Time-Constrained Indoor Keyword-aware Routing: Foundations and Extensions

Harry Kai-Ho Chan[1*], Tiantian Liu[2], Huan Li[3] and Hua Lu[2]

[1]Information School, University of Sheffield, United Kingdom.
[2]Department of People and Technology, Roskilde University, Denmark.
[3]Department of Computer Science, Aalborg University, Denmark.

*Corresponding author(s). E-mail(s): h.k.chan@sheffield.ac.uk;
Contributing authors: tliu@ruc.dk; lihuan@cs.aau.dk;
luhua@ruc.dk;

Abstract

With the increasingly available indoor positioning technologies, indoor location-based services (LBS) are becoming popular. Among indoor LBS applications, indoor routing is particularly in demand. In the literature, there are several existing studies on indoor keyword-aware routing queries, each considering different criteria when finding an optimal route. However, none of these studies explicitly constraint the time budget for the route. In this paper, we propose a new problem formulation TIKRQ that considers the time needed for a user to complete the route, in addition to other criteria such as static cost and textual relevance. A set-based search algorithm and effective pruning strategies are proposed as the foundations of processing TIKRQ. To further enhance the practicability of TIKRQ, we study the extensions of TIKRQ and propose efficient solutions. First, we present two TIKRQ variants, namely preferred visiting order and absence of a target point. Second, we present a session-based TIKRQ that keeps track of and refines a user's routing results when the user changes the query parameters. We conduct extensive experiments on both real and synthetic datasets to verify the efficiency of our proposals.

Keywords: indoor space, indoor query processing, keyword-aware, routing

1 Introduction

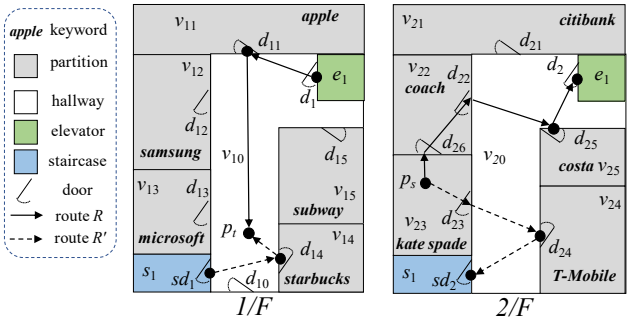
With recent developments in indoor positioning technologies and the widespread use of smartphones, indoor location-based services (LBS) [1] are becoming increasingly popular. Typical indoor LBS related applications include finding interested indoor objects and locations [18, 38, 42, 44, 45], indoor navigation and route planning [12, 25, 34–36], and indoor movement pattern mining [19, 20]. Among them, indoor route planning is particularly in demand, which assists users in planning a route satisfying their preferences, especially in an unfamiliar and large indoor environment like an airport or a shopping mall.

Consider that Alice has just passed the security check in the airport. As she has 90 minutes before the boarding time of the flight, she wants to buy a coffee, some souvenirs and a new charging cable. She can issue an indoor routing query from a source point (her current location) to a target point (i.e., the boarding gate), and specify the preferences by some keywords (e.g., coffee, souvenir, and charging cable). The query should return a route that passes through a shop that sells coffee, a souvenir shop, and a shop that sells charging cables. Most importantly, she should be able to complete the route within the 90-minute time constraint.

There are several existing studies on indoor keyword-aware routing query [12, 34–36], each of which considers different criteria when finding the result, including route distance, keyword relevance, and static cost. These existing route planning queries focus on minimizing the total length of the route that visits all requested keywords. However, none of these studies considers the *time constraint* as a hard constraint. In this paper, we propose a new problem formulation that is capable of taking all these criteria into account.

In practice, measuring the time needed to complete a route is much more reasonable and comprehensive than focusing on the route distance. First, instead of giving a concrete maximum walking distance (e.g., 500m), a user might feel more friendly to give the time she/he is willing to walk (e.g., 5 minutes), especially in some time-sensitive situations (e.g., as in our example, the user has to arrive at the boarding gate before a particular boarding time). Second, the distance metric overlooks the distances we travel by other means, such as elevators in the shopping malls and Automated People Mover (APM) in the airports. For example, the APM in Hong Kong International Airport needs 10 minutes to arrive at the farthest midfield concourse[1]. While these travel means do not incur any walking distance, the time needed on them should not be simply neglected when planning the route. A recent work [12] converts a time constraint into a distance constraint by multiplying the former by a maximum indoor walking speed, which, however, cannot handle these cases properly. Third, the waiting time of the shops (e.g., queuing time for a restaurant, or checkout time needed for a supermarket) should also be taken into account when returning a route to the user.

[1] https://en.wikipedia.org/wiki/Hong_Kong_International_Airport_Automated_People_Mover



Partition	Shop	Static Cost	Waiting Time (min.)
v_{11}	apple	6	8
v_{25}	costa	2	4
v_{14}	starbucks	3	3
v_{24}	T-mobile	8	5
...	...	...	...

Fig. 1: Running Example

Following a previous work [35], we also consider the static cost of shops in this paper. The static cost of a shop could refer to the average price of the products in the shop, or an estimated crowdedness of the shop. It could also be a composite cost of different criteria, such as price, popularity, and rating.

In this paper, we propose a new problem formulation that takes the route's *time needed* into account, rather than the (walking) distance. Specifically, we formulate a time-constrained indoor keyword-aware routing query (TIKRQ). A TIKRQ requires a source point p_s, a target point p_t, a set QW of query keywords and a time constraint Δ_{Max}. It returns the top-k routes from p_s to p_t such that each of their total time needed is less than Δ_{Max} and their costs are the minimum compared to those of the others. In our setting, the cost of a route captures the static cost and the textual relevance of the indoor partitions with respect to QW.

Figure 1 shows our running example, which consists of a floor plan of a shopping mall with two floors, and a table listing the shops' static costs and waiting time. Suppose a user will meet her friends in 20 minutes. During her spare time, she wants to buy a coffee and a charging cable for her phone. She then issues a query with her current location as the source point p_s (in v_{23} on 2/F), the meeting location as the target point p_t (in v_{10} on 1/F), two query keywords *coffee* and *charging cable*, and a route's time constraint of 20 minutes. Both routes R and R' are possible solutions of the query. Route R visits *costa* (which is a coffee shop) and *apple* (which sells charging cables), and uses the elevator e_1 to reach 1/F from 2/F, while route R' visits *starbucks* (which is another coffee shop) and *T-Mobile* (which also sells charging cables) and goes from 2/F to 1/F by the staircase s_1. Suppose the total walking time of R and

R' is 5 and 4 minutes, respectively, and the elevator takes 1 minute to go from 2/F to 1/F. The total time needed to complete R and R' is $5+1+8+4=18$ minutes and $4+5+3=12$ minutes, respectively. In this case, both R and R' can be completed within 20 minutes, while R should be more desirable to the user, since R has a total static cost of $2+6=8$ which is much smaller than that $8+3=11$ of R'.

In addition, we develop a concept of *unique partition set* to improve the diversity of the top-k results. Also, we organize and distinguish three types of indoor keywords to better capture the semantics of the query keywords. To answer the TIKRQ, we propose an algorithm based on a novel set-based search strategy to search for the top-k routes. Efficient pruning techniques and computation strategies are also designed to improve performance.

Compared to the existing studies [12, 34–36], our proposed TIKRQ (1) provides the flexibility for users to specify the *time constraint* of the routes, (2) is more comprehensive as it also considers the static cost of the partitions, (3) better organizes the keywords in an indoor setting by introducing an additional *category-word*, and (4) adopts the concept of unique partition set to improve the diversity of the top-k results.

To further enhance the practicability of TIKRQ, we study two extensions of TIKRQ. First, we present two TIKRQ variants, namely preferred visiting order and absence of a target point, to help users to specify special requirements when issuing a TIKRQ. In particular, the two variants can capture the cases that a user wants the routes to visit the query keywords in a particular order, and a user does not have a specific target point for the routes, respectively. To handle the two variants, we adapt the set-based search strategy to find the top-k routes.

Second, we formulate a session-based TIKRQ that keeps track of and refines a user's routing results when the user changes the parameters. In some scenarios, a user may want to update the query parameters of a TIKRQ and obtain the new results. In the previous example that the user has 90 minutes free time before the boarding time of the flight, if the flight is delayed and thus the boarding time is postponed, the user's time budget could be relaxed accordingly. Thus, on top of the TIKRQ, we formulate the session-based TIKRQ to accommodate such application needs. In particular, we analyze all possible parameters' update, and design algorithms for four cases.

