The Knowledge Engineering Review, Vol. 29:2, 217–233. © Cambridge University Press, 2014
doi:10.1017/S0269888914000058

Time constraint route search over multi-locations

GENG ZHAO[1], KEFENG XUAN[1], DAVID TANIAR[1], MAYTHAM SAFAR[2] and BALA SRINIVASAN[1]

[1]*Clayton School of Information Technology, Monash University, Australia;*
e-mail: David.Taniar@infotech.monash.edu.au;
[2]*Department of Computer Engineering, Kuwait University, Kuwait*

Abstract

Traditional Route Search aims at finding the path that goes through geographical entities that are relevant to the provided search terms from the start point to the end point. Without constraints, traditional Route Search visiting multiple locations is unreliable because locations may close after a specified time. In this paper, time constraint (operating hours of each location) is drawn into Route Search query in order to make the query more realistic. Two methods are proposed in this paper, namely Route Search for fixed locations (*RFix*) and Route Search for flexible locations (*RFlex*). These two queries are different from the existing Route Search query because (1) the end point is not pre-defined and (2) time constraint is involved. Our two proposal queries consider whether the locations are specifically pre-defined by the user or only the location types are specified. In each method, two propositions are presented for pruning expansion branches, which highly improves the performance. Our experiments verified the applicability of *RFix* and *RFlex* to solve Route Search queries with time constraint queries.

1 Introduction

As mobile GPS is becoming widely available in the market, coupled with publicly accessible online maps (e.g. Google Maps), route planning and direction that show a desirable route from a departure point to the destination has grown to be one of the most popular navigation applications these days (Safar, 2005; Xuan *et al*., 2008, 2011). The recommended routes by these systems are normally those with the shortest distance. Using these systems, users could easily adjust the turning points at any intersections, which alter the initial route to become a user-preferred route. More advanced navigation systems can even predict the traveling time of the chosen route. Route planning can be regarded as a successful application of shortest distance algorithms, like Dijkstra algorithm (Dijkstra, 1959).

Recently, route search has been extended to include locations to be visited along the planned route (Huang & Jensen, 2004; Terrovitis *et al*., 2005; Yoo & Shekhar, 2005; Zhang, 2008; Zhao *et al*., 2013; Kanza *et al*., 2008b; Zhao *et al*. 2011; Kanza *et al*., 2008a, 2009). The aim was to find the shortest distance, and sometimes the most reliable route, that covers all user-defined locations or places. Although this is certainly useful, it is often impractical, due to a couple of reasons: (i) each location or place, which are normally a spatial business entity (e.g. bank, dry cleaner, supermarket) has the opening hours—this implies that when this place is visited, it must be during their business hours; and (ii) the traveling time from one location to another needs to be considered, as in many cases, traveling time is more useful than the distance alone. Hence, in order to make route planning over visiting locations, one must take into account these two constraints.

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

In this paper, we refer to these constraints as Time Constraints. Therefore, our paper focuses on route search over multiple locations taking into consideration time constraints.

It is therefore imperative to assume that the route or path that arrives on the location outside the operation hour is considered as an invalid path. This problem exists in daily life, whereby we sometime have to choose a longer path to go back and forth places just to meet the business hours of one location before its closing time. Hence, we need to draw time constraints into our proposed methods.

Route search over multiple locations is often assumed to be the problem of kNN or continuous kNN in spatial and mobile databases (Roussopoulos *et al.*, 1995; Papadias *et al.*, 2003; Kolahdouzan & Shahabi, 2004; Safar, 2005). There is a huge distinction between kNN and route search. kNN finds spatial objects that closest located to the query point, without considering the path that needs to be established as the user has to visit each objects in the query result. Because of this, existing work on kNN is inapplicable to solve route search problems. Our previous work on incremental kNN (called *ik*NN; Zhao *et al.*, 2013) attempts to solve route search problem whereby it could find the shortest distance to cover k number of homogenous type of locations—however, it does not consider time constraints, nor heterogenous multiple types of locations.

In this paper, we focus on two problems of route search over multiple heterogenous locations: one for fixed locations, and the other for flexible locations. Fixed locations refer to predetermined locations by the user, such as Citibank on a specific location, Pharmore pharmacy on a specific location, etc. In this case, not only specific business entity is specified, such as Citibank and not any bank, or Pharmore pharmacy and not any pharmacy, but also the specific location, such as Citibank on 180 High Street, or Pharmore pharmacy on 25 Cure Road, etc. Hence, a Route Search over Fixed Locations (our proposed algorithm is then called *RFix*) finds the most efficient route to visit the user-defined fixed locations in a non-predefined order.

Flexible locations, on other hand, refer to predetermined location types, are not the exact location itself. For example, if user wants to visit a pharmacy, which can be the pharmacy anywhere; or to visit Citibank, but can be in any branch. So, a route search over flexible locations, for example, is to find the most efficient route to visit Citibank, a pharmacy, etc. in a non-predefined order. Our proposed algorithm for Route Search over Flexible Locations is abbreviated as *RFlex*. Both *RFix* and *RFlex* use the travel time network to estimate the travel time between any two locations, as well as using the time constraints imposed by not only the operating hours of each location, but also the traveling time itself.

2 Related work

Some variations of route search have been investigated in earlier works (Pearson & Guesgen, 1998; Huang & Jensen, 2004; Ehliar & Liu, 2005; Terrovitis *et al.*, 2005; Yoo & Shekhar, 2005; Zhang, 2008; Zhao *et al.*, 2013; Kanza *et al.*, 2008a; Kanza *et al.*, 2008b). In Yoo and Shekhar (2005) and Huang and Jensen (2004), they try to find the route with smallest deviation to visit a new point when user travels a pre-defined route. Zhao *et al.* (2013) and Terrovitis *et al.* (2005) have single type of interest points and no time constraint involved. Considering the inaccuracy and incomplete issues, some works assigned scores or probabilities to each locations and the result path should pass the locations with high probabilities (Kanza *et al.*, 2008b, 2009). In our query, the criteria is the path with shortest travel time which is different from shortest path (Xuan *et al.*, 2011). In addition, in some route search, the path can go through multiple locations of the same type (Kanza *et al.*, 2008b, 2009) while our path only visit one location for each type. Moreover, even if a lot of different constraints have been studied, there is no paper that draws time constraint into Route Search query. Although our proposed methods are significantly different from existing works (Ehliar & Liu, 2005), they are still worth to be reviewed as they become our motivation.

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

2.1 Route search query

Route search query was proposed by Yaron Kanza *et al.* in 2008 (Kanza *et al.*, 2008b). There are three semantics covered in this paper, such as given start point, end point and all types of user interests, the first semantics is to return the shortest route that goes via all relevant entities. A second semantics is to find the most-profitable route, which is the route having the highest accumulative relevance and the length of the route is within the given limit. A third semantics is to compute the most-reliable route, which goes through as much higher relevant entities as possible and the length of the route is within the given limit.

Route search query uses greedy insertion from both start point and end point to find the final path. In route search, similar as our methods, multiple location types are concerned and the final path must pass one location of each type in any order.

Outstanding Problems:

- Route search query chooses multiple criteria such as shortest distance, highest relevance or more relevant entities with higher relevance. While we choose the shortest travel time because we draw time constraint into the query and shortest distance cannot guarantee the shortest travel time.
- Route search query has a pre-defined path length limit while our methods are indifferent to travel distance.
- Route search query pre-defined the start and end point while our methods only give the start point.
- Route search query does not concern arriving time for any interest point while the optimum path of our methods should meet the time constraint of each type.

