Multiple Object Types KNN Search Using Network Voronoi Diagram

Geng Zhao[1], Kefeng Xuan[1], David Taniar[1], Maytham Safar[2], Marina Gavrilova[3], and Bala Srinivasan[1]

[1] Clayton School of Information Technology, Monash University, Australia
David.Taniar@infotech.monash.edu.au
[2] Department of Computer Engineering, Kuwait University, Kuwait
[3] Department of Computer Science, the University of Calgary, Canada

Abstract. Existing work on k nearest neighbor (kNN) in spatial/mobile query processing focuses on single object types. Furthermore, they do not consider optimum path in KNN. In this paper, we focus on multiple type kNN whereby the interest points are of multiple types. Additionally, we also consider an optimum path to reach the interest points. We propose three different query types involving multiple object types. Our algorithms adopt the network Voronoi Diagram (NVD). We describe two ways to solve multiple types of KNN queries: one is to create NVD for each object type, and two is to create one NVD for all objects. The comparison between these two approaches is presented in performance evaluation section.

1 Introduction

Mobile information systems affect our daily lives [2], and one of the most common mobile information services is k nearest neighbor (kNN) search [10,7,11,9,12]. Current approaches for kNN mainly use network expansion. Network expansion consumes a large amount of processing time because segments invoke functions iteratively. Consequently, Voronoi diagram is adopted as the most suitable tool to solve KNN queries because it aggregates lots of segments into polygons[9]. However, current approaches focus on one object type, which narrows down the mobile query scope. For example, find the nearest 3 hospitals from my current location. In some cases, users may want to get kNN of different object types (multiple object types), as well as to obtain the shortest routes. Motivated by these, this paper proposes new approaches on three different queries involving multiple object types using network Voronoi Diagram. In these queries, more than one object types are considered and the query result is highly related with the object types. Every object belongs to one of the category and there is no overlap between categories. That is the basic property of *multiple-object-type query.*

This paper focuses on three different types of KNN mobile queries, including: a) query to find nearest neighbor for multiple types of interest point (or 1NN for each object type), b) query to give the shortest path to cover multiple-object-types in a pre-defined sequence, and c) query to find an optimum path for multiple object types that gives the shortest path that covers the required interest objects in a random sequence.

2 Related Works

As this paper focuses on using Voronoi diagram to find k nearest neighbor, in this section, we firstly discuss an existing work called Voronoi-based network nearest neighbor (VN3) [7]. Also because our work is related to finding an optimum path, in this section, we will also introduce our previous work on the incremental k nearest neighbor (iKNN) [16] which is to find the shortest path through k interest points.

2.1 Voronoi-Based Network Nearest Neighbor (VN3)

Voronoi-based k nearest neighbor search (VN3) is proposed based on the properties of the Network Voronoi diagrams. It localized pre-computation of the network distances for a very small percentage of neighboring nodes in the network. It keeps the result in ascending order, adopts filter and refinement steps to generate and filter candidate result, and uses localized pre-computed network distances to save response time.

To be specific, firstly, the 1^{st} NN can be told directly by the intuition of Voronoi diagram. The polygon that contains query point will be the 1^{st} NN. Subsequently, a candidate set for other nearest neighbors of q is formed by 1^{st} NN's adjacent generators. Finally, the actual network distances from q to the generators in the candidate set can be pre-computed and this step will refine the set. The filter/refinement process in VN3 performs iteratively: at each step, firstly, generate a new set of candidates by the NVPs of the generators that are already selected as the nearest neighbors of q, then use the pre-computed distances to select the next nearest neighbor of q. Hence, the filter/refinement step must be invoked k times to find the first k nearest neighbors of q[7].

In summary, VN3 performs well if user just concern single types of interest points. If multiple object types should be retrieved or optimum path is the final result the user wanted, VN3 cannot be used directly. In the proposed approach section, we will show how we investigate the new query based on VN3.

2.2 Incremental k Nearest Neighbor (iKNN)

Incremental k nearest neighbor (iKNN) proposed by us [12] is given a set of candidate interest points to find a shortest path which starts at query point and goes through k interest points.

The approach of iKNN uses network expansion as Incremental Network expansion (*INE*) [10]. In the process of network expansion, iKNN records all expansion branches until one path is full of k interest points. The path is set as the boundary. Continue to do expansion, once there is a path shorter than the boundary; shrink it until all possible branches are expanded out of boundary. The result is the shortest path which passes k interest points.

iKNN can be used when all interest points are the same type and there is no difference between interest points. Our third kind of query is different from iKNN query because user wants to pass by k interest points and k interest points belong to k types. Also, iKNN uses network expansion which needs to expand the path segment by segment. That is the main drawback of iKNN performance. In our approach, we will use network Voronoi diagram instead of expansion which improves the performance of the algorithm.

3 Background

There are two important knowledge that needs to be discussed, namely Voronoi Diagram, and Network Voronoi Diagram, as they form the basis of our proposed approaches.

3.1 Voronoi Diagram

The Voronoi Diagram is a special kind of decomposition of a metric space determined by distances to a specified discrete set of objects in the space. In the mobile query processing, interest points are the polygon generators. Voronoi diagram is generated based on Euclidean distance, so it is not commonly used in mobile navigation. While some basic concepts are importance to migrate it in network Voronoi Diagram. Fig. 1 shows an example of Voronoi diagram based on Euclidean distance. P_i represents the interest points and the lines are the shared border edges between polygons.

The following points are basic properties associated with Voronoi diagram, which have been well presented by Okabe, Boots, Sugihara, and Nok Chiu [9].