This journal version extends our conference paper [5] with substantial new technical contributions. First, we propose two variants of TIKRQ, namely preferred visiting order and absence of a target point, which improve the practicability of our problem (Section 5). Second, we propose the session-based TIKRQ problem that maintains and refines the result of TIKRQ when the user changes the parameters (Section 6). In particular, we analyze all possible scenarios thoroughly and design algorithms for four cases. Third, we experimentally evaluate the new proposals (Sections 7.1.4 to 7.1.9). Fourth, we conduct additional experiments on a real dataset to evaluate all our proposals (Section 7.2).

The contributions of this work are summarized as follows.

- We formulate the time-constrained indoor keyword-aware routing query (TIKRQ) that takes the routes' time needed into consideration. We also propose a concept of unique partition set to diversify the top-k routes. (Section 2)
- We propose a keyword organization for indoor keywords, and a method to compute textual relevance for routes. (Section 3)
- We design a set-based search algorithm with effective pruning techniques to resolve TIKRQ. (Section 4)
- We present two variants of TIKRQ, and adapt the set-based search algorithm to resolve both of them. (Section 5)
- We propose a session-based TIKRQ, analyze all possible scenarios and design algorithms for four cases. (Section 6)
- We conduct extensive experiments and case studies to evaluate the proposed techniques. (Section 7)

In addition, we review the related work in Section 8 and conclude the paper in Section 9.

2 Problem Definition

2.1 Preliminaries

Table 1 shows the frequently used notations in this paper.

Table 1: Notations

Notation	Description
v, d, p	Partition, door, and point in an indoor space
$v.cost, v.waitTime$	Static cost and waiting time of a partition v
e	Transport in indoor space (e.g., an elevator)
w	A word in a partition
QW	The set of query keywords
$KPS(R)$	The set of key partitions on route R
$PC(R)$	Partition cost of route R
$\gamma(R)$	Time cost of route R
$\rho(R)$	Textual relevance of route R
$cost(R)$	Route cost of route R
CKP	A set of candidate key partitions
S	A key partition set
$s = (q, TopKRoutes)$	A session-based TIKRQ
Θ_c	The parameter changed in a session-based TIKRQ

A partition and a transport are the basic building blocks in an indoor space. A point is located inside a partition or a transport. In an indoor routing, one needs to move from one door to another through their common partition or transport. Following [27], we use the mapping to capture the indoor topology: Given a door d_i, we use $D2P_{\sqsupset}(d_i)$ and $D2P_{\sqsubset}(d_i)$ to denote the set of partitions and transports that one can enter and leave through d_i, respectively.

Given a partition v_k or a transport e_k, and two different doors d_i and d_j, the intra-partition door-to-door distance from d_i to d_j is defined as

$$\delta_{d2d}(d_i, d_j) = \begin{cases} |d_i, d_j|_E, & \text{if } v_k \in D2P_{\sqsupset}(d_i) \text{ and } v_k \in D2P_{\sqsubset}(d_j) \\ 0, & \text{if } e_k \in D2P_{\sqsupset}(d_i) \text{ and } e_k \in D2P_{\sqsubset}(d_j) \\ \infty, & \text{otherwise} \end{cases}$$

In the case that d_i and d_j are in the same partition (say v_k), one can enter the partition from d_i and leave by d_j. We measure the distance between d_i and d_j by Euclidean distance. Other distance metrics such as obstacle distance can also be adopted here. Following [12], we handle the special case of $d_i = d_j$, which happens when one needs to enter a partition due to its keyword relevance but then leave it from the same door for further routing, as follows. We set $\delta_{d2d}(d_i, d_i)$ to be the double of the longest non-loop distance one can reach inside the partition from the door d_i. In the case that d_i and d_j are in the same transport, we simply regard the door-to-door distance from d_i to d_j as zero. To reflect the practical time needed to pass a transport, we assign each transport a waiting time, as to be detailed in Section 3.1.

Given a point p_i, we use $v(p_i)$ to denote the partition or transport that contains p_i. Given a partition v_i, we use $P2D_{\sqsupset}(v_i)$ and $P2D_{\sqsubset}(v_i)$ to denote the set of doors through which one can enter and leave the partition v_i, respectively. Similarly, given a transport e_i, we use $E2D_{\sqsupset}(e_i)$ and $E2D_{\sqsubset}(e_i)$ to denote the set of doors which one can enter and leave the transport e_i, respectively. Given a door d_k and a point p_i, the point-to-door distance and door-to-point distance are defined as

$$\delta_{pt2d}(p_i, d_k) = \begin{cases} |p_i, d_k|_E, & \text{if } d_k \in P2D_{\sqsubset}(v(p_i)) \\ 0, & \text{if } d_k \in E2D_{\sqsubset}(v(p_i)) \\ \infty, & \text{otherwise} \end{cases}$$

$$\delta_{d2pt}(d_k, p_i) = \begin{cases} |d_k, p_i|_E, & \text{if } d_k \in P2D_{\sqsupset}(v(p_i)) \\ 0, & \text{if } d_k \in E2D_{\sqsupset}(v(p_i)) \\ \infty, & \text{otherwise} \end{cases}$$

When the context is clear, we use $\delta_*(x_i, x_j)$ to indicate the distance from a point/door x_i to a point/door x_j.

2.2 Problem Definition

Definition 1 (Route [12]). *A **route** $R = (x_s, d_i, ..., d_n, x_t)$ is a path through a sequence of doors from point/door x_s to x_t. A route is a **complete route** if x_s and x_t are the source and target points, respectively. Otherwise, it is a **partial route**.*

We can easily obtain the partitions and transports that a route R passes using the aforementioned indoor topological mappings.

Consider the partial route R on 2/F in Figure 1, we have $R = (p_s, d_{26}, d_{22}, d_{25}, d_{25}, d_2)$. We know that R passes v_{23}, v_{22}, v_{25} and v_{20} since $R = (p_s \xrightarrow{v_{23}} d_{26} \xrightarrow{v_{22}} d_{22} \xrightarrow{v_{20}} d_{25} \xrightarrow{v_{25}} d_{25} \xrightarrow{v_{20}} d_2)$.

We use the term **relevant partition** [12] to refer to a partition that covers p_s, p_t or a subset of query keywords. Given a route R, a **key partition** of R is a partition that R has been through and that has the maximum keyword relevance (to be given in Definition 6) for at least one query keyword. We use $KPS(R)$ to denote the set of key partitions in R. Considering the example in Figure 1 with query keywords *starbucks* and *apple*, partitions v_{10}, v_{11}, v_{14}, and v_{23} are relevant partitions and $KPS(R) = \{v_{11}, v_{14}\}$.

Definition 2 (Partition Cost)**.** *We define the partition cost of a route as the sum of the static cost of its key partitions.*

$$PC(R) = \sum\nolimits_{v \in KPS(R)} v.cost$$

Note that any monotonic function can be used to model a route's partition cost. In this paper, we use the sum function for conciseness.

Definition 3 (Route Cost)**.** *We define the cost of a route as the linear combination of its partition cost and textual relevance, i.e.,*

$$cost(R) = \alpha \cdot \frac{PC(R)}{PC_{max} \cdot |QW|} + (1 - \alpha) \cdot (1 - \rho(R)) \tag{1}$$

where $\alpha \in [0, 1]$ is a user parameter, PC_{max} is the maximum partition cost in the indoor venue, and $\rho(R)$ is the textual relevance of R (to be defined in Section 3.2).

The parameter α controls the weighting between the partition cost and textual relevance, and can be tuned according to the user needs. A smaller α puts a larger weight on route's textual relevance, while a larger α focuses more on the partition cost. We vary and evaluate this parameter in Section 7.1.2.

We define our problem as follows.

Problem 1 (Time-Constrained Indoor Keyword-aware Routing)**.** *Given a source point p_s, a target point p_t, a set QW of query keywords, a time constraint Δ_{Max} and an integer k, a* Time-Constrained Indoor Keyword-aware Routing Query *$TIKRQ(p_s, p_t, QW, \Delta_{Max}, k)$ returns k complete routes with the smallest route cost, and each such a route R from p_s to p_t (i.e., $R = (p_s, ..., p_t)$) has a time cost $\gamma(R)$ less than the time constraint (i.e., $\gamma(R) < \Delta_{Max}$).*

Above, $\gamma(R)$ captures the time needed for R to complete the given routing query (to be defined in Section 3.1). We say that a route is a **feasible route** if it satisfies the time constraint. Therefore, the TIKRQ is to find k feasible routes with minimum cost.

We prove the NP-hardness of the problem as follows.

Theorem 1. *The TIKRQ problem is NP-hard.*

Proof We prove by reduction from the Orienteering Problem [13] which is NP-hard. Given a graph with a set of nodes and a set of edges, where each node is associated with a score, and each edge is associated with a travel time, the goal of the Orienteering Problem is to find a route that visits some nodes from a starting point p_s to a target point p_t, such that the total collected score through the route is maximized and the total time spent is within a given time budget L.

