searching R-Tree index.

To do this, as shown by Algorithm 3, we associate each query point $q_i$ a variable $t_{ed}$ denoting the Euclidean distance within which all objects have been processed w.r.t. $q_i$, and set it to 0 initially. In lines 2 to 5, we call getkNN-O+ ($q_i$, $j$) only if the road network distance of the $j^{th}$ object is greater than $q_i.t_{ed}$. The correctness is based on the following result.

**Theorem 1.** When processing $q_i$ in the $j^{th}$ iteration, if $c_j.d \leq q_i.t_{ed}$, then there does not exist new object that can be found with road network distance to $q_i$ smaller than $c_j.d$.

Proof. As $q_i.t_{ed}$ is the maximum Euclidean distance of the candidate objects found when processing $q_i$ in the ($j$-1)$^{th}$ iteration, all objects with Euclidean distances $\leq c_j.d$ has been processed already for $q_i$. Therefore, no new candidate objects can be found.

Procedure getkNN-O+ () works as follows. It only pushes into $S$ nodes of the R-Tree $R$ with Euclidean distance less than $c_j.d$ (lines 8-10). In lines 11-23, for each popped out node $c$ =<$e$, $d$>, if $c.e$ is a leaf node of $R$, we call the A* algorithm to get the road network distance as Algorithm 2 does in line 15. After that, we update the candidate set $C$ in line 17 if the new object has road network distance smaller than $c_k.d$, then we update the maximum Euclidean distance denoting within which all objects have been processed in lines 18-19. If $c.e$ is not a leaf node of $R$, in lines 21-23, we push into $S$ $c.e$'s child nodes satisfying their Euclidean distance is less than $c_j.d$. Finally, in line 24, we set $q.t_{ed}$ as $d_{\max}$.

```
Algorithm 3: mkNN-O+(Q = {q_1, q_2, …, q_m})
1    C←{c_1, c_2, …, c_k}, where each c_i =<o, d>, c_i.d = ∞; q_i.t_ed = 0
2    for (j = 1 to k) do
3        for each (q_i ∈ Q) do
4    if (c_j.d > q_i.t_ed) then
5    getkNN-O+(q_i, j)
6    return C
Procedure getkNN-O+(q, j)
```

```
7   $d_{\max}\leftarrow 0$
8   **for each** (entry $e\in root(R)$) **do**
9       **if** ($d_E(q, e) < c_j.d$) **then**
10          Enqueue($S$, <$e$, $d_E(q, e)$>)
11  **while** (isEmpty($S$)=FALSE) **do**
12      $c\leftarrow$Dequeue($S$)
13      **if** ($c.d\geq c_j.d$) **then** return
14      **if** ($c.e$ is a leaf node of $R$) **then**
15          $d\leftarrow$A*($q$, $c.e$)
16          **if** ($d < c_k.d$) **then**
17              update($C$, $c.e$)
18              **if** ($c.d > d_{\max}$) **then**
19                  $d_{\max}\leftarrow c.d$
20      **else**
21          **for each** (child node $e'$ of $c.e$) **do**
22              **if** ($d_E(q, e') < c_j.d$) **then**
23                  Enqueue($S$, <$e'$, $d_E(q, e')$ >)
24  $q.t_{ed}\leftarrow d_{\max}$
```