Property 1. The Voronoi diagram of a point set P, $VD(P)$, is unique.

Property 2. The nearest generator point of p(e.g., pj) is among the generator points whose Voronoi polygons share similar Voronoi edges with VP (pi).

Property 3. Let n and n_e be the number of generator points and Voronoi edges, respectively, then $n_e \leq 3n - 6$.

Property 4. From property 3, and the fact that every Voronoi edge is shared by exactly two Voronoi polygons, we notice that the average number of Voronoi edges per Voronoi polygon is at most 6, i.e., $2(3n - 6)/n = 6 - 12/n \leq 6$. This means that on average, each generator has 6 adjacent generators.

3.2 Network Voronoi Diagram (NVD)

As mentioned above, Voronoi diagram is rarely used in mobile navigation because it is based on Euclidean distance. As we all know, in the real world, when the user wants to search nearest neighbor, the criterion of assessment is based on its network distance to the query point instead of Euclidean distance.

Network Voronoi diagram is the Voronoi diagram generated based on the network distance. In the typical Voronoi diagram, the shared border line is the mid perpendicular of the links connecting two corresponding generators. However, in the network Voronoi diagram, the border line is consisted of discrete points which are the middle points of network connections. A polygon contains a set of intersections and connections which are closer to its generator than any other generators. That is the difference between typical Voronoi diagram and network Voronoi diagram. In this paper, the network Voronoi diagram is fully investigated to solve target queries by using its basic properties listed above. Fig. 2 shows an example of network Voronoi diagram.

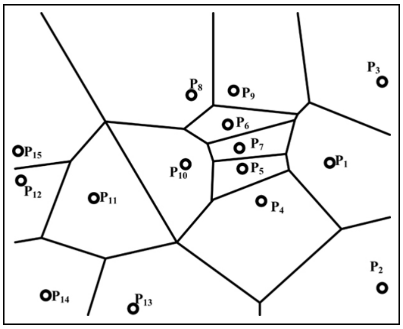


Fig. 1. Voronoi Diagram

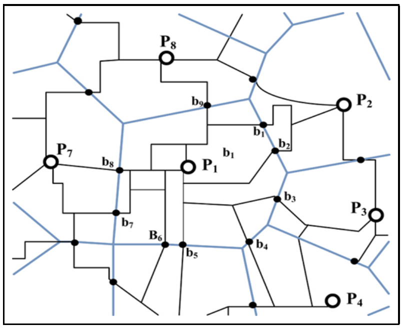


Fig. 2. Network Voronoi Diagram

4 A Taxonomy of Multiple Object Types *k*NN Queries

This paper aims at proposing approaches for KNN queries for multiple object types. We propose three types of queries involving multiple object types KNN:

1. *Multiple-object-types Nearest Neighbor* (M_NN) query is to find nearest neighbors for multiple object types. It is common in mobile navigation. Around the query point, there are k different types of interest points. For each object type, to find the nearest neighbor among the same object type is the objective of the query.

   Example 1. Suppose a group of colleagues wants to have dinner together, and around their company there are hundreds of restaurants. They prefer French, Italian and Chinese food. As a result, they want to know the nearest French, Italian and Chinese restaurant respectively first and then make the final decision.

2. *Incremental Multiple-object-types Nearest Neighbors* (iM_NN) query is to find optimum/shortest path to pass multiple object types in the pre-defined sequence. This query can be used when the sequence of passed interest points is critical for the user.

   Example 2. Suppose a person falls ill at home suddenly, the family wants to tell their driver the following path. Firstly, obviously they want to go to the nearest hospital because the sick is acute. Secondly, they need to go to the nearest medical checkup clinic. After that, they will go to find the nearest GP office to check the checkup result and finally go to the nearest pharmacy according GP's prescription.

3. *Optimum Path Multiple-object-type Nearest Neighbors* (PM_NN) query is to find optimum/shortest path to pass multiple object types in random sequence. Although it seems similar with 2^{nd} query, it is a novel issue actually because the interest points can be random passed.

   Example 3. Suppose a secretary has plan to do the following things: go to post office to post a letter, go to bank to deposit a cheque, go to shop to buy some print paper and go to dry cleaner to deliver a piece of clothes. So she wants to get the best routine which not only covers all places but also makes her travelling path shortest.

In summary, they are novel queries as there is no approach touching the query about kNN of multiple object types and they are reality-oriented and practical.

5 Proposed Approaches

In this section, we present our proposed algorithms for the three kinds of multiple-object-type KNN queries. The first two proposed query processing (M_NN and iM_NN) use two approaches, namely: using one NVD for each object type, and using one NVD for all object types, whereas the last proposed query processing for PM_NN uses one NVD for all object types model.

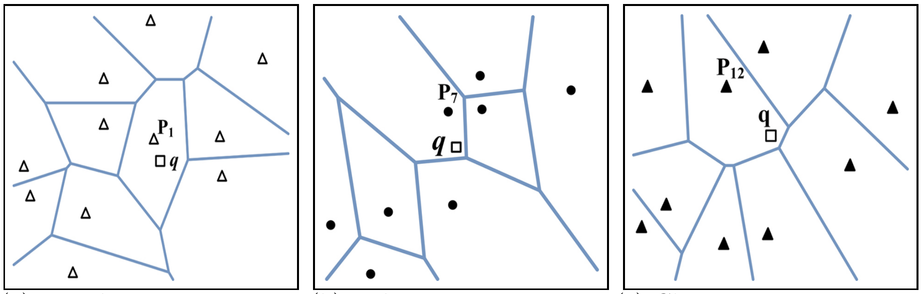
5.1 Multiple-Object-Types Nearest Neighbor(M_NN)

In this section, we propose two ways to solve M_NN query: (*i*) For each object type, generate a NVD and find the nearest neighbor. (*ii*) Generate one NVD for all objects then filter them while searching the target result.