The problem can be reduced to TIKRQ problem as follows. For each node, we create a partition v_i and a category c_i. All these categories form the query keyword set. Each partition v_i is associated to the corresponding category c_i, and the static cost is set to 0. Then, we construct two partitions, corresponding to the source point and the target point, respectively. For each constructed partition, we set $\delta_*(x_i, y_i) = 0$, where x_i and y_i are any points or doors inside the partition. For each edge, we construct a transport that connects the corresponding pair of constructed partitions with the edge's travel time as its waiting time. We set $\Delta_{Max} = L$. The above transformation can be done in polynomial time. Clearly, the problem of solving the TIKRQ problem is equivalent to that of the Orienteering Problem. Thus, the TIKRQ problem is NP-hard.

□

To ensure the resulting routes are meaningful, we use the following two principles of indoor routing [12].

Principle of Regularity. Unlike traditional outdoor routing algorithms [2, 46] that exclude loops in a route to avoid endless route searching, we allow a regular route in the indoor space to have a loop of doors in some cases. Consider the example in Figure 1, a user who wants to visit partition v_{14} must enter and leave d_{14}, producing a partial route $(..., d_{14}, d_{14}, ...)$. The principle of regularity disqualifies a route that contains a loop without any key partitions in the loop. That is, we exclude loops in a route between any two key partitions. For example, for a query with the keyword *starbucks*, $R' = (p_s, d_{26}, d_{26}, d_{23}, sd_2, sd_1, d_{14}, d_{14}, p_t)$ is not allowed since v_{22} visited by the loop (d_{26}, d_{26}) is not a key partition of the query.

Principle of Diversity. The concepts of diversifying top-k results [30, 47] and prime route [12] inspire us to avoid homogeneous routes in our routing results. We propose a concept of **unique partition set**. Specifically, for each of the k resulting routes, its key partition set must be unique. That is, for any two resulting routes R and R', we must have $KPS(R) \neq KPS(R')$.

Table 2: Four Example Routes from p_s to d_2 Each Covering *costa* and *citibank*

R_1	$(p_s \xrightarrow{v_{23}} d_{26} \xrightarrow{v_{22}} d_{22} \xrightarrow{v_{20}} d_{21} \xrightarrow{v_{21}} d_{21} \xrightarrow{v_{20}} d_{25} \xrightarrow{v_{25}} d_{25} \xrightarrow{v_{20}} d_2)$
R_2	$(p_s \xrightarrow{v_{23}} d_{26} \xrightarrow{v_{22}} d_{22} \xrightarrow{v_{20}} d_{25} \xrightarrow{v_{25}} d_{25} \xrightarrow{v_{20}} d_{21} \xrightarrow{v_{21}} d_{21} \xrightarrow{v_{20}} d_2)$
R_3	$(p_s \xrightarrow{v_{23}} d_{23} \xrightarrow{v_{20}} d_{25} \xrightarrow{v_{25}} d_{25} \xrightarrow{v_{20}} d_{21} \xrightarrow{v_{21}} d_{21} \xrightarrow{v_{20}} d_2)$
R_4	$(p_s \xrightarrow{v_{23}} d_{23} \xrightarrow{v_{20}} d_{21} \xrightarrow{v_{21}} d_{21} \xrightarrow{v_{20}} d_{25} \xrightarrow{v_{25}} d_{25} \xrightarrow{v_{20}} d_2)$

Consider Figure 1 as an example. Suppose a user wants routes from p_s to d_2 while covering two keywords $QW = \{costa, citibank\}$ in the route. Several possible routes are listed in Table 2. For ease of illustration, we insert the partitions that connect two consecutive items in the route. We can see that $KPS(R_1)$=$KPS(R_2)$=$KPS(R_3)$ = $KPS(R_4)$= $\{v_{23}, v_{21}, v_{25}, v_{20}\}$. The four routes pass the same set of key partitions with different orders and different partial routes in-between. Thus, only one of the four routes should be included in the query result.

Note that this requirement provides a more diversified result than a prime route [12], as the unique partition set is more *restrictive* than the prime route. In particular, the concept of prime route only requires $SRP(R) \neq SRP(R')$, where $SRP(R)$ denotes the *sequence* of relevant partitions in R. In our example, R_1 and R_2 (or R_3 and R_4) could be in the prime route query result at the same time, since $SRP(R_1)$=$\langle v_{23}, v_{21}, v_{25}, v_{20}\rangle$ is different from $SRP(R_2) = \langle v_{23}, v_{25}, v_{21}, v_{20}\rangle$.

3 Time Cost And Textual Relevance

In this section, we detail the formulation of the time cost $\gamma(R)$ in Section 3.1 and the textual relevance $\rho(R)$ in Section 3.2.

3.1 Time Cost

In this paper, we consider two types of time for a route R.

Travelling Time. The travelling time of a route R, denoted by $t_{travel}(R)$, refers to the time needed for a user to complete R. As discussed in Section 1, both walking and taking a transport incur travelling time, and we model them as follows. Assuming the average human walking speed s_{walk} is 5km/hour[2], we can easily compute the travelling time on walking as the walking distance divided by the walking speed.

To model the travelling time on taking transport, consider a user taking an elevator for illustration. To take an elevator, the total journey time includes waiting (outside the elevator) and travelling (inside the elevator). The estimation of this waiting time can be based on the average value of previous records, which is beyond the scope of this paper. This paper assumes a fixed waiting

[2]We use a universal walking speed in this paper for ease of illustration, but the proposed method can be easily adapted to the walking speed tailored for partitions.

time (e.g., 30 seconds), rather than a distribution, for ease of illustration. Similar to computing the walking time, the travelling time on an elevator is the height of travel from one floor to another divided by the elevator's speed.

Based on the above, given two doors d_i and d_j that connect to a partition v or a transport e, the time needed to travel from d_i to d_j is defined as

$$\gamma(d_i, d_j) = \begin{cases} \frac{\delta_{d2d}(d_i,d_j)}{s_{walk}}, & \text{if } v \in D2P_{\sqsupset}(d_i) \text{ and } v \in D2P_{\sqsubset}(d_j) \\ \frac{|d_i,d_j|}{s_e} + e_{wait}, & \text{if } e \in D2P_{\sqsupset}(d_i) \text{ and } e \in D2P_{\sqsubset}(d_j) \\ \infty, & \text{otherwise} \end{cases}$$

where $|d_i, d_j|$ is the actual distance of the two doors in e, s_e is the moving speed of the transport, and e_{wait} is the waiting time of the transport. For simplicity, we assume that the start point and target point are located in partitions only[3].

The travelling time of $R = (p_s, d_i, \ldots, d_k, p_t)$ can be computed as follows.

$$t_{travel}(R) = \frac{\delta_*(p_s, d_i)}{s_{walk}} + \sum_{k=i}^{n-1} \gamma(d_k, d_{k+1}) + \frac{\delta_*(d_n, p_t)}{s_{walk}}$$

Partition Time. The partition time of a route R, denoted by $t_{part}(R)$, is the sum of time spent in the key partitions where the user stays to fulfill her purposes implied by the keywords, i.e.,

$$t_{part}(R) = \sum\nolimits_{v \in KPS(R)} v.waitTime$$

where $v.waitTime$ denotes the waiting time of the partition v. Similar to the transport's waiting time, we assume a fixed value for the waiting time in each partition.

Time Cost. Based on the above, we define the time cost of a route by the following cost function.

Definition 4 (Time Cost)**.** *Given a route R, the time cost of R, denoted by $\gamma(R)$, is defined as the sum of the travelling time and the waiting time of R.*

$$\gamma(R) = t_{travel}(R) + t_{part}(R) \tag{2}$$

3.2 Textual Relevance

Keywords in Indoor Space. In the literature, an **identity word** (i-word) [12] identifies the specific name of a partition (e.g., *starbucks, apple*), and a **thematic word** (t-word) [11, 12] refers to a tag relevant to that partition

[3]We do not consider the extreme case that the source and target points are located in the transport, but our technique can easily support it.

(e.g., *coffee, laptop*). In addition, we employ a **category word** (c-word) that specifies the type of the partition (e.g., *coffee shop, supermarket*). A partition can be associated with one c-word and one i-word, but a set of t-words. For example, a partition in a shopping mall is associated with a c-word *coffee shop*, an i-word *starbucks* and t-words *coffee, mocha, latte*; another partition can be associated with a c-word *electronics*, an i-word *apple*, and t-words *smartphone, laptop, headphone*. Note that it is possible to extend our organization to support one partition associated with multiple or hierarchical c-words, which is left for future work.