**Example 3.** Consider the same query of Example 1 again. In the first iteration, we firstly process $q_1$, then get three candidate objects $C$ = {<$o_1$,9>,<$o_2$,10 >,<$o_3$,11>} and set $q_1.t_{ed}$ = 9. After that, $c_1.d$ = 9 and we process $q_2$. We only process nodes of $R$ with Euclidean distance smaller than 9. The first found object is $o_6$ with Euclidean distance 3. As $d(q_2, o_6)$ = 6, we use it update the candidate set $C$. As $c_1.d$ = 6 now, the next found object is $o_4$ with both Euclidean and road network distances equals 5. Then we use it to update $C$ and $c_1$.d = 5. After that, no one in $S$ satisfies Euclidean distance smaller than 5, the processing of $q_2$ stops and $q_2.t_{ed}$ = 5. The last processed query point is $q_3$. The unique satisfied object is $o_4$ with Euclidean distance 5 and road network distance 6, which is already in $C$ with road network distance equals 5, therefore $C$ is not updated, and $C$ = {<$o_4$, 5>,<$o_6$, 6>, <$o_1$, 9>}. After that, we begin the second iteration. As $c_2.d = 6 < q_1.t_{ed}$ = 9, we don't need to process $q_1$ in this iteration according to Theorem 1. The second processed query point is $q_2$ and we don't find satisfied objects for $q_2$. For $q_3$, we find $o_5$, $o_2$, $o_1$ with all Euclidean distances 6 and road network distances 10, 8 and 7, respectively. Then update $C$ using $o_1$, and update $q_3.t_{ed}$ = 6. After this iteration, $C$ = {<$o_4$, 5>, <$o_6$, 6>, <$o_1$, 7>}. Finally, we return $o_4$, $o_6$ and $o_1$ as the final results. Compared with mkNN-O which needs to call the A* algorithm 18 times, mkNN-O+ reduces the calling times to 9.

## 5. Experiment

### 5.1. Experimental Setup

All experiments were run on a PC with AMD Athlon(tm) II X2 270 3.4 GHz CPU, 2 GB memory, and the operating system is Windows 7. The compared algorithms include mkNN, mkNN-O and mkNN-O+ implemented using Microsoft VC++.

We use five real datasets to test the performance of different algorithms, include CA (21,047 nodes and 21,692 edges), OL (6,105 nodes and 7,034 edges), TG (18,262 nodes and 23,873 edges), NA (175,813 nodes and 179,178 edges) and SF (174,955 nodes and 223,000 edges). For each dataset, we randomly generate 1,000 queries, and report the average result of each query by executing all queries 100 times.

We make comparison based on the following metrics: (1) running time, (2) the number of visited nodes to process each query; (3) the number of calling times of the A* algorithm, which is used for mkNN-O and mkNN-O+. Besides, we test the overall performance of the three algorithms from three aspects by: (1) changing the number of results, i.e., $k$'s value; (2) changing the number of objects, which is denoted as $s = |V|/|O|$, where $V$ is the set of nodes of the given road network $G$, $O$ is the set of objects in $G$. *E.g.*, $s = 100$ means that the number of objects in $G$ is one percent of the number of nodes of $G$; (3) changing the number of query points, i.e., $m$'s value. The default values of $k$, $s$ and $m$ are 5, 1000 and 10, respectively.

## 5.2. Performance Comparison

Figure. 3 shows the comparison of the running time of the three algorithms with the default values of $k$, $s$ and $m$, from which we have the following observations: (1) the performance of each algorithm is not affected by the size of the given road network, this is because if the value of $s$ is given, then running time of each algorithm is independent of the given graph; (2) mkNN-O cannot beat mkNN on all datasets; (3) by reducing the number of calling times of A* algorithm, mkNN-O+ achieves the best performance. We show the detailed comparison in the following by changing the value of each of the three parameters, i.e., $k$, $s$ and $m$. Note that when we show the impact of each parameter, the other two are with default values.