One NVD for Each Object Type: A straight approach is firstly generating NVD for each object type. For each type, find its nearest neighbor for query point using its NVD. The result comes out directly when all nearest neighbor of each type have been found. There is no reason to doubt its correctness. But concerning its efficiency, it becomes infeasible because if there are too many different kinds of objects, loading different NVDs will consume most of the processing time.

Consequently in this section, an alternative way is proposed for the query: one NVD for all objects.

Based on example 1 in section 4, Fig. 3(a), 3(b) and 3(c) represent NVDs of French, Italian and Chinese restaurants respectively. As a result, the nearest French (P_1), Italian (P_7) and Chinese (P_{12}) restaurant can be told directly.



(a) French restaurant NVD (b) Italian restaurant NVD (c) Chinese restaurant NVD

Fig. 3. Example 1 - One NVD for each object type

One NVD for All Object Types: Generate just one NVD for all objects which not only includes the types that the user concerns but also includes the objects of other types. It will definitely improve the performance both in time and storage aspects. Algorithm performs as follows.

Firstly, generate NVD considering all objects as polygon generators. Then "contain" function is invoked to get the generator whose polygon covers query point. This generator is the first nearest neighbor of its type.

Secondly, do expansion within this polygon and record the distance from query point to all border points. Calculate the distance from query point to all adjacent polygon generators. As all border points to generators' distance are pre-computed, this process can be finished transitorily.

Thirdly, the generators will be put in a queue sorting by their distance to query point. From the shortest one, if by now query result for its type have not found, it will be recorded as query result for this type; otherwise, just discard it. Then add its adjacent generators into the list and sort again. Do this step iteratively until all object types' nearest neighbors have been found.

Finally, we get a result list which is for each object type there is an interest point nearest to query point among others in this type.

The algorithm can be expressed in Algorithm 1..

Following example fully illustrates how the algorithm works. The scenario is based on example 1 in section 4 as well. In this case, query should retrieve 3 restaurants because user just concerns 3 types of restaurants, French, Italian and Chinese. The processing steps are as follows:

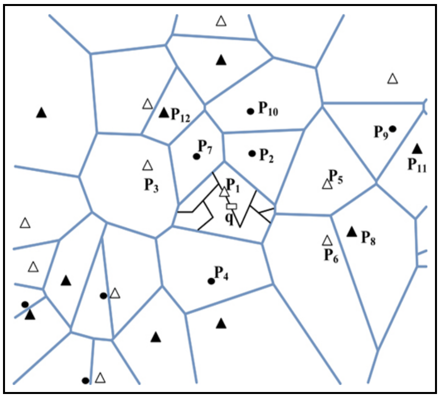


Fig. 4. Example 1 - One NVD for all objects

- Generate NVD as in Fig. 4. White triangle, black dot and black triangle indicate French, Italian and Chinese restaurants respectively. Use *contain*() function to locate P_1 which is the 1st NN of q.
- As Type(P_1) = French, initial RL = {(French, P_1), (Italian, ∅), (Chinese, ∅)} Initial NP = {P_2, P_3, P_4, P_5, P_6, P_7} by adding all P_1's adjacent into NP.
- Expand q within P_1's polygon and record all distance from q to border points. Calculate the distance from q to each P in NP and sort them in ascending order. Update NP, suppose NP = {(P_5, 5), (P_7, 7), (P_2, 9), (P_3, 11), (P_6, 16), (P_4, 18)}
- Pop out P_5. As type(P_5) = French & in RL, French already has value P_1, ignore P_5. Add P_5's adjacent neighbors into NP and update NP. Suppose the distance is: NP = {(P_7, 7), (P_2, 9), (P_3, 11), (P_8, 14), (P_6, 16), (P_4, 18), (P_9, 19), (P_{10}, 22), (P_{11}, 28)}
- Then Pop out P_7. As type(P_7) =Italian & in RL, Italian has null value, update RL as RL = {(French, P_1), (Italian, P_7), (Chinese, ∅)}. After that, add all P_7's adjacent neighbors into NP and update NP. Suppose the distance is: NP = {(P_2, 9), (P_{12}, 10), (P_3, 11), (P_8, 14), (P_6, 16), (P_4, 18), (P_9, 19), (P_{10}, 22),(P_{11}, 28)}
- Then pop out P_2 and ignore it as it is Italian restaurant. Then Pop out P_{12}. As type(P_{12}) = Chinese & in RL, Chinese has null value, update RL as RL = {(French, P_1), (Italian, P_7), (Chinese, P_{12})}. Algorithm terminates.

5.2 Incremental Multiple-Object-Types Nearest Neighbors (iM_NN)

The query of incremental nearest neighbors for sequential multiple object types is to find the shortest path which goes through multiple object types in predefined sequence. In this case, the sequence is crucial to the user and the user wants to pass these object types in certain order as example 2 in section 4.