Insufficiency of Existing Setting. The previous work [12] differentiates two types of keywords associated with indoor partitions. In particular, they assumed that two partitions having the same i-word *must* have the same set of t-words. However, this assumption over-simplifies the case. For example, depending on the shop's size and location, two starbucks can have different menus. A smaller one might not have some products (e.g., cakes and juices) for sale, while the one close to a train station sells more grab-and-go foods. As another example, some ATMs offer different currencies in cash, while some others do not. Compared to the assumption and the limitations in the organization of indoor space keywords in [12], our keyword organization, which we introduce below, is more general and comprehensive.

We assume that the three sets of words are disjoint for ease of illustration. Given a partition v_i, a **P2I** mapping $P2I(v_i)$ maps v_i to its associated i-word, and a **P2T** mapping $P2T(v_i)$ maps v_i to its associated t-words. Given an i-word w_i, an **I2C** mapping $I2C(w_i)$ maps w_i to its associated c-word, and an **I2P** mapping $I2P(w_i)$ maps w_i to the partitions associated with it. Given a t-word w_t, a **T2P** mapping $T2P(w_t)$ maps w_t to the partitions associated with it. Given a c-word w_c, a **C2I** mapping $C2I(w_c)$ maps w_c to the associated i-words.

To better represent the real-world setting, we maintain P2I as a many-to-one mapping and I2P as a one-to-many mapping such that a partition can be associated with one i-word, and each i-word can be associated with multiple partitions. For example, there could be multiple *starbucks* in a mall. We maintain P2T and T2P as two many-to-many mappings, meaning that each partition can be associated with multiple t-words and vice versa. Besides, we maintain I2C as a many-to-one mapping and C2I as a one-to-many mapping. Figure 2 shows an example of the organization of indoor space keywords.

Given the organization described above, we are now ready to introduce the calculation of keyword relevance between the query keywords and a route as follows.

Keyword Relevance Computation. Given a set QW of query keywords, we first match each query word $w \in QW$ to the **candidate partitions** for facilitating the routing afterwards.

Definition 5 (Candidate Partitions)**.** *Given a query keyword $w \in QW$, its candidate partitions $\mathcal{CP}(w)$ is represented as a set of entries each of which is in*

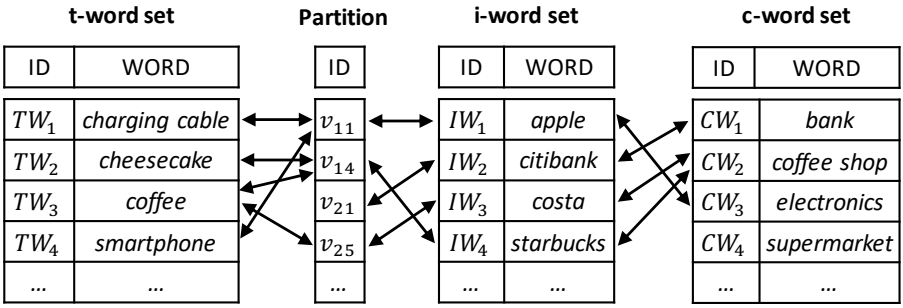


Fig. 2: Keyword Mappings in Indoor Space

the form of (v_i, rs)*, where* v_i *is the matching partition and* rs *is the relevance score between* v_i *and* w*. We discuss different cases based on the type of* w *as follows.*

- *Case 1 (*w *is a c-word): All partitions associated with the matching i-words in* $C2I(w)$ *are matched with* $rs = 1$*.*
- *Case 2 (*w *is an i-word): All partitions associated with the i-word* w *are matched with* $rs = 1$*. To enrich the result, we also include partitions associated with other i-words. In particular, all partitions associated with an i-word* w_i' *such that* $I2C(w_i') = I2C(w)$ *are matched with* $rs = 0.1$[4]*.*
- *Case 3 (*w *is a t-word): All partitions associated with the matching t-word* w *are matched with* $rs = 1$*. All partitions* v_j *associated with t-words* t_i' *such that* $t_i' \in \bigcup_{v_i \in T2P(w)} P2T(v_i)$ *are matched with* $rs = \frac{P2T(v_j) \cap \bigcup_{v_i \in T2P(w)} P2T(v_i)}{P2T(v_j) \cup \bigcup_{v_i \in T2P(w)} P2T(v_i)}$ *based on the Jaccard Similarity.*

Compared to [12], our definition has an extra case that w is a c-word, and it uses a different scoring scheme to handle the case that w is an i-word, which is designed based on our new keyword organization.

Definition 6 (Keyword Relevance)**.** *Given a route* R *and a query keyword* w_Q*, we define the keyword relevance of* w_Q *w.r.t.* R *as the maximum* rs *of* $v_i \in R$ *as follows.*

$$rel(R, w_Q) = \max_{(v_i, rs) \in \mathcal{CP}(w_Q) | v_i \in R} \mathcal{CP}(w_Q).rs$$

Definition 7 (Textual Relevance)**.** *Given a route* R*, we define the textual relevance* $\rho(R)$ *as the sum of keyword relevance of all query keywords as follows.*

$$\rho(R) = \Big(\sum\nolimits_{w_Q \in QW} rel(R, w_Q) \Big) / |QW|$$

where $|QW|$ *is the normalization term to make* $\rho(R)$ *fall in* $[0, 1]$*.*

[4]Any small value can be used here as long as the original i-word w has a higher score. The routes with w will have higher rankings than those with w_i'.

Consider our example in Figures 1 and 2. Suppose the query keywords are *coffee* and *charging cable* (both keywords are t-words). R passes the key partitions v_{11} and v_{25}, which is associated with *charging cable* (i.e., $v_{11} \in T2P(TW_1)$) and *coffee* (i.e., $v_{25} \in T2P(TW_3)$), respectively. Thus, we have $\rho(R) = \frac{1+1}{2} = 1$. If the query keywords are changed to *starbucks* (which is an i-word) and *electronics* (which is a c-word), v_{11} and v_{25} are still the key partitions of R, and we have $\rho(R) = \frac{0.1+1}{2} = 0.55$ since $I2P(v_{11}) = costa$ is of the same category *coffee shop* with *starbucks*, and $I2P(v_{25}) = apple$ is associated with the category *electronics*.

4 TIKRQ Processing Framework

In this section, we propose our Set-Based Search Algorithm to find the resulting routes. Before we present the algorithm, we extend the concept of skeleton distance [41] to skeleton time which will be used in our pruning rules. Given two indoor items x_i and x_j, the skeleton time $\gamma(x_i, x_j)_L$ can be used as a lower bound of the time needed from x_i to x_j.

$$\gamma(x_i,x_j)_L = \begin{cases} \frac{|x_i,x_j|_E}{s_{walk}}, & \text{if } x_i \text{ and } x_j \text{ are on the same floor} \\ \min\Bigg(\min_{\substack{sd_i \in SD(x_i),\\ sd_j \in SD(x_j)}} \frac{|x_i,sd_i|_E + \delta_{s2s}(sd_i,sd_j) + |sd_j,x_j|_E}{s_{walk}}, \\ \min_{\substack{ed_i \in ED(x_i),\\ ed_j \in ED(x_j)}} \Big(\gamma(x_i,ed_i)_L + \\ \gamma_{e2e}(ed_i,ed_j) + \gamma(ed_j,x_j)_L\Big)\Bigg), & \text{otherwise} \end{cases}$$

where $\gamma(x_i, x_j)_L$ is the time needed to walk in Euclidean distance from x_i to x_j if they are on the same floor. Otherwise, we find the time needed for the fastest path that goes through the staircase doors (e.g., $sd_i \in SD(x_i)$ and $sd_j \in SD(x_j)$) or the transport doors (e.g., $ed_i \in ED(x_i)$ and $ed_j \in ED(x_j)$) to reach x_j from x_i.

4.1 Set-Based Search Algorithm (SSA)

We give the following observation which provides a clue to developing an efficient algorithm for TIKRQ.

Observation 1 (Partition Set)**.** *Given a set S of key partitions, any route R formed by the partitions in S has the same partition cost and textual relevance.*

With a slight abuse of notations, we denote the partition cost and textual relevance of a set S of key partitions by $PC(S)$ and $\rho(S)$, respectively. It is easy to see that both metrics are not affected by the *order* of visiting, and thus $PC(S) = PC(R)$ and $\rho(S) = \rho(R)$ for any route R formed by the key partitions

in S. Based on this observation, we propose a set-based search algorithm *SSA* as follows.

High Level Idea. This algorithm searches for the resulting routes by focusing on the partition sets, as shown in Figure 3. For each set S of key partitions, we check whether any feasible route R exist (i.e., $\gamma(R) < \Delta_{Max}$). If such a route exists, the top-k results are updated accordingly.