2.2 Incremental kNN

Incremental k nearest neighbor (*iKNN*; Zhao *et al.*, 2013) is given a set of candidate interest points to find the shortest path, which starts at query point and goes through k interest points. For example, find the shortest path, which goes through three restaurants from q. It is a novel kNN because the result is the shortest path and the interest points are visited one by one. While it is not a Route Search query neither because all candidate interest points are single type, while our Route Search query involves multiple types, which needs access to different data structures such as multiple Network Voronoi Diagram (Okabe *et al.*, 2000; Kolahdouzan & Shahabi, 2004; Zhao *et al.*, 2009) and the result path must cover every type.

iKNN uses network expansion as Incremental Network expansion (*INE*). 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 the expansion, once there is a path shorter than the boundary; shrink it until all possible branches are expanded out of boundary. iKNN is similar with our proposed methods, such as both methods have a boundary identifier (D/T_{max}) to shrink the expansion scope and numbers of points to be visited are given.

Outstanding Problems:

- *iKNN* considers all interest points as single type while our methods consider multiple location types.
- *iKNN* does not concern arriving time constraint for any interest point while the optimum path of our methods should meet the time constraint.

2.3 Additional specifications

Before we move on to the proposed methods, more specifications are described and compared with other existing works:

- *Why cannot kNN solve Route Search query?*
  kNN cannot guarantee all locations to be fully covered, since the distance from query point to all interest points is the only criteria, and not the complete path.

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

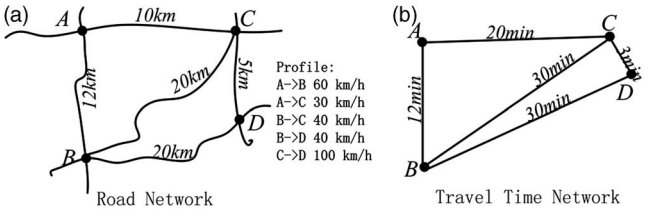


Figure 1 Road network vs. travel time network

- *What are the differences between incremental kNN and Route Search query?*
  In incremental *k*NN, all interest points are considered as single type while in route search query, there are multiple types of interest points and the optimal path should go through all types of interest points.
- *Can multiple 1_NN find all interest points?*
  If we use 1_NN to find the nearest neighbor until all types have been found, there is high possibility that the final path is not the most optimum one. The first nearest interest point may lead to a further distance to other interest points, as a result, it does not yield an optimum path. This is a common local minimal problem in scheduling.
- *Why does the shortest time need to be used instead of the shortest distance?*
  Since the problem of answering Route Search queries is a generalization of the traveling salesman problem, it is unlikely to have an efficient solution, that is, there is no polynomial-time algorithm that solves the problem (unless $P = \text{NP}$; Kanza *et al.*, 2008a). Hence, as a solution, this paper incorporates time constraint in order to prune as many expansion branches as possible and makes the query more realistic. If we use time constraint to prune the expansion branch, choosing traveling time as criteria is straightforward and can be adjusted to different travel time period.

3 Proposed methods

In this section, our proposed Route Search for fixed locations (*RFix*) and flexible locations (*RFlex*) are described. In each method, the query is given first followed by detailed explanations of the key issues, then after listing the algorithm, an example will illustrate the processing steps. As travel time is chosen as criteria, *travel time network* needs to be introduced first.

In Travel time network (Ku *et al.*, 2005), the measurement between nodes is the travel time instead of network distance. This is often more desirable because under certain conditions travel time is more meaningful than network distance, such as whether the path arriving at locations within their operating hours depends on the travel time, not travel distance. In this paper, we use the average travel speed in routine profile to estimate the approximate travel time. Figure 1 gives an example of a travel time network. We assume that the average travel time for each road segment is read from the traffic profile.

3.1 Route search for fixed locations (RFix)

When the query is a *RFix*, this query can be categorized as *RFix*. Example 1 illustrates a *RFix* query. It can be expressed as follows: Start at q at 4:30pm, find the optimum path whose travel time is shortest and this path should visit A, B, C and D between 9:00am–5:00pm, 9:00am–5:30pm, 4:30pm–5:40pm and 6:00pm–6:30pm, respectively. Now we can treat q, A, B, C and D as locations and invoke our proposed method *RFix* to find an optimum path. The pruning conditions are explained as follows:

Example 1: *Secretary will leave her office at 4:30pm. She has a plan to do*:

- *Fetch a suit from dry cleaner A and A's trading time is 9:00am–5:00pm.*
- *Fetch a contract from Company B and B's open hours are 9:00am–5:30pm.*

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

- *Send a report to manager's apartment C and he is at home 4:30pm–5:40pm.*
- *Pick up her son from kindergarten D and D's pick up period is 6:00pm–6:30pm.*

3.1.1. RFix *definition*

The *RFix* query can be formally defined like this:

DEFINITION 1 *RFix is a route search query consisting of:*
Input: Type Set $T = \{t_1, t_2, \ldots, t_n\}$, *Locations set* $P = \{p_1, p_2, \ldots, p_n\}$, $\forall\ p_i \in t_i$.
Output: A Path l which goes through all ps in P and distance$_l$ *is the shortest.*

3.1.2 *Pruning conditions*

Since the problem of answering Route Search queries is a generalization of the traveling salesman problem, it is unlikely to have an efficient solution, hence an efficient pruning method is crucial. With the prune conditions, the candidate permutation is greatly reduced and that is the basis of our solutions. Two pruning conditions are discussed in this section. First, definition for *Invalid Path* and *Valid Path* are introduced here.

DEFINITION 2 *A path is invalid when at least one location's arriving time followed by this path is out of its operating hours; otherwise if it visits all user defined location types, it is a valid path.*

$$\left.\begin{array}{l} \exists P_i \\ Path(q \rightarrow P_1 \rightarrow \ldots \rightarrow P_i \rightarrow \ldots \rightarrow P_j) \\ T(q \rightarrow P_1 \rightarrow \ldots \rightarrow P_i) \notin OperatingHour(P_i) \end{array}\right\} \Rightarrow Invalid(q \rightarrow P_1 \rightarrow \ldots \rightarrow P_i \rightarrow \ldots \rightarrow P_j) \tag{1}$$

Pruning condition 1: non-reversible visiting sequence. If $q \rightarrow P_i \rightarrow P_j$ is a valid path while $q \rightarrow P_j \rightarrow P_i$ is an invalid path as in Equation (1), (P_i, P_j) have non-reversible visiting sequence $(q \rightarrow P_i \rightarrow P_j)$. Hence, any solution visiting P_j before P_i should be pruned.

$$\left.\begin{array}{l} valid(q \rightarrow P_i \rightarrow P_j) \\ Invalid(q \rightarrow P_j \rightarrow P_i) \\ Type(P_i), Type(P_j) \subseteq LocationTypelist \end{array}\right\} \Leftrightarrow P_i \overrightarrow{\leftrightarrow} P_j \tag{2}$$

PROPOSITION 1 *Given a query point q and two locations* P_1 *and* P_2

$$\left.\begin{array}{l} StartTime > OpenTime(P_1) \\ StartTime > OpenTime(P_2) \\ CloseTime(P_1) < CloseTime(P_2) \\ Invalid(q \rightarrow P_1 \rightarrow P_2) \end{array}\right\} \nRightarrow Invalid(q \rightarrow P_2 \rightarrow P_1) \tag{3}$$

Proof If $CloseTime(P_1) < CloseTime(P_2)$ which means P_1 closes earlier than P_2, it is possible that (P_1, P_2) have a non-reversible visiting sequence $q \rightarrow P_1 \rightarrow P_2$ because if we visit P_2 first, when we arrive P_1, it is already closed. It is self-evidence.