Algorithm 1. M_NN(k, query point)

```
1: Generate Voronoi diagram using all interest points
2: $RL$ (Result List) = $\{(type_1, \emptyset), (type_2, \emptyset), ..., (type_k, \emptyset)\}$
3: $P_i$ = $1^{st}$NN = contain ($q$)
4: $type_i$ = Check_type ($P_i$)
5: $RL$ (Result List) = $\{(type_1, \emptyset), (type_2, \emptyset), ..., (type_i, P_i), ..., (type_k, \emptyset)\}$
6: Initial NP (Neighbor point) = $\{P_i$'s adjacent generator$\}$
7: Expand $q$ within this polygon & record distance from $q$ to border point.
8: Calculate distance from $q$ to each $P$ in $NP$ & sort them in ascending distance order. $NP$ = $\{(P_1$, dist($q$, $P_1$)),..., $P_i$, dist($q$, $P_i$)$\}$
9: Pop out the first $P$ in $NP$, suppose it is $P_j$
10: $type_j$ = Check_type ($P_j$)
11: if in $RL$, $type_j$ has null values then
12:     update $RL$ as ($type_j$, $P_j$)
13: else
14:     ignore $P_j$
15: end if
16: if all type has values in $RL$ then
17:     terminate algorithm
18: else
19:     Add $P_j$'s adjacent neighbor into $NP$ & go to step 8
20: end if
```

From the example, we can tell that the sequence of object types is crucial, in other words, the routine should begin at home then pass hospital, medical checkup clinic, GP office and end at one pharmacy. It is not hard to see that the approach performs as following steps: when the path reaches the interest point, it is treated as the new query point. Then continue to search the nearest neighbor of the next type until all object types have been found. There are two ways which can solve this query, named as one NVD for each object type and one NVD for all objects.

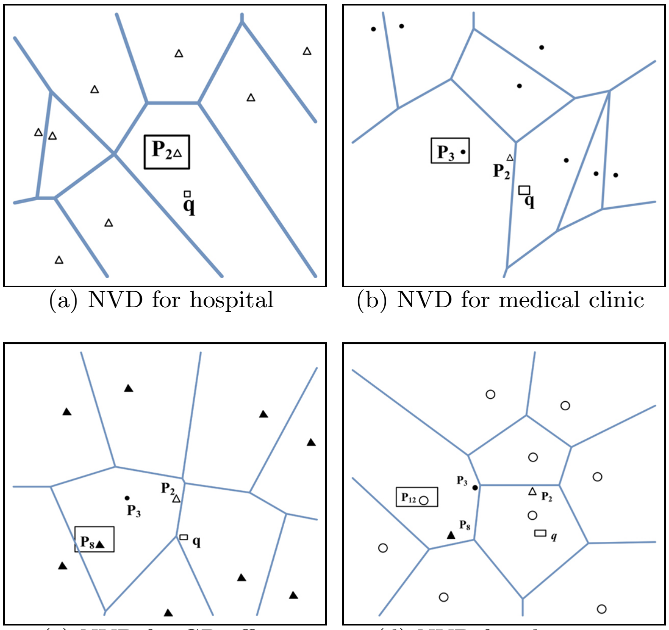
One NVD for Each Object Type: This method firstly generates NVD for the 1^{st} object type and finds the nearest one of this type. Then it generates NVD for the 2^{nd} object type and finds the nearest one of this type considering 1^{st} NN as the query point. Continue to do so for the following object types until nearest neighbors for all types have been found.

Fig. 5 shows the processing steps based on example 2 in section 4. The result is automatically shown in the figures: Shortest path starts at q, firstly goes to hospital P_2, then heads to checkup clinic P_3, after that, towards to GP office P_8, finally arrives pharmacy P_{12} for medicine.

One NVD for each object type is actually dividing this query into multiple 1_NN queries. There is no reason to doubt its correctness. But concerning its efficiency, it becomes infeasible because if there are too many different kinds of interest points, loading different NVDs will consume most of the processing time.

One NVD for All Object Types: This method generates just one NVD for all object types, including not only the type user concerns but also other object types. It saves time and storage. The following steps illustrate how it works.

Firstly, one NVD is generated considering all objects as polygon generators. Then invoke "contain" function to get the first nearest neighbor. Check whether



(a) NVD for hospital (b) NVD for medical clinic
(c) NVD for GP office (d) NVD for pharmacy

Fig. 5. Example 2 - One NVD for each object type

it is the 1st type user wants to. If yes, go to the 2nd step; otherwise check the adjacent neighbors of this interest point until find the nearest neighbor of 1st type.

Secondly, consider the 1st NN as query point; find its nearest neighbor of 2nd type using the pre-computed distance to its adjacent neighbors.

Thirdly, do these operations iteratively until all object types have been found.

Finally, a shortest path comes out which begins at query point, pass multiple object types in user defined sequence until reaches the last object. That is the optimum path of this kind of query.

The algorithm can be expressed in Algorithm 2..

A case study based on example 2 in section 4 fully illustrates the approach. In this case, the user concerns 4 object types ($k = 4$) because they want to pass hospital, checkup clinic, GP office and pharmacy one by one. The processing steps are as follows:

- Generate NVD as Fig. 6. White triangle, black dot, black triangle and white dot indicate hospital, checkup clinic, GP office and pharmacy respectively.
- Initial RL={(hospital, $\emptyset$), (checkup clinic, $\emptyset$), (GP office, $\emptyset$), (pharmacy, $\emptyset$)} & $M = 1$
- Use *contain*() function to locate P_1 which is the 1stNN of q.
- Expand q within P_1's polygon and record all distances from q to borders.