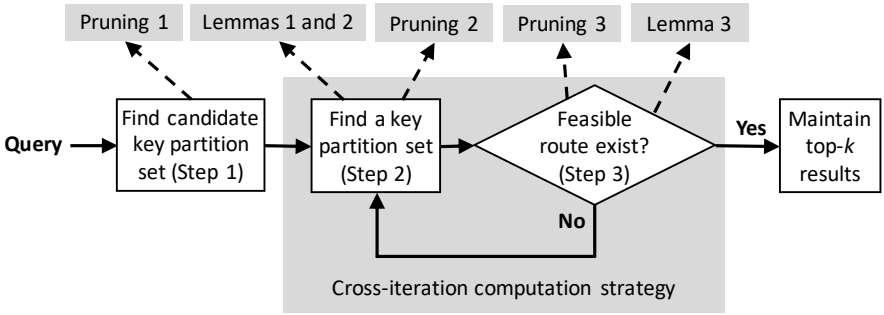


Fig. 3: Flow of Set-based Search Algorithm

The advantage of *SSA* is that it *separates* the time needed of a route R from its cost part. Thus, effective pruning techniques based on partition cost and textual relevance can be applied to filter out unpromising routes quickly, without performing the time-consuming route search and expansion. Compared to the graph-based algorithms in [12], *SSA* maps *multiple* routes into *one* set, resulting in a much smaller search space. Note that this search strategy naturally conforms with our requirement of the unique partition set, which improves the diversity of our top-k results.

Specifically, *SSA* maintains a list *TopKRoutes* storing the current top-k best feasible routes, and *curKCost* storing the cost of the k-th route found so far. It has four major steps.

- *Step 1 (Candidate Key Partition Set Finding):* Find the candidate key partition set (CKP) of candidate key partitions from the set of query keywords QW.
- *Step 2 (Key Partition Set Finding):* Find a set S of key partition set from CKP to be the key partition set of a route R to be found.
- *Step 3 (Feasible Route Finding):* Find a feasible route R which starts from p_s, passes all key partitions in S and ends at p_t (if any), and update *TopKRoutes* with R correspondingly if $cost(R) < curKCost$.
- *Step 4 (Iterative Step):* Resume Step 2 until all key partition sets are traversed.

The above search strategy is based on the set of all possible combinations of CKP. A straightforward implementation of this strategy would enumerate $2^{|CKP|}$ key partition sets, and each set would have $|S|!$ possible routes. This is prohibitively expensive in practice. Thus, we need a careful design to prune the search space effectively. In the following, we discuss the pruning techniques enjoyed by *SSA*.

4.1.1 Pruning at Step 1

Pruning Rule 1 (Candidate Key Partitions)**.** *For a partition v_i in a key partition set S, if its time cost lower bound $LB(\gamma(v_i)) > \Delta_{Max}$, then v_i can be pruned, where*

$$LB(\gamma(v_i)) = \gamma(p_s, v_i, p_t) + t_{wait}(v_i)$$

$$\gamma(p_s, v_i, p_t) = \min_{\substack{d_i \in P2D_{\sqsupset}(v_i),\\ d_j \in P2D_{\sqsubset}(v_i)}} \big(\gamma(p_s, d_i)_L + \frac{\delta_{d2d}(d_i, d_j)}{s_{walk}} + \gamma(d_j, p_t)_L\big)$$

4.1.2 Pruning at Step 2

Firstly, we utilize an inverted file indexed by QW to organize CKP to avoid generating sets that contain 'unnecessary' key partitions. That is, only the partition sets with each partition *contributing* to a query keyword will be considered. Note that in this way, we also bound the size of each set S to $|QW|$.

Secondly, given a subset S' of the key partition set S to be generated, we impose a cost lower bound $cost_{LB}(S|S')$ of S, as follows.

$$cost_{LB}(S|S') = \alpha \frac{PC(S')}{PC_{max} \cdot |QW|} + (1-\alpha)(1 - \frac{\rho(S') + (|QW| - |S'|)}{|QW|})$$

Lemma 1 (Set Cost)**.** *Let S be a key partition set and $S' \subset S$, we have $cost(S) \geq cost_{LB}(S|S')$.*

Proof Since $|S| = |QW| > |S'|$ and each key partition has its relevance $rs \leq 1$ for each query keyword, we have $\rho(S) \leq \rho(S') + (|QW| - |S'|)$. It is easy to see that $PC(S') \leq PC(S)$. Thus, we have $cost(S) > cost_{LB}(S|S')$. □

The above Lemma suggests that if $cost_{LB}(S|S') > curKCost$, we can terminate the enumeration on S'.

Thirdly, we sort the partitions v_i in each inverted list in ascending order of $f(v_i)$, where

$$f(v_i) = \alpha \frac{PC(v_i)}{PC_{max}} - (1-\alpha)\ v_i.rs$$

Lemma 2 (List Ordering)**.** *Let v_i and v_j be two key partitions in an inverted list with $f(v_i) \leq f(v_j)$, S' be a key partition set containing v_i and $S'' = S' \setminus \{v_i\} \cup \{v_j\}$. Then, we have $cost_{LB}(S|S') \leq cost_{LB}(S|S'')$.*

Proof Consider the set $S'_o = S' \setminus \{v_i\}$. It can be proven that $cost_{LB}(S|S'_o) \geq cost_{LB}(S|S') - \frac{f(v_i)}{|QW|}$. Since $S'' = S'_o \cup \{v_j\}$, we have $cost_{LB}(S|S'_o) + \frac{f(v_j)}{|QW|} \geq cost_{LB}(S|S'')$. As $f(v_i) \leq f(v_j)$, we have $cost_{LB}(S|S') \leq cost_{LB}(S|S'')$. □

By ordering the inverted lists in this way, we can impose an early stopping condition: If $cost_{LB}(S|S') > curKCost$, we can terminate the enumeration of the remaining partitions in the list.

Fourthly, some key partition sets can be excluded by considering their upper bound of total waiting time.

Pruning Rule 2 (Set Waiting Time)**.** *Given a key partition set S, we upper bound S's total waiting time of partitions in S, $\sum_{v \in S} v.waitTime$, by $waitTime_{max}$, which is defined as follows.*

$$waitTime_{max} = \Delta_{Max} - \gamma(p_s, p_t)_L$$

Note that $waitTime_{max}$ can be pre-computed because $\gamma(p_s, p_t)_L$ is identical for all queries with the same pair of p_s and p_t.

4.1.3 Algorithm SSA

We design *SSA* as shown in Algorithm 1. Specifically, it maintains a list *TopKRoutes* storing the k best-known routes found so far (line 2). Then, it finds the set of candidate key partitions *CKP* (line 4), by utilizing Pruning 1. Next, it performs an iterative process as follows (lines 5 to 13). It iterates through each key partition set S, and if S passes our lower bound cost checking (Lemma 1) and waiting time checking (Pruning 2), it finds the feasible route R of S by $findFeasibleRoute()$ (to be detailed in Algorithm 2). If such a route R exists, we update the *TopKRoutes* by R. The algorithm terminates when all sets have been processed. The *TopKRoutes* is then returned as the result.

Algorithm 1 SSA (p_s, p_t, QW, Δ_{Max}, k)

```
1:  if δ(p_s, p_t) > Δ_Max then return ∅
2:  TopKRoutes ← ∅
3:  waitTime_max ← Δ_Max − δ(p_s,p_t) / s_walk
4:  CKP ← ∪_{w_Q ∈ QW} CP(w_Q)                                  ▷ Step 1
5:  for each possible subset S of CKP do                        ▷ Step 2
6:      if cost_LB(S) > curKCost then continue
7:      if Σ_{v ∈ S} v.waitTime > waitTime_max then continue
8:      R ← findFeasibleRoute(S, p_s, p_t, Δ_Max)               ▷ Step 3
9:      if R ≠ ∅ then
10:         update TopKRoutes with R
11:         curKCost ← cost of the k-th route in TopKRoutes
12:     end if
13: end for
14: return TopKRoutes
```

One remaining issue is that given a set S, how to efficiently find the feasible route, if it exists. We present an algorithm for that in the following section.

4.2 Feasible Route Search

Given a set S of partitions, we want to find a feasible route from p_s to p_t that passes all partitions in S. A naive approach is to try all permutations of the partitions in S. Instead, we propose a best-first search algorithm that can find the feasible route efficiently. It returns the feasible route R if it exists. Otherwise, it returns $\emptyset$.

Before we present the algorithm, we introduce some pruning techniques to speed up the feasible route search. First, not all partial routes need to be explored. In particular, given a partial route $R = \{p_s, ..., d_k\}$, we compute its lower bound time cost and introduce a pruning as follows.

Pruning Rule 3 (Route Time Cost)**.** *A partial route* $R = \{p_s, ..., d_k\}$ *can be pruned if* $\gamma_{LB}(R) = \gamma(R) + \gamma(d_k, p_t)_L \geq \Delta_{Max}$.