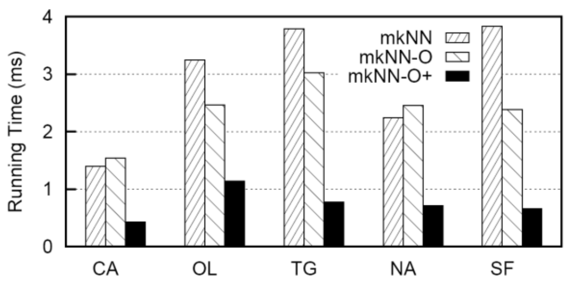


**Figure 3. Comparison of Average Running Time**

**Impacts of *k*'s value** Figure. 4 shows the performance changes with the change of $k$'s value, from which we know that: (1) the running time of all algorithms will increase with the increase of $k$ (Figure. 4 (a)); (2) mkNN-O+ works best for each of $k$'s values; (3) mkNN-O only works better than mkNN when $k = 5$; for other cases, mkNN is better than mkNN-O. This can be further explained by the number of visited nodes shown in Figure. 4 (b), from which we know that mkNN-O visits less nodes of the road network than mkNN when $k \leq 10$, thus works better than mkNN when $k = 5$. Even though it visits less nodes than mkNN when $k = 10$, it is still beaten by mkNN due to that it needs to afford the cost of visiting the R-Tree index. As a comparison, mkNN-O+ visits much less nodes than mkNN-O for all cases, thus achieves best performance. Figure 4 (c) is the comparison of mkNN-O and mkNN-O+ on the number of calling times of the A* algorithm, which also explains why mkNN-O+ works much better than mkNN-O.

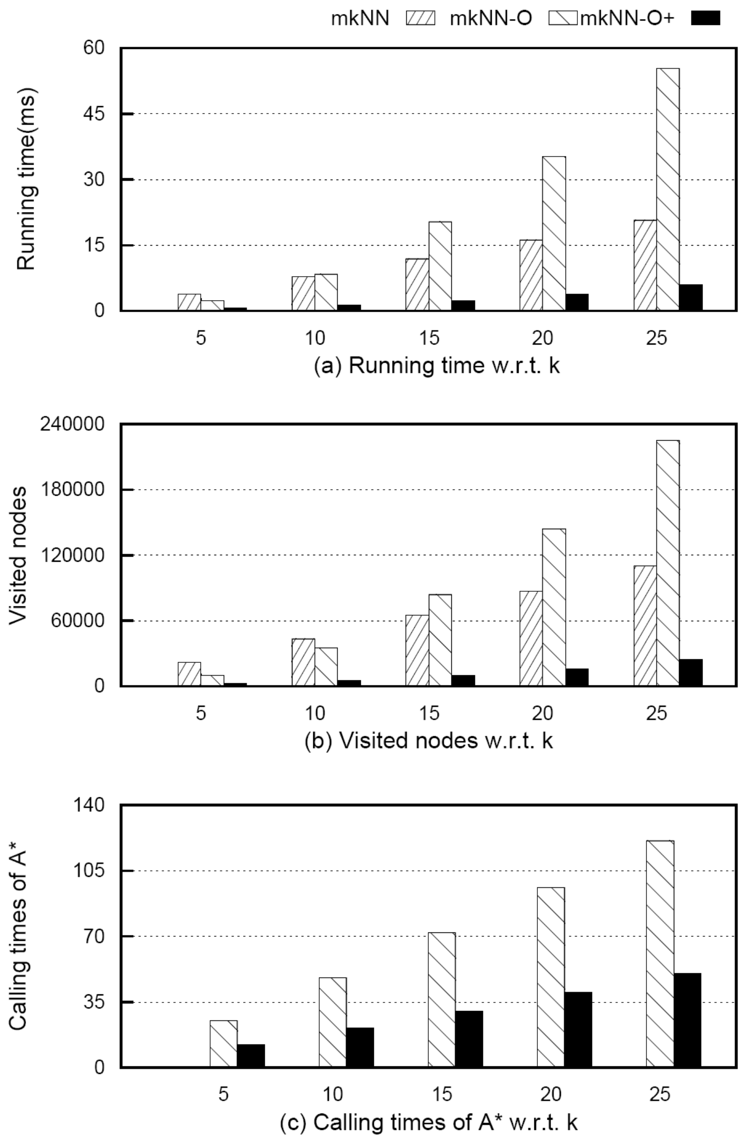
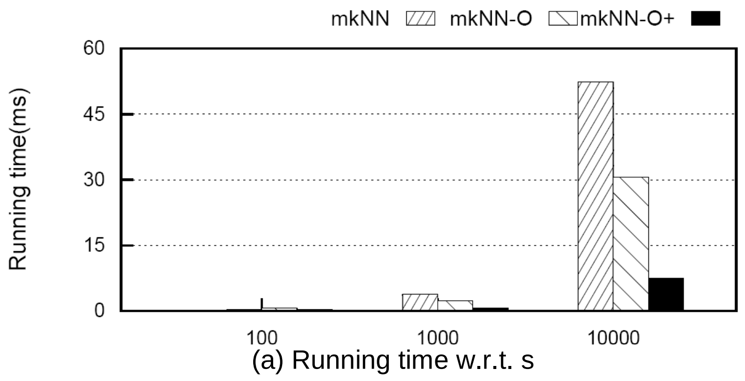
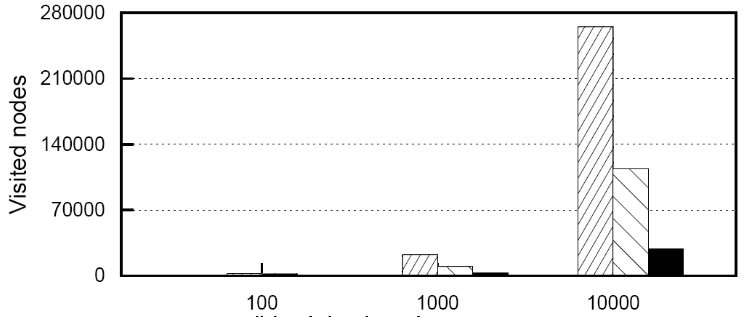
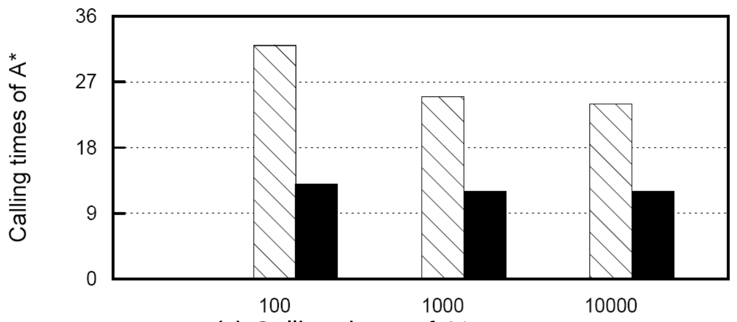