If $CloseTime(P_1) < CloseTime(P_2)$ and $q \rightarrow P_1 \rightarrow P_2$ is invalid, we will prove that we cannot conclude $q \rightarrow P_2 \rightarrow P_1$ is invalid by contrapositive.

$$\left.\begin{array}{l} T(q, P_1) + T(P_1, P_2) \approx 2 \times T(q, P_1) > CloseT(P_2) \Rightarrow Invalid(q \rightarrow P_1 \rightarrow P_2) \\ T(q, P_2) + T(P_2, P_1) \approx T(q, P_1) < CloseT(P_1) \Rightarrow Valid(q \rightarrow P_2 \rightarrow P_1) \end{array}\right\} \Leftrightarrow \exists P_2 \overrightarrow{\leftrightarrow} P_1 \tag{4}$$

□

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

Pruning condition 2: Invalid sub-paths make the entire path invalid. Any permutation containing invalid sub-path should be pruned. See Proposition 2.

PROPOSITION 2

$$\left.\begin{array}{l} Invalid(q \rightarrow P_i \rightarrow P_j) \\ StartTime > Max(OpenTime(P_i), OpenTime(P_j)) \end{array}\right\} \Rightarrow Invalid(q \rightarrow \ldots \rightarrow P_i \rightarrow \ldots \rightarrow P_j) \tag{5}$$

Proof

$$\left.\begin{array}{l} T(q \rightarrow P_i \rightarrow P_j) > CloseTime(P_j) \\ T(q \rightarrow \ldots \rightarrow P_i \rightarrow P_k \rightarrow P_j) > T(q \rightarrow P_i \rightarrow P_j) \\ P_k \neq P_i \neq P_j \end{array}\right\}$$

$$\Rightarrow T(q \rightarrow \ldots \rightarrow P_i \rightarrow \ldots \rightarrow P_j) > CloseTime(P_j) \Rightarrow Invalid(q \rightarrow .. \rightarrow P_i \rightarrow .. \rightarrow P_j) \tag{6}$$

□

The *RFix* method is processed in the following steps and the algorithm is shown in Algorithm 1.

Algorithm 1: RFix(*q*,*StartTime*,*LocationSet*,*LocationOperatingHour*)

```
1: Load routine traffic speed and calculate travel time for all segments
2: Initial EntitySet = q + LocationSet
3: For any two in EntitySet, calculate its travel time
4: First=All locations whose travel time to q within earliest CloseTime
5: For any two locations P_i and P_j in LoctionSet, check q → P_i → P_j is valid or not. If it is invalid, put
   q → P_i → P_j in PruneList
6: CandidatePath=Permutations of LocationSet whose first point in First and contain no subpath in
   PruneList
7: Initial Total_TimeCost = ∞
8: for each candidate path in CandidatePath do
9:       TimeCost=sum up all travel time and once TimeCost>Time_TimeCost, terminate this loop
10:      if at some step, TimeCost is out of P_i's LocationOperatingHour then
11:         ignore this path and prune all path from CandidatePath whose has it as sub path
12:         TimeCost = ∞
13:      end if
14:      if Total_TimeCost > TimeCost then
15:         Total_TimeCost = TimeCost
16:      end if
17: end for
18: Final Total_TimeCost is travel time cost and its path is the optimum path
```

First, depending on current time period, retrieve the traffic speed and calculate the time cost between any two locations in the set, which includes the query point and all fixed locations. The process is similar to Dijkstra algorithm if the weight between entities are travel time cost.

Second, find all locations in *First* whose travel time to q is within the earliest close time, meaning that if the path goes to the other points first, the path already misses the location with the earliest close time. After checking its arriving time is in the operating hours, put it into *First*. As a result, *First* holds all possible locations which can be the first visited.

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

Third, for any two locations (P_i and P_j), calculate whether $q \to P_i \to P_j$ match i, j's operating hours or not. If not, record $q \to P_i \to P_j$ into *PruneList*. Then generate all permutations of visiting sequence whose first visiting node is in *First* and do not include any sub path in *PruneList*.

Fourth, for each candidate path, sum up its cost time and compare the time with time constraint. Once it exceeds the time constraint, ignore it and filter the other path who has the same sub path. For example if $q \to P_1 \to P_2 \to P_3$ fails to match time constraint and when query starts, P_1, P_2 and P_3 have opened already, $q \to P_1 \to P_5 \to P_2 \to P_6 \to P_3$ should be pruned as well.

Finally, compare the time cost of the path left and choose the optimum one.

3.1.3 A case study

To clarify the algorithm, a case study (see in Figure 2) is presented. This case study is based on Example 1. Table 1 shows the *RFix* filter process.

Firstly, $q \to B \to A$ will make A over its close time, so B is not in the *First* list. The same goes for C.

Secondly, according to Proposition 2, $q \to C_{4:55pm} \to B_{5:19pm}$ is valid and $q \to B_{5:18pm} \to C_{5:42pm}$ cannot meet C close time (5:40pm), so $B \overrightarrow{\leftrightarrow} C$. Add $q \to B \to C$ into *PruneList*.

Thirdly, the same as second step, add $q \to D \to B$ into *PruneList*.

Finally, generate the permutation whose first node is A or D (in *First*) and does not contain sub path in *PruneList*. In this case, only one path lists. After checking this path satisfy all time constraints, it is the result of this query ($q \to A_{4:35pm} \to C_{5:03pm} \to B_{5:27pm} \to D_{6:22pm}$).

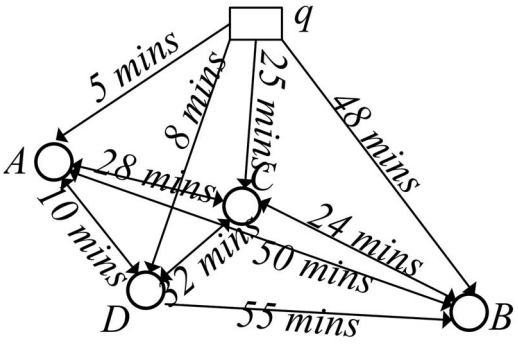


Figure 2 *RFix* example. *RFix* = Route Search for fixed locations

Table 1 *RFix* filter process

Traversal Permu.	Action and Reason	*CandidatePath*
$q \to B \ldots$	Pruned (B not in *First*)	None
$q \to C \ldots$	Pruned (C not in *First*)	None
$q \to A \to B \to C \to D$	Pruned	None
$q \to A \to B \to D \to C$	Invalid subpath	None
$q \to A \to D \to B \to C$	($q \to B \to C$)	None
$q \to D \to A \to B \to C$		–
$q \to D \to B \to A \to C$		None
$q \to D \to B \to C \to A$		None
$q \to A \to C \to D \to B$	Pruned	None
$q \to A \to D \to C \to B$	Invalid subpath	None
$q \to D \to A \to C \to B$	($q \to D \to B$)	None
$q \to D \to C \to A \to B$		None
$q \to D \to C \to B \to A$		None
$q \to A \to C \to B \to D$	Unpruned	$q \to A \to C \to B \to D$

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

*3.2 Route search for flexible locations (*RFlex*)*

When the user has pre-defined the location types whereby any locations of that type can be visited, this query can be categorized as *RFlex*, see example 2. As there are no fixed locations, we should distill the location types first according to the query specification. Then the query can be summarized as finding an optimum path which goes through these types within the time constraint.