Algorithm 2. iM_kNN(k, query point)

```
1: Generate Voronoi diagram using all interest points within given types
2: $RL$ (Result List) = $(type_1, \emptyset), (type_2, \emptyset), \ldots, (type_k, \emptyset)$
3: Initial $M$ =1
4: $P_i$ = $1^{st}$NN = contain($q$)
5: $type_i$ = Check_type ($P_i$)
6: if $type_i = type_m$ then
7:     update $type_m$'s values as $P_i$&$M$=$M$+1
8: else
9:     Expand $q$ within this polygon&record distance from $q$ to border
10:    Initial $NP$ (Neighbor point) = $P_i$'s adjacent generator
11:    Calculate distance from $q$ to each $P$ in $NP$ & sort them in ascending order.
       NP = {($P_1$, dist($q$, ($P_1$))),...,($P_1$, dist($q$, ($P_1$))) }
12:    Pop out the first $P$ in $NP$, suppose it is $P_j$. $type_j$ = Check_type ($P_j$)
13:    if $type_j = type_m$ then
14:        update $type_m$'s values as $P_j$&$M$=$M$+1
15:    else
16:        update $NP$ by adding $P_j$'s adjacent neighbor into $NP$&go to step 11
17:    end if
18: end if
19: while $M \leq k$ do
20:    Suppose $P_{m-1}$= $type_{m-1}$'s values in $RL$
21:    Initial $NP$(Neighbor point)=$P_{m-1}$'s adjacent generator
22:    Calculate distance from $P_{m-1}$ to each $P$ in $NP$ and sort them in ascending order. NP =
       {($P_1$, dist($q$, $P_1$)),..., ($P_i$, dist($q$, $P_i$))}
23:    Pop out the first $P$ in $NP$, suppose it is $P_n$. $Type_n$ = Check_type ($P_n$)
24:    if $Type_n = type_m$ then
25:        update $type_m$'s values as $P_n$&$M$=$M$+1
26:    else
27:        update $NP$ by adding $P_n$'s neighbor into $NP$&go to step 22
28:    end if
29: end while
30: return $NP$
```

– As type(P_1) = pharmacy $\neq type_m$, Initial NP = {P_2, P_3, P_4, P_5, P_6, P_7, P_8} by adding all P_1's adjacent into NP.
– Calculate the distance from q to each P in NP and sort them in ascending order. Update NP as NP={(P_2,2),(P_3,4),(P_4,6),(P_8,8), (P_7,9),(P_5,12),(P_6,16)}
– Pop out P_2. As Type (P_2) = hospital = $type_m$, update RL as RL = (hospital, P_2), (checkup clinic, $\emptyset$), (GP office, $\emptyset$), (pharmacy, $\emptyset$). M = 2

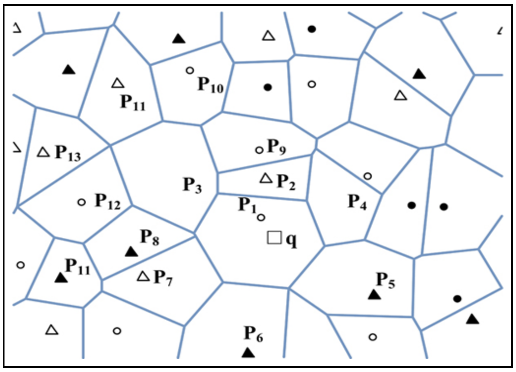


Fig. 6. Example 2 - One NVD for all objects

– As $M < k$, $type_{m-1}$'s value is P_2. Initial $NP = \{P_1, P_3, P_4, P_9\}$ by adding all P_2's adjacent into NP.
– Calculate the distance from P_2 to each P in NP and sort them in ascending distance to P_2. Update NP, suppose NP=$\{(P_1,3),(P_3,6),(P_9, 8),(P_4,9)\}$
– Then pop out P_1. As type(P_1)=pharmacy $\neq$ $type_m$=checkup clinic, ignore P_1. Add P_1's adjacent neighbors into NP and update NP. Suppose the distance is NP=$\{(P_3,6),(P_9,8),(P_4,9),(P_8,13),(P_7,15),(P_5,18),(P_6,21)\}$
– Pop out P_3. As type(P_3) = checkup clinic = $type_m$, update RL as RL = {(hospital, P_2), (checkup clinic, P_3), (GP office, $\emptyset$), (pharmacy, $\emptyset$)}. $M = 3$
– As $M < k$, $type_{m-1}$'s value is P_3. Initial NP = {(P_8, 12), (P_1, 14), (P_2, 17), (P_9, 23), (P_{12}, 25), (P_{13}, 27), (P_{11}, 30), (P_{10}, 31)}.
– Then pop out P_8. As type(P_8) = GP office = $type_m$, update RL as RL = {(hospital, P_2), (checkup clinic, P_3), (GP office, P_8), (pharmacy, $\emptyset$)}. $M = 4$
– As $M = k$, $type_{m-1}$'s value is P_8. Initial NP = {(P_{12}, 15), (P_1, 16), (P_7, 17), (P_{14}, 26)}.
– Then pop out P_{12}. As type(P_{12})=pharmacy=$type_m$, update RL as RL = {(hospital,P_2), (checkup clinic,P_3), (GP office,P_8), (pharmacy,P_{12})}. $M = 5$
– As $M > k$, algorithm terminates.

Results are: The optimum path firstly goes to hospital P_2, then heads to checkup clinic P_3, GP office P_8 and finally arrives at pharmacy P_{12} for medicine.

5.3 Optimum Path Multiple-Object-Type Nearest Neighbors (PM_NN)

Optimum path for multiple object types' query is similar with the 2_{nd} query except that object types can be passed in any sequence. In this query, the length of whole path is the criterion of assessment. As multiple 1_NN cannot guarantee the final path is the shortest one, this approach is different with IM_NN approach in the last section. More details can be told based on example 3 in section 4.