Second, not all doors in each key partition need to be considered when expanding a partial route. Given a key partition v', a partial route $R = \{p_s, ..., d_{y-1}, d_y\}$ that has expanded to v', if both d_{y-1} and d_y are connected to v', R' can be safely discarded. Formally, we have the following lemma.

Lemma 3 (Route Pruning)**.** *Consider a partial route* $R' = \{p_s, .., d_{y-1}, d_y\}$, *where* $d_{y-1}, d_y \in P2D_{\sqsupset}(v')$. *There exists a route* R *that connects to* v' *and is faster than* R'.

Proof Consider another partial route $R = \{p_s, .., d_{y-1}\}$. It is easy to see that $\gamma(R') \geq \gamma(R)$ since $\gamma(R') = \gamma(R) + \frac{|d_{y-1}, d_y|_E}{s_{walk}}$. □

Based on the above Lemma, a partial route $R' = \{p_s, .., d_{y-1}, d_y\}$ can be pruned if $d_{y-1}, d_y \in P2D_{\sqsupset}(v')$.

Algorithm 2 presents the $findFeasibleRoute$ algorithm. Specifically, a minimum priority queue Q (initialized in line 1) is used to handle the order of route expansion. An element in Q is a four-tuple (v, R, γ_{LB}, S') that stores the local information of the current partial route, where v is the last partition that R reaches, $R = \{p_s, d_i, ..., d_k\}$ is the partial route that has been expanded so far, γ_{LB} is the lower bound time cost of $R \cup \{p_t\}$, and S' is the set of remaining partitions that have not been explored by R yet. The elements in Q are sorted in ascending order of $|S'|$.

The algorithm initializes a route R_0 by p_s and puts it into Q (lines 2 to 3). It then performs the expansion iteratively (lines 4 to 23). In each iteration, it pops out the element (v, R, γ_{LB}, S') with the smallest $|S'|$ from Q (line 5), and check if S' is empty. If so, all key partitions in (the original) S is covered by R, and it connects R from d_k to p_t to form a complete route R'. If $\gamma(R') < \Delta_{Max}$, it returns R' immediately as R' is a feasible route (lines 7 to 11). Otherwise, it expands the current partial route to cover a key partition $v' \in S'$ (lines 12 to 21). For each $d_y \in P2D_{\sqsupset}(v')$, it finds the fastest route from d_k to d_y, checks if the route passes the checking (Lemma 3). If so, it generates a new route R'

Algorithm 2 findFeasibleRoute ($S = (v_1, v_2, ..., v_n)$, p_s, p_t, Δ_{Max})

```
1: Initialize a priority queue Q
2: R_0 ← (p_s)
3: Q.push(v(p_s), R_0, 0, S)
4: while Q is not empty do
5:     (v, R, γ_LB, S') ← Q.pop()
6:     d_k = R.tail
7:     if S' = ∅ then
8:         find fastest route from d_k to p_t
9:         R' ← append (d_k, ..., p_t) to R
10:        if γ(R') ≤ Δ_Max then return R'
11:    end if
12:    for each v' ∈ S' do
13:        for each d_y ∈ P2D_⊐(v') do
14:            find fastest route from d_k to d_y
15:            if d_{y-1} ∈ P2D_⊐(v') then continue
16:            R' ← append (d_k, ..., d_y) to R
17:            γ'_LB ← γ(R') + γ(d_y, p_t)_L
18:            if γ'_LB ≤ Δ_Max then
19:                Q.push(v', R', γ'_LB, S' \ {v'})
20:            end if
21:        end for
22:    end for
23: end while
24: return ∅
```

by appending the fastest route to R, and pushes it to Q if it passes the time constraint checking (Pruning 3). The expansion continues until all elements in Q have been processed. It returns $\emptyset$ if no feasible route can be found.

Cross-Iteration Computation Strategy. To further speed up the algorithm, we have the following strategy to store and reuse the information computed in the current iteration for future iterations.

Consider an iteration. Given two doors d_i and d_j that we have processed, we maintain the fastest partial route information from d_i to d_j in a global hashmap H_{fpr}, to avoid re-computation in the future iterations. In particular, when we find the fastest route from d_i and d_j (line 14), we check whether $key = (d_i, d_j)$ exists in H_{fpr}. If so, we can append the saved route to the current route directly. Otherwise, we proceed to search for the route. Once such a route is found, it is inserted into H_{fpr}.

Moreover, we maintain another global hashmap H_{in} to store those (partial) routes that are found to be infeasible. To illustrate, consider an example with $S = \{v_1, v_2, v_3\}$. If we found that there does not exist a feasible (partial) route R that contains $\langle p_s, v_1, v_2 \rangle$ when we process S, we add $key = \langle v_1, v_2 \rangle$ into H_{in}. Then, when we find the feasible route for another set $S' = \{v_1, v_2, v_4\}$, we do not need to consider R that contains $\langle p_s, v_1, v_2 \rangle$ since it must also be infeasible.

Formally, we perform a checking when we search for the feasible route for a set S as follows. If any route R_{in} is in H_{in}, we know that R_{in} is infeasible and return $\emptyset$ immediately. Note that H_{in} can be updated accordingly when we check the feasibility of new partial routes to a remaining key partition. If no new partial route satisfies the time constraint checking, R_{in} is inserted into H_{in}.

4.3 Time Complexity

The time complexity of SSA is dominated by the key partition sets processing part (lines 5 to 21 in Algorithm 1). Let $|KPS|$ be the number of key partition sets processed in SSA and θ be the time complexity of executing one iteration. The time complexity of SSA is $O(|KPS| \cdot \theta)$. In practice, we have $|KPS| << 2^{|CKP|}$ since it utilizes the pruning techniques.

Consider θ. It is dominated by the time cost of executing the $findFeasibleRoute()$ method (i.e., Algorithm 2). Let d_{max} be the maximum number of doors a partition has, and m be the cost for computing the fastest route from a door to another. The time cost is $O(|S|! \cdot |S| \cdot d_{max}^2 \cdot m)$, where $|S| = |QW|$, since the number of possible permutations of S is $O(|S|!)$, and the time complexity of computing a complete route for one permutation is $O(|S| \cdot d_{max}^2 \cdot m)$. In summary, the time complexity of SSA is $O(|KPS| \cdot |S|! \cdot |S| \cdot d_{max}^2 \cdot m)$.

The time complexity of SSA is exponential with the number of query keywords, i.e., $O(2^{|QW|})$. This is because the algorithm needs to iterate different partition sets, where each set size is at most $|QW|$. It also has a $O(|QW|!)$ component, which is because it needs to iterate different visiting order of the partitions in the partition sets to form a route. Note that in practice the running time should be much faster since our cross-iteration computation strategy can help to reduce the computation needed.

5 Variants of TIKRQ

In practice, the users might have some special requirements when specifying a query. To support such requirements, we propose two variants of TIKRQ. In this section, we also briefly discuss how to handle the variants.

5.1 Preferred Visiting Order

One may want to specify a preferred visiting order of the shops. In this case, the query keywords are *ordered*. For example, the users might want to first buy a coffee to drink along the route. Formally, given a list of ordered query keywords $QW = \langle w_1, w_2, \ldots, w_{|QW|} \rangle$, each returned route $R \in TopKRoutes$ visits the key partitions according to the ordering of QW.

To handle this variant, we modify our algorithm as follows. In Step 3 of SSA (Algorithm 1), instead of using an arbitrary key partition ordering, we follow the visiting order to find the feasible route. Let $v_i' \in S$ denote the key

partition corresponding to the query keywords w_i. We amend the loop in line 12 of Algorithm 2 as follows. Given a key partition set S and a partial route R that has visited m key partitions in S, where $0 \leq m < |QW|$, we only need to expand R to the $(m+1)^{th}$ key partition that covers the query keyword w_{i+1} next to w_i in the specified visiting order in QW. The other steps remain the same.

5.2 Absence of A Target Point

A user may not have a dedicated target point when issuing a routing query. For example, a user may want to buy something without a clear idea of where to stop in a shopping mall. This can be modelled by having $p_t = \emptyset$ in the original TIKRQ. In this case, each resultant route $R \in TopKRoutes$ ends at the last visited key partition instead of a well-defined p_t. We can extend our algorithms to handle this variant easily by making the following two changes. First, we define $\delta(p_s, p_t) = 0$ and $\gamma(d_y, p_t)_L = 0$. Second, we skip lines 8 and 9 in Algorithm 2.

The two variants above are orthogonal to each other, and thus they can be applied at the same time. It is easy to see that neither variant affects the time complexity of SSA (Algorithm 1).