Example 2: *Secretary will leave her office at 4:30pm. She has a plan to do*:

- *Deposit a cheque in any bank and all banks' trading hour is 10:00am–5:00pm.*
- *Buy a printer in any shop and all shops' operating hours is 11:00am–5:30pm.*
- *Post a letter in any post office and posts' trading hour is 10:00am–5:40pm.*
- *Buy some medicine in any pharmacy and all pharmacies' trading hour is 10:00am–6:30pm.*

3.2.1 RFlex *definition*

The *RFlex* query can be formally defined like this:

DEFINITION 3 *RFlex is a route search query consisting of:*
Input: Type Set $T = \{t_1, t_2, \ldots, t_n\}$, *Locations set* $P = \{p_1, p_2, \ldots, p_m\}$, $m > n$. $Type(p_i, \ldots, p_j) = t_k \in T$
Output: A Path l which goes through ps and ps cover all types in T. Also $distance_l$ *is the shortest.*

3.2.2 Pruning conditions

Traditional Route Search query is an NP complete problem and the focus of this section is how to use time constraint to prune most of expansion branches. Basically, our pruning conditions prune the path which leads to *unreachable point* or which is *out of time*.

PROPOSITION 3 **(Pruning condition 3):** *If P satisfies Equation 7, P_NN holds Ps' nearest NN of all types in unvisited_type, P, leads to a unreachable point. P will be pruned out candiate_next set (see Algorithm 3).*

$$\left.\begin{array}{l} TimeCost(P, P_NN(Type_i)) + TimeCost(q, P) > CloseTime(Type_i) \\ Type_i \in unvisited_type \end{array}\right\} \Rightarrow Unreachable(P) \quad (7)$$

Proof

$$\left.\begin{array}{l} CloseTime(Type_i) < TimeCost(P, P_NN(P_k) \leqslant TimeCost(P, \forall P_i) \\ P_i \,\&\, P_k \in Type_m \in unvisited_type \end{array}\right\}$$
$$\Rightarrow TimeCost(P, \forall P_i) > CloseTime(Type_i) \Rightarrow Unreachable(P) \quad (8)$$

□

Example of Pruning condition 3: Figure 3 is a case study of Example 2. Suppose P_1 is the *candidate_next* point, according to Proposition 3, whether it should be pruned or not depends on the data in Table 2. P_i is the nearest neighbor of P_1 in $Type_i$. As P_1's type is T_1, then $unvisited_type = T_2, T_3, T_4, T_5$. Find P_1's NN for each type in *unvisited_type* (Column 1 in Table 2). We can easily tell that P_2 is out of *OperatingHours* of $Type_2$, in other words, P_1 leads unvisited type T_2 unreachable. Consequently P_1 cannot be the *candidate_next* point.

PROPOSITION 4 **(Pruning condition 4):** *Locations whose types are in unvisited_Type within TimeCon (see equation 9) can be in candidate_next. Equation 10 which collects the candidate_next set can prune lots of interest points.*

$$StartTime > \max_{\forall i \in n}(OperatingHours(P_i)) \Rightarrow TimeCon = \max(CloseTime(unvisited_Type)) \quad (9)$$

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

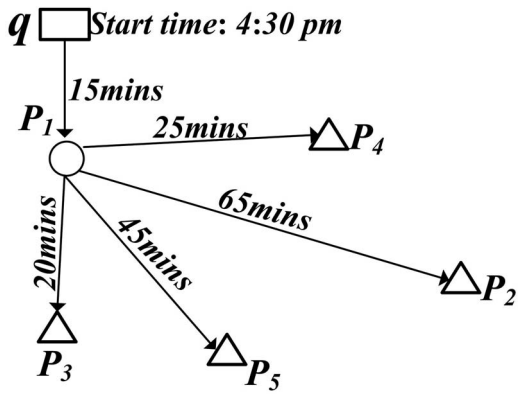


Figure 3 Pruning condition 3

Table 2 Proposition 3 Demo for *RFlex*

P_1 NN	Type	Oper.Time	Arriving Time
P_2	T_2	5:00pm	$StartT + 15 + 65 = 5{:}20\text{pm}$
P_3	T_3	5:30pm	$StartT + 15 + 45 = 5{:}00\text{pm}$
P_4	T_4	5:40pm	$StartT + 15 + 25 = 4{:}40\text{pm}$
P_5	T_5	6:30pm	$StartT + 15 + 20 = 4{:}35\text{pm}$

$$candidate_next(P) = \begin{cases} \forall P_i \\ TimeCost(P, P_i) < TimeCon \\ Type(P_i) \in unvisited_Type \end{cases} \tag{10}$$

Proof Suppose when we start the query, all locations are open, *TimeCon* should be set as the earliest close time in *unvisited_Type* as if the earliest close time type has not been visited, the locations that are going to be visited must be finished ahead of the earliest close time, otherwise when expanding to the earliest close time type, the arriving time is already out of the time constraint.

Algorithm 2: Untouch(*candidate_next,visited_type,T,TimeCost*)

```
1: for each P in candidate_next do
2:    P_type=get_Location_type()
3:    visited_type = visited_type + P_type
4:    unvisited_type = unvisited_type − P_type
5:    Initial boolean=0
6:    Find Ps′ nearest NN of all types in unvisited_type and put in P_NN
7:    P_TimeCost=get_POI_TimeCost()
8:    TimeCost=TimeCost+P_TimeCost
9:    for each NN in P_NN do
10:      NN_type=get_POI_type(NN)
11:      NN_TimeCost=get_Location_TimeCost(NN)
12:      if NN_TimeCost + TimeCost > NN_type's close time then
13:         boolean=1
14:         Return
15:      end if
16:   end for
17:   if boolean=1 then
18:      Delete this P from candidate_next
19:   end if
20: end for
```

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

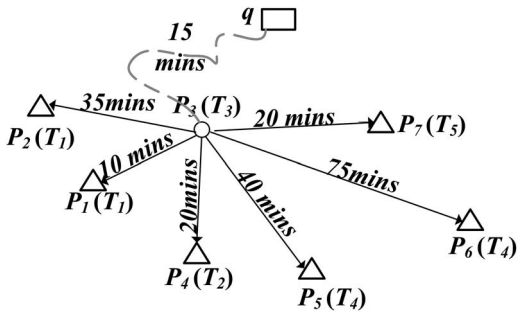


Figure 4 Pruning condition 4

Table 3 Proposition 4 Demo for *RFlex*

Visit_Type	Unvisit_Type	*TimeCon*	Cand_Next
T_3	T_1, T_2, T_4, T_5	5:00pm	P_1
T_1, T_3	T_2, T_4, T_5	5:30pm	P_4, P_5, P_7
T_2, T_3	T_1, T_4, T_5	5:00pm	P_1
T_1, T_2, T_3	T_4, T_5	5:40pm	P_5, P_7

Example of Pruning condition 4: Referring to Figure 4 and Table 3, suppose P_3 of T_3 is the current expansion point. In Table 3, each line represents one scenario as the *unvisited_Type* is different. Take the first line as an example, if the only visited type is T_3, then the *unvisited_Type* = $\{T_1, T_2, T_4, T_5\}$ and *TimeCon* = T_1's close time = 4:30pm. Only P_1 can be the *candidate_next* point because all of the other points are out of *TimeCon* according to Proposition 4. Algorithm 3 shows the process of how to find *candidate_next* points, which satisfies pruning conditions 4.