In example 3, the sequence of interest points is unimportant because posting letter, depositing cheque and so on are independent tasks and it does not matter which task the user does first. In addition, the objective of this query is to make the whole path short not to find any nearest object. There may be an instance that after choosing the nearest post office, the path to other place will become farther. Maybe choosing the second or even third nearest post office is better. In addition, how to arrange the sequence of interest points is another issue needed to be solved. The following steps illustrate the process of the approach.

Firstly, generate NVD considering all objects as polygon generators. Then invoke "contain" function to get the nearest generator P. Check its type and record P as the first object type that the user will visit. For all P's adjacent neighbors, sort them in the ascending sequence of their distance to P. Check their types one by one, if the path has not visited that object type, record it as the next P. From now on, start from this P, do the same operation as the first P until all types have been found and the path is completed. Above operation cannot guarantee this path shortest but it did set a boundary for the query (d_{max}) which means once expansion is over this boundary, it should be terminated.

Secondly, every object whose distance to q is smaller than d_{max} can be treated as potential first interest point. Sort them in a queue by their distance to q.

Thirdly, for each interest point in the queue, pop it out, find its closest neighbor whose type has not been covered and then from that neighbor do the same things until all types of interest points have been covered. If in the process of the expansion, the distance is over the boundary, terminate it directly. If the path is completed, compare its path length with the boundary and update the boundary if it is smaller.

Terminate the algorithm when no interest point in the queue. The optimum path shows how the user can pass multiple object types in random sequence.

The algorithm can be express in Algorithm 3.

Algorithm 3. PM_NN(k, query point)

```
 1: Generate Voronoi diagram using all interest points within given types
 2: $d_{max}=\infty$
 3: Initial $TS=\{type_1,type_2,...,type_k\}$ $R=\{dist_q, \emptyset_1,\emptyset_2,..., \emptyset_k\}$
 4: $RL = \emptyset$ $S = \emptyset$
 5: $P_1=1^{st}$NN=contain($q$)
 6: $t_{p1}$=Check_type($P_1$)
 7: Suppose $type_i = t_{p1}$, remove it from $TS$
 8: $R=\{dist_q, P_1, \emptyset_2,..., \emptyset_k\}$
 9: Initial $NP$(Neighbor point)=$P_1$'s adjacent generator
10: Calculate distance from $P_1$ to each $P$ in $NP$ & sort them in ascending order.
    NP = $\{(P_1$, dist($q$, ($P_1$))),...,($P_i$, dist($q$, ($P_i$)))$\}$
11: Pop out the first $P$ in $NP$, suppose it is $P_j$
12: $t_{pj}$=Check_type($P_j$)
13: if If $t_{pj}$ is in $TS$ then
14:     update $t_{pj}$ in $R$ & remove $t_{pj}$ from $TS$
15:     if $TS$ is not $\emptyset$ then
16:         add $P_j$'s neighbor into $NP$ & go to step 11
17:     else
18:         if $dist_q < d_{max}$ then
19:             update $d_{max} = dist_q$ & $RL = R$
20:         else
21:             ignore it
22:         end if
23:     end if
24: else
25:     add $P_j$'s adjacent neighbor into $NP$ & go to step 11
26: end if
27: Expand $q$ within this polygon & record distance from $q$ to border point
28: Update $S$=$\{$all objects *(dist) to q ¡* $d_{max}$ *sort in ascending distance order*$\}$
29: for each $P$ in $S$ do
30:     Pop out the first $P$ & Initial $TS=\{type_1,type_2,...,type_k\}$
31:     $t$=Check_type($P$)
32:     $R=\{dist_q,P,\emptyset_2,...,\emptyset_k\}$
33:     Initial $NP$(Neighbor point)=$P$'s adjacent generator
34:     Calculate distance from $P$ to each $P_i$ in $NP$ & Wipe out the $P$ whose $dist(q, P) > d_{max}$
        NP=$\{(P_1$,dist($q$,($P_1$))),...,($P_i$,dist($q$,($P_i$)))$\}$
35:     if $NP \neq \emptyset$ then
36:         Pop out the first $P$ in $NP$, suppose it is $P_j$
37:     else
38:         go to step 28
39:     end if
40:     $t_{pj}$=Check_type($P_j$)
41:     if $t_{pj}$ is in $TS$ then
42:         update $t_{pj}$ in $R$ & remove $t_{pj}$ from $TS$
43:         if $TS$ is not $\emptyset$ then
44:             add $P_j$'s neighbor into $NP$ & go to step 28
45:         else
46:             if $dist_q < d_{max}$ then
47:                 update $d_{max} = dist_q$ & $RL = R$ & wipe off $P$ in $S$ $dist(P) > d_{max}$
48:             end if
49:             go to step 34
50:         end if
51:     else
52:         add $P_j$'s adjacent neighbor into $NP$ & go to step 34
53:     end if
54: end for
```

To clarify the algorithm, a case study will fully illustrate how it works.