6 Session-based TIKRQ

The TIKRQ and its varations discussed in the previous sections are *one-off*. They do not consider the user's status after returning the resultant routes. In this section, we extend the TIKRQ into a *session-based* scenario, which keeps track of the user's routing and refines the ongoing route when needed.

A session is started when the user issues a new TIKRQ, and continues until the user explicitly terminates it (e.g., when the user finishes the route). During the session, the user follows the route from p_s to p_t. In some situations, as the parameters change during the session, the user might want to *update* the query parameters and obtain the corresponding updated result routes. For example, the time constraint Δ_{Max} could be relaxed if the flight is delayed and the boarding time is postponed. Note that this update can be triggered multiple times during a session. We formalize the session-based TIKRQ problem as follows.

Problem 2 (Session-based TIKRQ)**.** *Let a session $s = (q, TopKRoutes)$, where q is a TIKRQ and TopKRoutes is the set of result routes. Let Θ_c specify the parameters changed in q (i.e., p_s, p_t, QW, Δ_{Max} or k) and the corresponding new values. Given s and Θ_c, the session-based TIKRQ problem is to find the result routes satisfying Θ_c and update them to TopKRoutes.*

While the same query q and the corresponding results *TopKRoutes* are alive during a session, for ease of illustration, we refer the updated query and results as q' and *TopKRoutes*$'$, respectively, where q' is identical to q except

for the changed parameter values in Θ_c. A naive way to answer q' is to issue a new TIKRQ and compute the result routes from scratch. However, in some cases, we can handle the changes efficiently by utilizing the *TopKRoutes* and the intermediate results (e.g., *CKP*) computed when processing the original query. We focus on updating one parameter each time, but the techniques can be easily extended to handle the case that multiple parameters are amended in one update.

In the following, we present an algorithm ResultUpdate to answer the session-based TIKRQ, which involves two phases, namely a reusing phase and a searching phase. The reusing phase checks each route $R \in TopKRoutes$ in the results of q, and decides whether or not it is still feasible for the new query q'. If so, R is included in the initial results of $TopKRoutes'$. The searching phase searches for the new routes that are possibly in the top-k results of q'.

The implementations of the two phases depend on the type of parameter changes. Given a session $s = (q, TopKRoutes)$, we list and analyze all possible changes in s, as shown in Table 3. Depending on the potential reuse of the routes in *TopKRoutes* and the intermediate results, these changes can be categorized into 4 groups. Group 1 enjoys the highest degree of result reuse while group 4 is the lowest.

Table 3: Analysis on Parameter Changes

Group	Case	Θ_c	Analysis on Potential Reuse of $R \in TopKRoutes$
1	a	$\Delta_{Max} \leftarrow \Delta'_{Max}$ where $\Delta'_{Max} > \Delta_{Max}$	**Direct Reuse:** It is easy to see that R is still feasible. We can simply initialize $TopKRoutes'$ by *TopKRoutes*.
	b	$k \leftarrow k'$ where $k' > k$	
2	c	$\Delta_{Max} \leftarrow \Delta'_{Max}$ where $\Delta'_{Max} < \Delta_{Max}$	**Check and Reuse:** If R is still feasible, it can be proven that R is one of the top-k minimum cost routes for q', and thus can be put in $TopKRoutes'$. Otherwise, we can reuse the intermediate result to find a feasible route for $KPS(R)$.
3	d	$p_s \leftarrow p'_s$	**Amend, Check and Reuse:** We amend R according to Θ_c. If it is feasible, we include it in the initial $TopKRoutes'$. Otherwise, we can reuse the intermediate result to find a feasible route for $KPS(R)$.
	e	$p_t \leftarrow p'_t$	
	f	$QW' \leftarrow QW \setminus \{w\}$	
4	g	$QW' \leftarrow QW \cup \{w\}$	**Limited Reuse:** It is likely that R is not in the top-k minimum cost routes for q', since there are additional key partitions that contribute to w. We can reuse the *CKP* for QW to construct CKP' (for QW'), and find the new routes.

As the handling of cases in groups 1 and 4 are trivial, we discuss the details on groups 2 and 3. In the following, we present how to handle the cases in groups 2 and 3 in Section 6.1 and 6.2, respectively.

6.1 ResultUpdate in Group 2

Algorithm 3 ResultUpdateCaseC (q, $TopKRoutes$, Δ'_{Max})

```
1: TopKRoutes' ← ∅
2: for each R ∈ TopKRoutes do                                  ▷ Reusing Phase
3:     if δ(R) < Δ'_Max then
4:         add R to TopKRoutes'
5:     else
6:         R' ← findFeasibleRoute(KPS(R), p_s, p_t, Δ'_Max)
7:         if R' ≠ ∅ then
8:             add R to TopKRoutes'
9:         end if
10:    end if
11: end for
12: if |TopKRoutes'| == k then return TopKRoutes'
13: waitTime'_max ← Δ'_Max − δ(p_s,p_t) / s_walk
14: curKCost ← 0
15: search(q, CKP, waitTime'_max, curKCost, TopKRoutes') ▷ Searching Phase
16: return TopKRoutes'
```

Algorithm 4 search(q, CKP, $waitTime'_{max}$, $curKCost$, $TopKRoutes'$)

```
1: for each remaining subset S of CKP do
2:     if cost_LB(S) > curKCost then continue
3:     if Σ_{v∈S} v.waitTime > waitTime_max then continue
4:     R ← findFeasibleRoute(S, p_s, p_t, Δ_Max)
5:     if R ≠ ∅ then
6:         update TopKRoutes' with R
7:         curKCost ← cost of the k-th route in TopKRoutes'
8:     end if
9: end for
```

Reusing Phase. Consider a route $R \in TopKRoutes$. We prove that R is one of the top-k minimum cost routes if it is still feasible under the new time budget $\Delta'_{Max} < \Delta_{Max}$ by the following lemma.

Lemma 4. *Let $TopKRoutes'$ be the results of q'. If $\delta(R) < \Delta'_{Max}$ for a route $R \in TopKRoutes$, where $\Delta'_{Max} < \Delta_{Max}$, we have $R \in TopKRoutes'$.*

Proof We prove this by contradiction. Suppose $R \notin TopKRoutes'$, then there must exist at least k routes that have a smaller cost than R for q'. Since $\Delta'_{Max} < \Delta_{Max}$, these k routes must also be feasible and have a smaller cost than R for q, which contradicts the fact that R is in $TopKRoutes$. □

Based on this lemma, we can keep the routes in $TopKRoutes$ if they are feasible. If all routes are feasible, we do not need to search for any new routes.

Searching Phase. In this phase, we first calculate the updated $waitTime_{max}$ based on the new Δ'_{Max}. Subsequently, We search for the feasible routes from the remaining subset S of CKP until the top-k routes are found.

The ResultUpdate algorithm for group 2 is shown in Algorithm 3. Specifically, it checks, for each route $R \in TopKRoutes$ in the current result, whether R is still feasible (i.e., $\delta(R) < \Delta'_{Max}$). If so, it updates $TopKRoutes'$ by R (lines 2–3). Otherwise, it searches for the feasible route R' with the same key partition set as R. If such a route R' exists, it updates $TopKRoutes'$ by R'. (lines 6–8). If the $TopKRoutes'$ is of size k, $TopKRoutes'$ is deemed to be optimal, and thus is returned as the result. (line 12). Otherwise, the algorithm updates $waitTime'_{max}$, and invokes the search() method as the searching phase.

The search method is shown in Algorithm 4. It iterates the remaining key partition set S of CKP that has not yet been processed by SSA when answering q, which can be known by recording the position of the corresponding pointers when processing q. The iteration is similar to the way that SSA finds the result routes. The algorithm terminates when all sets have been processed. Finally, $TopKRoutes'$ is returned as the result.

6.2 ResultUpdate in Group 3

In the following, we discuss the details of cases (d), (e) and (f).

6.2.1 Case (d)

Reusing Phase. For each route $R \in TopKRoutes$, we can reuse its $KPS(R)$ to find a feasible route R'. After each route is processed, we initialize $TopKRoutes'$ by all these feasible routes.

Searching Phase. In fact, the searching phase is identical to that in case (c) except that we do not need to update $waitTime_{max}$. Thus, the search() method can also be used for case (d).

Algorithm 5 shows the algorithm for result updating for case (d). Specifically, it maintains a list $TopKRoutes'$ storing the k best-known routes found so far. Then, the reusing phase is carried out as follows. It iterates through each route $R \in TopKRoutes$, and find if a feasible route with the same key partition set $KPS(R)$ exists under the new source point p'_s. If so, it then updates $TopKRoutes'$ by R'. After all routes in $TopKRoutes$ have been processed, it proceeds to the searching phase by invoking search() method with the new source point p'_s. Finally, the $TopKRoutes'$ is returned as the result.