Algorithm 3: CandidateNext(*P*,*visited_type*,*T*,*TimeCost*)

1: *unvisited_type* = *T* − *visited_type*
2: *TimeCon* = Earliest *CloseTime* of *unvisited_type*
3: *Candidate_next* = all points whose type in *unvisited_type* and travel time within *TimeCon*

With these two pruning conditions, most expansions branches have been pruned. Although the execution time could potentially be exponential in the worst case because the pruning strategy is only heuristic, the pruning conditions do improve the performance significantly.

The detailed steps are shown as follows:

First, initialize T_{max} as ∞ and it is the boundary identifier, which will hold the final result. Initialize the *visited_type* as $\emptyset$, which is the collection of the location types that have been visited. Also, initialize T as all location types of user interest.

Second, according to the current time period, load routine traffic speed and get all interest points around q whose travel time to q is within *TimeCon*. Collect them in a set call *candidate_next* and prune these points using pruning condition 3.

Third, for any point in *candidate_next*, do the following. Remove any P in *candidate_next*, add P's type into *visited_type*. Then repeat the second step until all types have been visited. Once this path's cost time is shorter than T_{max}, replace T_{max} with its cost time.

Finally, T_{max} is the shortest travel time and its path is the optimum path.

The algorithm of *RFlex* is shown in Algorithm 4.

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

4 Performance evaluation

We used network and interest points data in Los Angeles in our experiments. We extracted eight different types of interest points to simulate different location types, including 15 parks, 29 coffee lounges, 31 bank branches, 54 hotels, 78 post offices, 158 pharmacies, 283 shops and 597 restaurants and all interest points are normally distributed.

In our experiments, we varied the following parameters: the number of location types, the congestion level (speed), density of interest points and the average time interval between locations to observe their effects on average processing time, memory as well as their improvement compared with the exhaustive traversal of all permutation approach.

4.1 Experimental results of RFix

Since number of locations highly influences our method's performance, we test the processing time (Figure 5(a)) and memory (Figure 5(b)) for two to eight locations based on three different traffic status (low, medium and high congestion).

From Figure 5(a) and Figure 5(b), we can easily tell that with the increasing number of locations, the processing time and memory are direct proportional to the number of locations. In addition, when the congestion level increases from low to high, the speed decreases at the same time and it causes a slightly increase of the processing time and memory. When it is extremely congested, the processing time and memory will cost $\sim$8–14% more than the lower level.

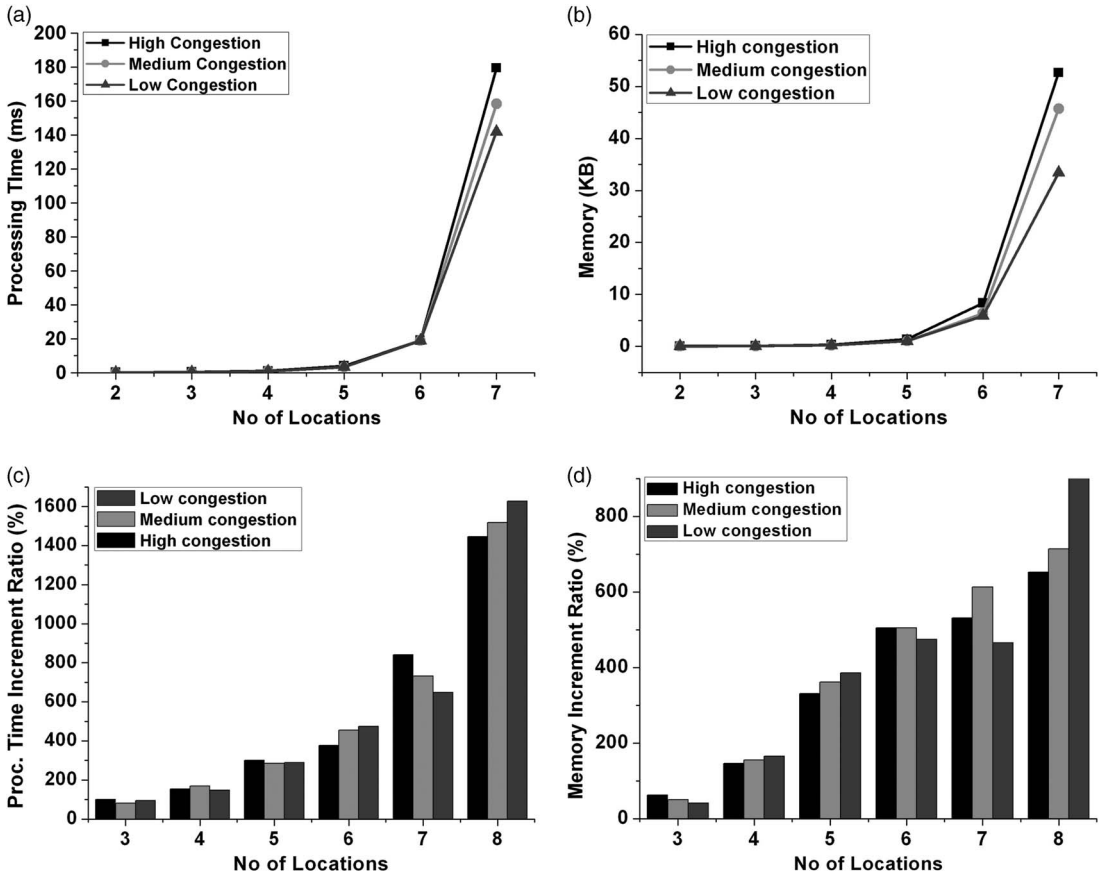


Figure 5 Time and memory comparison between different number of locations and traffic status in *RFix* and time and memory incremental ratio when adding more locations. *RFix* = Route Search for fixed locations

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

Algorithm 4: RFlex(q,$StartTime$,T,$LocationTtable$)

```
1: if q = start point then
2:     Load routine traffic speed in current period
3:     Initial T_max = ∞
4:     visited_type = ∅
5:     unvisited_type = T
6:     Static TotalTimeCost=0
7:     TimeCost = 0
8: end if
9: CandidateNext(q,visted_type,T,TimeCost)
10: Untouch(candidate_next,visited_type,T,P_TimeCost,TimeCost)
11: for each P in candidate_next do
12:     TotalTimeCost = TotalTimeCost+TimeCost
13:     StartTime = StartTime+TimeCost
14:     if visited_type = T then
15:         if TimeCost ≤ T_max then
16:             Update T_max = TotalTimeCost and record its path tree
17:         end if
18:         Break
19:     else
20:         RFlex(P,StartTime,T−visited_type,LocationTtable)
21:     end if
22: end for
23: Final T_max is Time Cost and its path is optimum path
```

While adding one more location into query list, there will be exponential growth in processing time and logarithmic growth in memory size required, referring to Figure 5(c) and Figure 5(d) when location number is greater than 8, the performance scale increases sharply.