**Figure 4. Performance Comparison with the Change of the Number of Results**





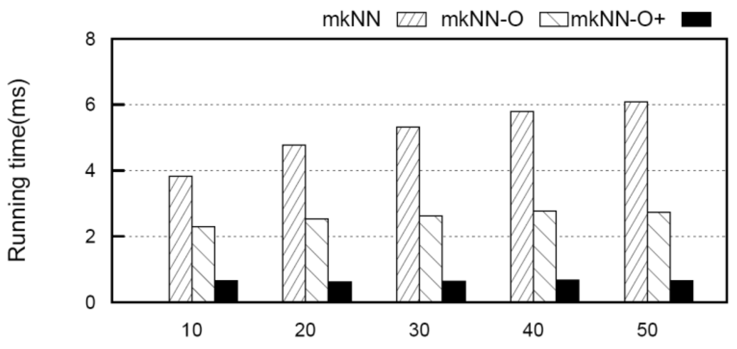
(b) Visited nodes w.r.t. s



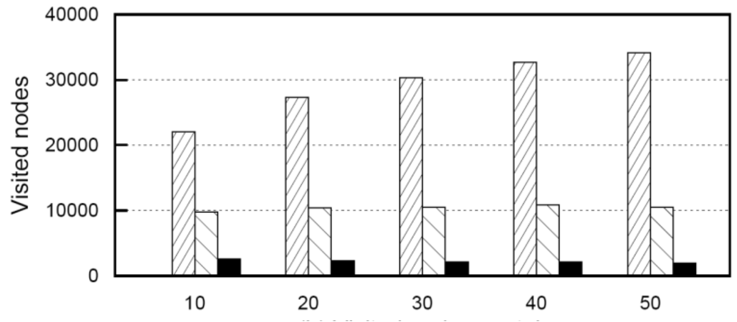
(c) Calling times of A* w.r.t. s

**Figure 5. Performance Comparison with the Change of the Number of Objects**

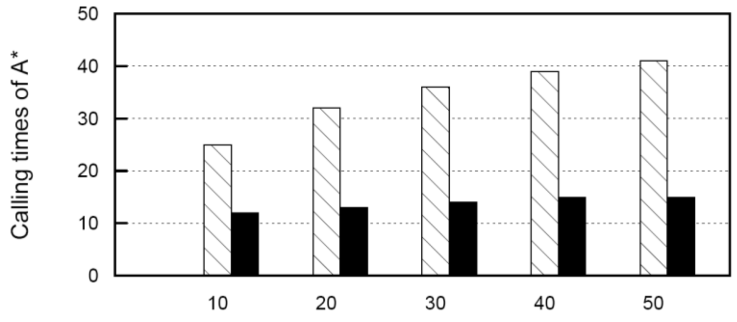
**Impacts of *m*'s value** Figure. 6 shows the performance changes with the change of *m*'s value, from which we know that: (1) the running time of mkNN increases with the increase of *m*'s value due to its blindness on the direction from each query point to objects. (2) the performance of both mkNN-O and mkNN-O+ is not affected by *m*'s value, and mkNN-O+ works much better than both mkNN and mkNN-O for all cases due to that it visits much less number of nodes of the road network than the other two and the calling times of the A* algorithm for mkNN-O+ is also much less than that of mkNN-O.



(a) Running time w.r.t. m



(b) Visited nodes w.r.t. m



(c) Calling times of A* w.r.t. m

**Figure 6. Performance Comparison with the Change of the Number of Query Points**

## 6. Conclusions

Given a road network, we studied the problem of returning *k* objects w.r.t. *m* query points, which is a variant of the classical *k*NN problem.We firstly proposed a baseline algorithm, namely mkNN, to solve it in a naive way. We then proposed an optimized algorithm, i.e., mkNN-O, to avoid the redundant computation of mkNN on visiting large number of nodes. Considering that mkNN-O is inefficient due to that it uses the $k^{th}$ object's road network distance as the threshold, we proposed the third algorithm, i.e., mkNN-O+, which gets the *k* objects in *k* iterations. In the $j^{th}$ iteration, it uses the $j^{th}$ object's road network distance as the threshold to reduce the number of calling times of the A* algorithm, such that to greatly reduce the number of visited nodes. Our experimental results show that mkNN-O+ works best on all cases.

## References

[1] C. S. Jensen, J. Kolrvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In GIS, pages 1-8, 2003.
[2] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In VLDB, pages 802-813, 2003.
[3] M. Yiu, N. Mamoulis, and D. Papadias. Aggregate nearest neighbor queries in road networks[J]. TKDE, 17(6), 2005, 820-833.
[4] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In ICDE, pages 301-312, 2004 .
[5] D. Yan, Z. Zhao, and W. Ng. Efficient algorithms for finding optimal meeting point on road networks[J]. PVLDB, 4(11), 2011, 968-979.
[6] Weiwei Sun, Chong Chen, Baihua Zheng, Chunan Chen, Liang Zhu, Weimo Liu. Merged Aggregate Nearest Neighbor Query Processing in Road Networks. In CIKM, pages 2243-2248, 2013.
[7] Joao B. Rocha-Junior, Kjetil Norvag: Top-k spatial keyword queries on road net-works. In EDBT, pages 168-179, 2012.

[8] G. Li, J. Feng, and J. Xu. Desks: Direction-aware spatial keyword search. In ICDE, pages 474-485, 2012.
[9] R. Zhong, J. Fan, G. Li, K.-L. Tan, and L. Zhou.Location-aware instant search. In CIKM, pages 385-394, 2012.
[10] K. Deng, X. Zhou, H. T. Shen, S. W. Sadiq, and X. Li. Instance optimal query processing in spatial networks. VLDB J., 18(3), 2009, 675-693.
[11] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In VLDB, pages 802-813, 2003.
[12] Ruicheng Zhong, Guoliang Li , Kian-Lee Tan, and Lizhu Zhou. G-Tree: An Efficient Index for KNN Search on Road Networks. In CIKM, pages, 39-48, 2013.
[13] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases [J]. TODS, 24(2), 1999, 265-318.
[14] Antonm Guttman. R-TREE: A Dynamic Index Structure For Spatial Searching. ACM, New York, 1984.