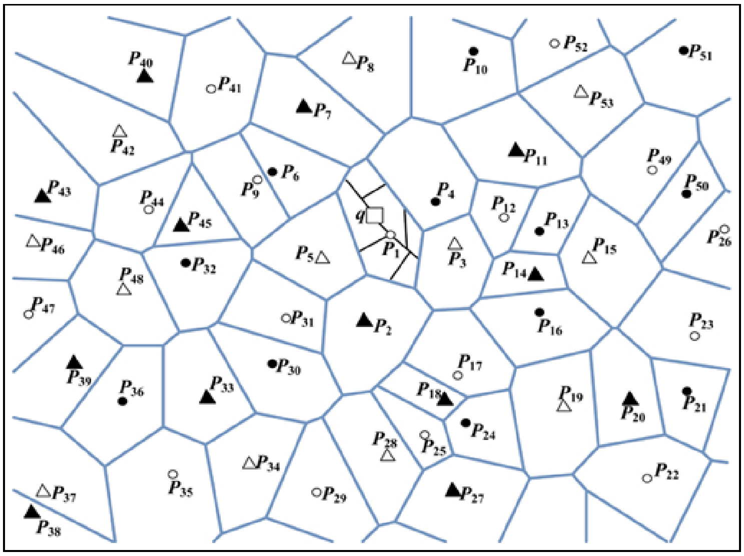


Fig. 7. Example 3 - One NVD for all objects

- Generate NVD as in Fig. 7. White triangle, black dot, black triangle and white dot indicate post office, bank, shop and dry cleaner respectively.
- Initial $d_{max} = \infty$, TS = {post office, bank, shop, dry cleaner}, R= {$dist_q$, $\emptyset_1$, $\emptyset_2$,..., $\emptyset_k$}, $RL = \emptyset$
- Use $contain()$ function to locate P_1 which is the 1_{st}NN of q.
- As Type(P_1)=dry cleaner, update TS={post office, bank, shop} by removing it from TS & $R = \{1, P_1, \emptyset_2, ..., \emptyset_k\}$// suppose P_1 to P_q's distance is 1
- From P_1, find nearest neighbor whose type in TS. Suppose P_4. As Type(P_3) = bank, update TS = {post office, shop} & $R = \{5, P_1, P_4, ..., \emptyset_k\}$// suppose P_4 to P_1's distance is 4
- Do the same to P_4 as the step above until $TS = \emptyset$. Suppose R = {15, P_1, P_4, P_{11}, P_{53}} Update d_{max} =15
- Expand q within P_1's polygon and record all distances from q to borders.
- Initial $S = \{P_4, P_3, P_6, P_5, P_7, P_2...P_n\}$ // whose distance to q within d_{max}
- Pop out P_4&update TS={post office,dry cleaner,shop} & R= {3,P_4,$\emptyset_2$,...,$\emptyset_k$}. Search P_4's nearest neighbor whose type in TS and do the same for the rest interest points iteratively until $TS = \emptyset$. Suppose $R = \{12, P_4, P_3, P_{12}, P_{11}\}$, then update d_{max}=12. Wipe out P in S $dist_q > 12$.
- Do the same operation to every P in S until S is empty. In the process, once the distance is over d_{max}, terminate expansion for this path.

The result comes out finally $R = \{10, P_3, P_{12}, P_{13}, P_{14}\}$. The user firstly goes to post office P_3, then heads to dry cleaner P_{12}, after that, towards bank P_{13} and finally arrives at shop P_{14} and the length of the final path is 10.

6 Performance Evaluation

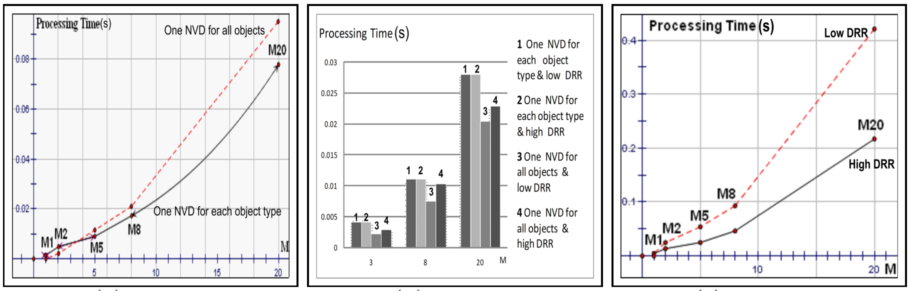
In the experimentations, Melbourne city map and Frankston map in Australia are chosen from the whereis website [14]. In these maps, shops and restaurants represent high-density scenario of interest points, in the other hand, hospitals and shopping centers represent low-density scenario of interest points. All interest points are real-world data. The performance of our approaches is analyzed in runtime aspect in different diversity of interest point or in different interest points' density.

For the nearest neighbor for multiple object types, the processing time is increasing with the number of object types (M). In Fig. 8(a), the dash line indicates the performance of one NVD for each object type approach and the solid line indicates the performance of one NVD for all objects approach. In this case, we use different types of shops as candidate types and the average density is $5/km^2$. From Fig. 8(a), we can easily tell that one NVD for each object type performs better than one NVD for all objects if objects types are small, especially smaller than 4. Otherwise, one NVD for all objects is a better choice because it saves time for generating NVD. We can also tell that with the increasing object types, the processing time increases sharply because more polygon expansions will be invoked and more NVDs should be generated.

For incremental nearest neighbors for sequential multiple object types query, the processing time is increasing with the number of object types (M). Here a definition is introduced: density relative rate (DRR). DRR is ratio of the highest density to lowest density of all object types. As a result, DRR is not smaller than 1. The closer to 1 DRR is, the more evenly objects distribute. For example, if the user concerns 4 object types and their densities are $5.5/km^2$, $3.6/km^2$, $2.5/km^2$ and $1.1/km^2$ respectively. So this scenario's DRR is $5.5/km^2$ (highest) $1.1/km^2$(lowest)=5. In Fig. 8(b), the first two bars indicate the processing time of one NVD for each object type approach and the last two bars indicate the processing time of one NVD for all objects approach. The first and third bars are operating in low DRR scenario (DRR=1) and the second and forth bars are in high DRR scenario (DRR=10).