Without using our method, *RFix* query can be solved by traversing all permutations at the cost of processing time and memory size. To improve its performance, we have proposed two pruning conditions in this paper. Figure 6(a) and Figure 6(b) give the performance comparison in processing time and memory between with and without pruning conditions. Figure 6(c) highlights the advantages of our pruning conditions.

To sum up, with the increasing number of locations, our methods with pruning conditions outperform the traversal methods in both processing time and memory aspects especially when the location number is greater than 3.

In this paper we include time constraint into Route Search query, normally user will follow rules: *earliest close, first visit*. In other words, the path is sorted by the close time sequence and this path is abbreviated as *PDT*. *PDT* ratio is the possibility that *PDT* is the optimum path (Equation 11). In this section, we analyze factors that will affect the visiting path, see Figure 7(a) and Figure 7(b).

$$PDT\ Ratio = \frac{number\ of\ times(PDT = optimum\ path)}{n\ times\ experiments} \tag{11}$$

Before analyzing our experiment results, first we define a factor called *Travel distance span in average Time interval to objects Distribution region* (*TTD*). Object Distribution Region (*ODR*) is the size of the region that user can arrive within the last location close time. *TTD* represents the coverage percentage of *ODR* in average time interval between locations.

$$ODR = \pi((\max CloseTime_i - StartTime) \times \bar{t})^2 \tag{12}$$

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

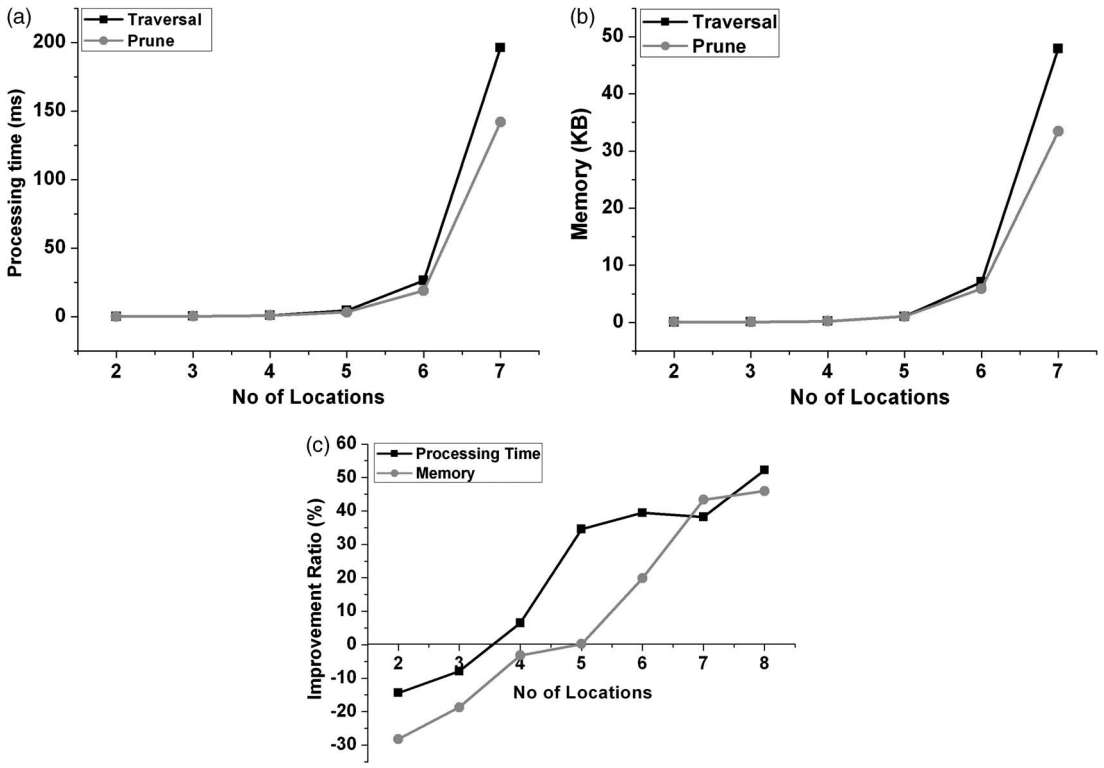


Figure 6 Proc. time and memory comparison between *RFix* and traversal methods. *RFix* = Route Search for fixed locations

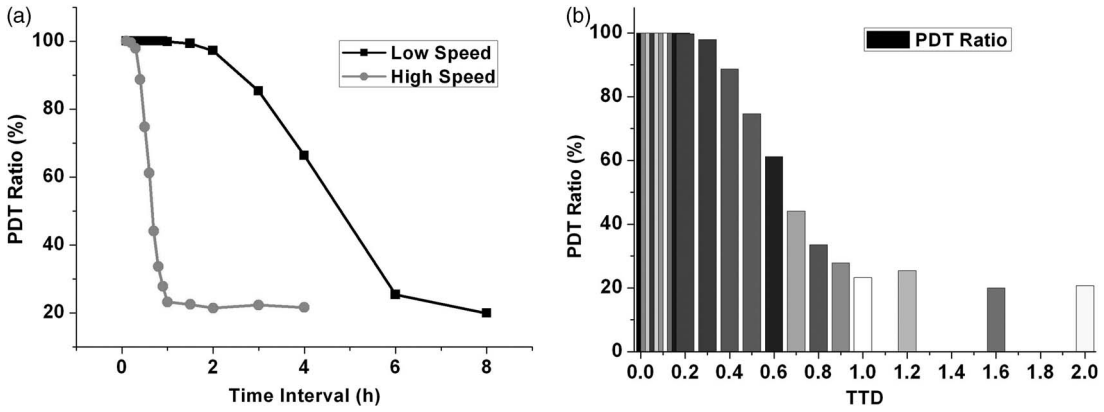


Figure 7 *PDT* is optimum (*RFix*) ratio. *RFix* = Route Search for fixed locations

$$TTD = \frac{\left(\sum_{i=1}^{n}(CloseTime_n - CloseTime_{n-1})/n\right) \times \bar{t}}{ODR} \tag{13}$$

Figure 7(a) shows that low travel speed will lead a high possibility that an optimum path is *PDT* until the average interval increases to 2 h or more, while high travel speed leads more possibility that optimum path is different from *PDT* when the average interval is >0.4 h. This result is in conformity with a common sense as if the average time interval between locations is small and speed is relatively slow, visiting locations along the close time sequence has a higher possibility to meet the time constraint because if we visit the later close time location, there will be a high possibility that we cannot catch the earlier location, and vice versa.

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

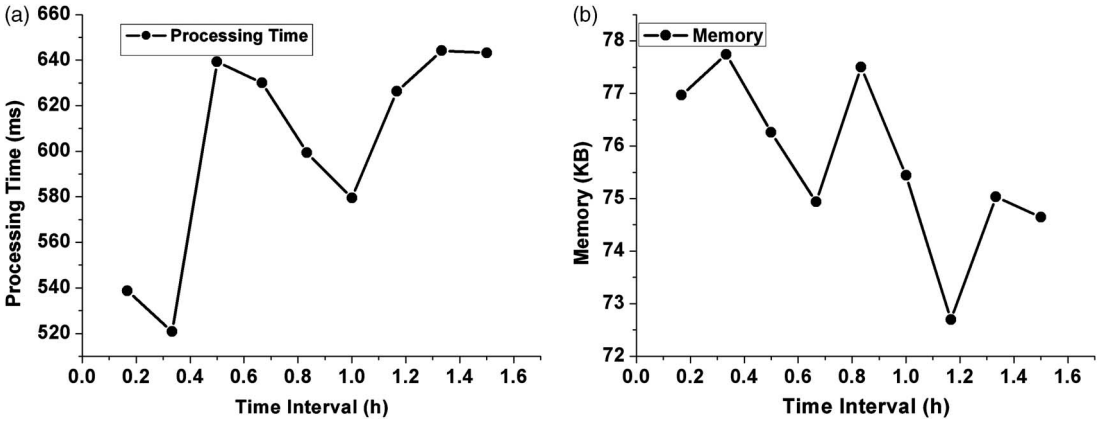


Figure 8 *RFlex* in different time interval. *RFix* = Route Search for fixed locations

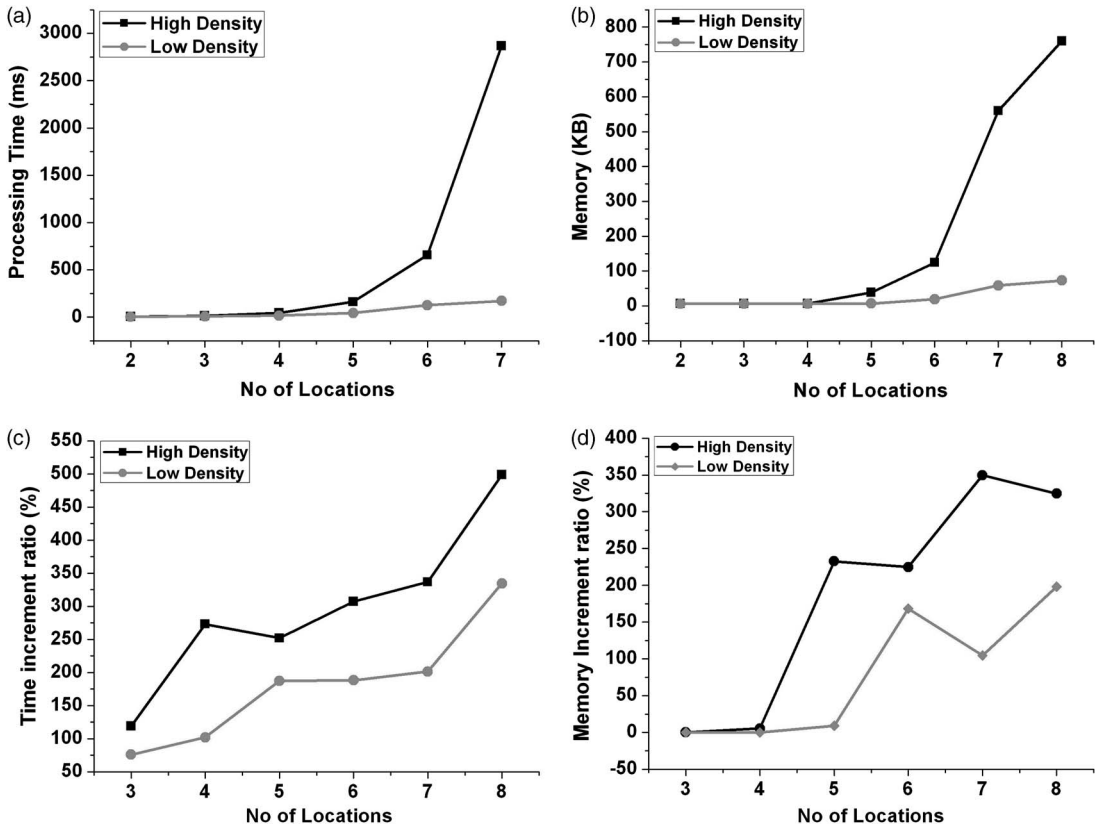


Figure 9 Processing time and memory comparison for different object densities in *RFlex* and Processing time and memory incremental ratio when adding more location. *RFix* = Route Search for fixed locations

Figure 7(b) illustrates that if *TTD* increases, which means its coverage percentage in average time interval increases, the possibility that *PDT* is the optimum path drops. The percentage remains stable until *TTD* increases to around 1.

4.2 Experimental results of RFlex

For *RFlex* query, the average time interval between locations is a factor that affects the visiting sequence. People generally believe that if the average time interval between locations is large, the performance falls badly. While Figure 8(a) and Figure 8(b) prove that this conjecture is not correct

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

because our *RFlex* goes down to get the first path and this path is set as boundary. As a result, the processing time is nearly linear and the memory decreases a little with the increasing average time interval between locations. As a result, our approach perform well even there is no time constraint when the number of locations remains constant.

The processing time and memory will steeply increase with the increasing number of locations and this is already proven in *RFix*. In this section, we compare the differences between high density distributed objects (e.g. restaurant, density = 0.09375) and low density distributed objects (e.g. parks, density = 0.0024). In our experiments, we test the processing time and memory for locations numbers from 2 to 7 refer to Figure 9(a) and Figure 9(b) for low and high densities of locations.

Figure 9(c) and Figure 9(d) show that if Route Search query involves more low density objects, with the increasing number of locations, the processing time and memory do increase, but much slower than the high density objects.

5 Conclusion and future work

This paper proposes novel Route Search methods with time constraint involving multiple object types. *RFix* provides a solution to users if they want to find the shortest travel time path for multiple location types and the locations of these types are fixed. *RFlex* helps users to find the shortest travel time path if the locations of these types are flexible. Both queries do not concern visiting sequence of objects subject to the final path as long as they arrive at each location within its operating hours. In our method, network Voronoi Diagram is used to find the candidate next visiting point within certain time range and it enriches the content of our mobile navigation system and gives more benefits to mobile users as well.

We performed several experiments to measure the performance of *RFix* and *RFlex* in different network conditions and object distributions. In general, our algorithm performs better if the number of locations is small. If the number of locations is <7, the performance is acceptable no matter how complex the road condition is and how objects are distributed. However, as expected, if the number of locations is >7, the processing time and memory increase sharply. In addition, if the average location close time interval is large, our optimum path has a high possibility that it is not *PDT*, which means discarding *PDT* and using our methods can give users a better path choice. Lastly when comparing *RFix* with the traditional traversal permutation method, it performs better and the advantage becomes obvious when number of locations increase up to 4.

In the future, we are going to incorporate intelligence techniques and context-aware in mobile navigation and mobile query processing (Waluyo *et al*., 2003, 2004; Goh & Taniar, 2004a, 2004b; Taniar & Rahayu, 2013). In addition, Route Search combined with dynamic query point or locations will also be investigated (Taniar & Rahayu, 2002, 2004; Waluyo *et al*., 2005b; Taniar & Goh, 2007; Mammeri *et al*., 2009). Performance in mobile query processing is always an issue, we plan to examine more thoroughly the performance issues of mobile query processing including the use of data broadcast techniques (Ilarri *et al*., 2012).

Acknowledgments

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

References

Dijkstra, E. W. 1959. A note on two problems in connection with graphs. *Numerische Mathematik* **1**(22), 269–271.