Fig. 8(b)illustrates that processing time will increase if DRR increases. In addition, the higher DRR is, the closer two approaches (one NVD for each & one NVD for all) performs. In addition, generally, one NVD for all interest points performs better than one NVD for each object types because generating and loading NVD are time consuming tasks.

The processing time for optimum path for multiple object types query, the processing time is increasing with the object types (M). In Fig. 8(c), the dash line indicates the performance of the query when density relative rate



(a) M_NN (b) iM_NN (c) PM_NN

Fig. 8. Processing Time Comparison

(DRR) = 1 and the solid line indicates the performance of the query when density relative rate (DRR) = 5.

From Fig. 8(c), with the increasing object types, the processing time increases sharply because more polygon expansions will be invoked. Moreover, DRR is another critical factor for the performance of the approach. The processing time increases more sharply if DRR increases from 1 to 5.

7 Conclusion and Future Work

This paper inspires novel KNN search involving multiple object types. The first query (nearest neighbor for multiple object types) provides a solution if the user wants to get 1_NN for each category of interest points. The second query (incremental nearest neighbors for sequential multiple object types) helps user to find the shortest path to pass through multiple object types in pre-defined sequence. The last query (optimum path for multiple object types) provides an optimum path for users if they want to pass multiple object types without any sequential constrain. These approaches investigate novel KNN in multiple object types using network Voronoi Diagram which enriches the content of our mobile navigation system and gives more benefits to mobile users.

In the future, we are going to incorporate intelligence techniques and context-aware in mobile navigation and mobile query processing [5,4,6,1,3,8]. In addition, range and kNN search combined with dynamic query point will also be investigated [15]. Performance in mobile query processing is always an issue, and we plan to examine more thoroughly the performance issues of mobile query processing including the use of data broadcast techniques also deserves further investigation [13].

Acknowledgments. This research has been partially funded by the Australian Research Council (ARC) Discovery Project (Project No: DP0987687).

References

1. Aleksy, M., Butter, T., Schader, M.: Architecture for the development of context-sensitive mobile applications. Mobile Information Systems 4(2), 105–117 (2008)
2. Bohl, O., Manouchehri, S., Winand, U.: Mobile information systems for the private everyday life. Mobile Information Systems 3(3,4), 135–152 (2007)
3. Doci, A., Xhafa, F.: A wireless integrated traffic model. mobile information systems. Mobile Information Systems 4(3), 219–235 (2008)
4. Goh, J.Y., Taniar, D.: Mobile data mining by location dependencies. In: Yang, Z.R., Yin, H., Everson, R.M. (eds.) IDEAL 2004. LNCS, vol. 3177, pp. 225–231. Springer, Heidelberg (2004)
5. Goh, J., Taniar, D.: Mining frequency pattern from mobile users. In: Negoita, M.G., Howlett, R.J., Jain, L.C. (eds.) KES 2004. LNCS, vol. 3215, pp. 795–801. Springer, Heidelberg (2004)
6. Gulliver, S.R., Ghinea, G., Patel, M., Serif, T.: A context-aware tour guide: User implications. Mobile Information Systems 3(2), 71–88 (2007)
7. Kolahdouzan, M.R., Shahabi, C.: Voronoi-based k nearest neighbor search for spatial network databases. In: Proc. of 30th VLDB, Toronto, Canada, pp. 840–851. Morgan Kaufmann Publishers Inc., San Francisco (2004)
8. Luo, Y., Xiong, G., Wang, X., Xu, Z.: Spatial data channel in a mobile navigation system. In: Gervasi, O., Gavrilova, M.L., Kumar, V., Laganá, A., Lee, H.P., Mun, Y., Taniar, D., Tan, C.J.K. (eds.) ICCSA 2005. LNCS, vol. 3481, pp. 822–831. Springer, Heidelberg (2005)
9. Okabe, A., Boots, B., Sugihara, K., Chiu, S.N.: Patial Tessellations: Concepts and Applications of Voronoi Diagrams, 2nd edn. John Wiley and Sons Ltd., Chichester (2000)
10. Papadias, D., Zhang, J., Mamoulis, N., Tao, Y.: Query processing in spatial network databases. In: Proc. of 29th VLDB, Berlin, Germany, pp. 802–813. Morgan Kaufmann Publishers Inc., San Francisco (2003)
11. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: Proc. of ACM SIGMOD, San Jose, California, pp. 71–79. ACM Press, New York (1995)
12. Safar, M.: K nearest neighbor search in navigation systems. Mobile Information Systems 1(3), 1–18 (2005)
13. Waluyo, A.B., Srinivasan, B., Taniar, D.: Optimal broadcast channel for data dissemination in mobile database environment. In: Zhou, X., Xu, M., Jähnichen, S., Cao, J. (eds.) APPT 2003. LNCS, vol. 2834, pp. 655–664. Springer, Heidelberg (2003)
14. Telstra Corporation whereis Melbourne (February 2006), `http://www.whereis.com`
15. Xuan, K., Zhao, G., Taniar, D., Srinivasan, B.: Continuous range search query processing in mobile navigation. In: Proceedings of the 14th ICPADS 2008, Melbourne, Victoria, Australia, pp. 361–368 (2008)
16. Zhao, G., Xuan, K., Taniar, D., Srinivasan, B.: Incremental k-nearest-neighbor search on road networks. JOIN 9(4), 455–470 (2008)