Ehliar, A. & Liu, D. 2005. Flexible route lookup using range search. In *Communications and Computer Networks*, Sanadidi, M. Y. (ed.). IASTED/ACTA Press, 345–350, ISBN 0-88986-546-9.

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

Goh, J. & Taniar, D. 2004a. Mining frequency pattern from mobile users. In *KES*, *Lecture Notes in Computer Science*, Negoita, M. G., Howlett, R. J. & Jain, L. C. (eds), **3215**, 795–801. Springer, ISBN 3-540-23205-2.

Goh, J. Y. & Taniar, D. 2004b. Mobile data mining by location dependencies. In *IDEAL*, *Lecture Notes in Computer Science*, Yang, Z. R., Everson, R. M. & Yin, H. (eds), **3177**, 225–231. Springer, ISBN 3-540-22881-0.

Huang, X. & Jensen, C. S. 2004. In-route skyline querying for location-based services. In *W2GIS*, *Lecture Notes in Computer Science*, Kwon, Y. J., Bouju, A. & Claramunt, C. (eds), **3428**, 120–135. Springer, ISBN 3-540-26004-8.

Ilarri, S., Mena, E., Illarramendi, A., Yus, R., Laka, M. & Marcos, G. 2012. A friendly location-aware system to facilitate the work of technical directors when broadcasting sport events. *Mobile Information Systems* **8**(1), 17–43.

Kanza, Y., Safra, E. & Sagiv, Y. 2009. Route search over probabilistic geospatial data. In *SSTD*, *Lecture Notes in Computer Science*, Mamoulis, N., Seidl, T., Pedersen, T. B., Torp, K. & Assent, I. (eds), **5644**, 153–170. Springer, ISBN 978-3-642-02981-3.

Kanza, Y., Safra, E., Sagiv, Y. & Doytsher, Y. 2008b. Heuristic algorithms for route-search queries over geographical data. In *Proceedings of ACM GIS*, Aref, W. G., Mokbel, M. F. & Schneider, M. (eds). Irvine, California, USA, 11. ACM Press, November.

Kolahdouzan, M. R. & Shahabi, C. 2004. Voronoi-based k nearest neighbor search for spatial network databases. In *Proceedings of 30th VLDB*, Nascimento, M. A., Özsu, M. T., Kossmann, D., Miller, R. J., Blakeley, J. A. & Schiefer, K. B. (eds). Toronto, Canada, 840–851. Morgan Kaufmann Publishers Inc., ISBN 0-12-088469-0.

Ku, W.-S., Zimmermann, R., Wang, H. & Wan, C.-N. 2005. Adaptive nearest neighbor queries in travel time networks. In *Proceedings of ACM GIS*, Shahabi, C. & Boucelma, O. (eds). Bremen, Germany, 210–219. ACM Press, November.

Mammeri, Z., Morvan, F., Hameurlain, A. & Marsit, N. 2009. Location-dependent query processing under soft real-time constraints. *Mobile Information Systems* **5**, 205–232.

Okabe, A., Boots, B., Sugihara, K. & Chiu, S. N. 2000. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, 2nd edition. John Wiley and Sons Ltd.

Papadias, D., Zhang, J., Mamoulis, N. & Tao, Y. 2003. Query processing in spatial network databases. In *Proceedings of 29th VLDB*, Freytag, J. C., Lockemann, P. C., Abiteboul, S., Carey, M. J., Selinger, P. G. & Heuer, A. (eds). Berlin, Germany, 802–813. Morgan Kaufmann Publishers Inc., ISBN 0-12-722442-4.

Pearson, J. & Guesgen, H. W. 1998. Some experimental results of applying heuristic search to route finding. In *Proceedings of FLAIRS Conference*, Cook D. J. (ed.). Sanibel Island, Florida, USA, 394–398. AAAI Press, ISBN 1-57735-051-0.

Roussopoulos, N., Kelley, S. & Vincent, F. 1995. Nearest neighbor queries. In *Proceedings of ACM SIGMOD*, San Jose, California, 71–79. ACM Press.

Safar, M. 2005. K nearest neighbor search in navigation systems. *Mobile Information Systems* **1**(3), 207–224.

Taniar, D. & Goh, J. 2007. On mining movement pattern from mobile users. *IJDSN* **3**(1), 69–86.

Taniar, D. & Rahayu, J. W. 2002. A taxonomy of indexing schemes for parallel database systems. *Distributed and Parallel Databases* **12**(1), 73–106.

Taniar, D. & Rahayu, J. W. 2004. Global parallel index for multi-processors database systems. *Information Science* **165**(1–2), 103–127.

Taniar, D. & Rahayu, W. 2013. A taxonomy for nearest neighbour queries in spatial databases. *Journal of Computer and System Sciences* **79**(7), 1017 –1039.

Terrovitis, M., Bakiras, S., Papadias, D. & Mouratidis, K. 2005. Constrained shortest path computation. In *Proceedings of SSTD*, *Lecture Notes in Computer Science*, Medeiros, C. B., Egenhofer, M. J. & Bertino, E. (eds), **3633**, 181–199. Springer, ISBN 3-540-28127-4.

Waluyo, A. B., Srinivasan, B. & Taniar, D. 2003. Optimal broadcast channel for data dissemination in mobile database environment. In *Proceedings of APPT*, *Lecture Notes in Computer Science*, Zhou, X., Jähnichen, S., Xu, M. & Cao, J. (eds), **2834**, 655–664. Springer, ISBN 3-540-20054-1.

Waluyo, A. B., Srinivasan, B. & Taniar, D. 2004. A taxonomy of broadcast indexing schemes for multi channel data dissemination in mobile database. In *Proceedings of AINA*. IEEE Computer Society, Fukuoka, Japan, 213–218, ISBN 0-7695-2051-0.

Waluyo, A. B., Srinivasan, B. & Taniar, D. 2005b. Research in mobile database query optimization and processing. *Mobile Information Systems* **1**(4), 225–252.

Xuan, K., Zhao, G., Taniar, D., Safar, M. & Srinivasan, B. 2011. Voronoi-based multi-level range search in mobile navigation. *Multimedia Tools and Applications* **53**(2), 459–479.

Xuan, K., Zhao, G., Taniar, D. & Srinivasan, B. 2008. Continuous range search query processing in mobile navigation. In *Proceedings of ICPADS*. IEEE, Melbourne, Australia, 361–368.

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press

Xuan, K., Zhao, G., Taniar, D., Rahayu, W., Safar, M. & Srinivasan, B. 2011. Voronoi-based range and continuous range query processing in mobile databases. *Journal of Computer and System Sciences* **77**(4), 637–651.

Yoo, J. S. & Shekhar, S. 2005. In-route nearest neighbor queries. *GeoInformatica* **9**(4), 117–137.

Zhang, Q. 2008. Hierarchical route representation, indexing, and search. *IEEE Pervasive Computing* **7**(2), 78–84.

Zhao, G., Xuan, K., Rahayu, W., Taniar, D., Safar, M., Gavrilova, M. & Srinivasan, B. 2011. Voronoi-based continuous k nearest neighbor search in mobile navigation. *IEEE Transactions on Industrial Electronics* **58**(6), 2247–2257

Zhao, G., Xuan, K. & Taniar, D. 2013. Path kNN query processing in mobile systems. *IEEE Transactions on Industrial Electronics* **60**(3), 1099–1107.

https://doi.org/10.1017/S0269888914000058 Published online by Cambridge University Press