6.2.2 Case (e)

The method of handling case (e) is exactly the same as case (d) by replacing the new source point p'_s to a new target point p'_t. Thus, we can simply reuse Algorithm 5 by modifying line 3 to $findFeasibleRoute(KPS(R), p_s, p'_t, \Delta_{Max})$ and line 9 to $q.p_t \leftarrow p'_t$.

Algorithm 5 ResultUpdateCaseD (q, CKP, $TopKRoutes$, p'_s)

1: $TopKRoutes' \leftarrow \emptyset$
2: **for** each $R \in TopKRoutes$ **do** ▷ Reusing Phase
3: $R' \leftarrow findFeasibleRoute(KPS(R), p'_s, p_t, \Delta_{Max})$
4: **if** $R' \neq \emptyset$ **then**
5: add R' to $TopKRoutes'$
6: **end if**
7: **end for**
8: $curKCost \leftarrow$ cost of the k-th route in $TopKRoutes'$
9: $q.p_s \leftarrow p'_s$
10: $search(q, CKP, curKCost, TopKRoutes')$ ▷ Searching Phase
11: **return** $TopKRoutes'$

6.2.3 Case (f)

Reusing Phase. For each route $R \in TopKRoutes$, we remove the key partitions that are only relevant to w from R, denoted by R'. It is easy to see that R' is feasible. After each route is processed, we initialize $TopKRoutes'$ by using all these amended routes.

Searching Phase. In fact, the searching phase is identical to that in case (c) except that we do not need to update $waitTime_{max}$. Thus, the search() method can also be used for case (f).

Algorithm 6 shows the algorithm for result updating. Specifically, it maintains a list $TopKRoutes'$ storing the k best-known routes found so far. Then, it refines the set of candidate key partition by removing the keyword w (lines 1–2). Next, it performs the reusing phase as follows. It iterates through each route $R \in TopKRoutes$, and removes the key partitions that are in $\widetilde{\mathcal{CP}}(w)$ from R. It then updates $TopKRoutes'$ by R. After all routes in $TopKRoutes$ have been processed, it proceeds to the searching phase by invoking search() method. Finally, the $TopKRoutes'$ is returned as the result.

Algorithm 6 ResultUpdateCaseF (q, CKP, $TopKRoutes$, w)

1: $TopKRoutes' \leftarrow \emptyset$
2: $\widetilde{\mathcal{CP}}(w) \leftarrow \mathcal{CP}(w) \setminus \cup_{w' \in QW'} \mathcal{CP}(w')$
3: **for** each $R \in TopKRoutes$ **do** ▷ Reusing Phase
4: $KPS(R) \leftarrow KPS(R) \setminus \cup_{v_i \in \widetilde{\mathcal{CP}}(w)} v_i$
5: add R to $TopKRoutes'$
6: **end for**
7: $curKCost \leftarrow$ cost of the k-th route in $TopKRoutes'$
8: $q.QW \leftarrow q.QW \setminus \{w\}$
9: $search(q, CKP, curKCost, TopKRoutes')$ ▷ Searching Phase
10: **return** $TopKRoutes'$

6.3 Time Complexity

Let d_{max} be the maximum number of doors of a partition, m be the cost for computing the fastest route from a door to another, and β be the time complexity of Algorithm 4. The time complexity of Algorithm 3 and Algorithm 5 are $O(k \cdot |S|! \cdot |S| \cdot d_{max}^2 \cdot m + \beta)$, which is because in the worse case they checks, for each route in *TopKRoutes*, whether a feasible route exists. This checking costs $O(|S|! \cdot |S| \cdot d_{max}^2 \cdot m + \beta)$, as analyzed in Section 4.3.

The time complexity of Algorithm 6 is $O(k \cdot |QW| \cdot |\widetilde{\mathcal{CP}}(w)| + \beta)$, since it updates each route R in *TopKRoutes* by removing the partitions in $\widetilde{\mathcal{CP}}(w)$.

Next we analyze β, it is easy to see that the time complexity of the Algorithm 4 is same as that of SSA, which is $O(|KPS| \cdot |S|! \cdot |S| \cdot d_{max}^2 \cdot m)$. Thus, combining the above, the time complexities of Algorithm 3, Algorithm 5 and Algorithm 6 are dominated by Algorithm 4.

7 Empirical Studies

In this section, we evaluate our proposed algorithms. The experiments on synthetic and real dataset are presented in Sections 7.1 and 7.2, respectively. For easy reference, we use Table 4 to show the roadmap of this section.

Table 4: Organization of Experiment Results

		Synthetic Dataset	**Real Dataset**
Set-up		Section 7.1.1	Section 7.2.1
TIKRQ		Section 7.1.2 Section 7.1.3	Section 7.2.2
Variants	1. Preferred Visiting Order	Section 7.1.4	Section 7.2.3
	2. Absence of Target Point	Section 7.1.5	Section 7.2.4
Session-based TIKRQ	Case (c): Reduced Time Budget	Section 7.1.6	Section 7.2.5
	Case (d): Changed source point	Section 7.1.7	Section 7.2.6
	Case (e): Changed target point	Section 7.1.8	Section 7.2.7
	Case (f): Keyword Removal	Section 7.1.9	Section 7.2.8

7.1 Experiment on Synthetic Dataset

7.1.1 Set-up

Indoor Space. Following [12], we generate a n-floor building based on a real world floor plan[5], where $n = \{3, 5, 7, 9\}$. Each floor is 1368m × 1368m, consists of 96 rooms, 4 hallways, and 4 staircases. We obtain 141 partitions and 200 doors on each floor by decomposing those irregular hallways into smaller and regular partitions. To model the *elevators* in an indoor space, we convert two staircases to elevators that connect to all floors, rather than the adjacent floors only. Each elevator has a waiting time of 30 seconds and takes 10 seconds to

[5] https://longaspire.github.io/s/fp.html

traverse from one floor to another. The remaining two staircases connect two adjacent floors, each being 20m long. By default, we set $n = 5$ and the indoor space contains 705 partitions and 1100 doors.

Partition Keywords. We assign keywords to each room as follows. First, we crawl the shop information of five shopping malls in Hong Kong[6] online using Scrapy. We obtain 2074 documents for 1225 shop brands. All the 1225 brand names are used as i-words. Second, we manually categorize these brand names into 11 categories, following the categorization used in the shopping malls (e.g., *clothing, cosmetics* and *restaurant*). These categories are used as the c-words. Each category contains 111 i-words on average. Third, we feed these i-words into the RAKE algorithm [32] to extract keywords from the corresponding documents. Only 1120 i-words yield extracted keywords. For each such i-word, we use up to 60 extracted keywords with the highest TF-IDF values. In total, we have 9195 extracted keywords and each i-word corresponds to an average of 16.6 extracted keywords. For the partitions, their static cost and waiting time are picked uniformly at random, in the range $[1, 10]$ and $[0, 100]$, respectively.

Queries. We generate the query keywords as follows. The number of query keywords $|QW|$ is in the range $[1, 5]$, as over 95% of web search queries have at most 5 keywords[7]. We vary the fractions of c-words/i-words/t-words in QW, as the parameter $c/i/t$. The procedure is to vary the fraction of one type with the other two types being changed accordingly. Take the c-word as an example, we vary its fraction from $p = [0.1, 0.9]$, and the fractions of i-words and t-words are both set to be $(1 - p)/2$. In addition, we vary the parameters α in Equation 1. Table 5 summaries the parameters setting with default values in bold.

Table 5: Parameter Settings

Parameters	Settings
k	1, 3, 5, **7**, 9, 11
$\|QW\|$	1, 2, 3, **4**, 5
Δ_{Max}	3000, **3500**, 4000, 4500, 5000 (seconds)
$c/i/t$	0.1/0.45/0.45, **0.2/0.4/0.4**, ..., 0.05/0.05/0.9
n	3, **5**, 7, 9
α	0.1, 0.3, **0.5**, 0.7, 0.9

Algorithms. We compare our SSA algorithm with a baseline algorithm $SSA\backslash P$. $SSA\backslash P$ follows the workflow of SSA, but the proposed pruning features and computation strategy in SSA are removed. Also, we adapt the algorithm KoE [12], which is originally designed for IKRQ. The adaption is as follows. It expands the partial routes from p_s to search one of the key partitions that can cover some of those uncovered query keywords, until all keywords are covered, and finally connects to p_t. For each complete route, it calculates the cost of the routes and maintains the top-k feasible routes. The

[6] https://longaspire.github.io/s/hkdata.html
[7] http://www.keyworddiscovery.com/keyword-stats.html

route-based speed-up techniques (i.e., Pruning 3 and Lemma 3) and cross-iteration computation strategy are also employed in the adaption to allow a fair comparison.

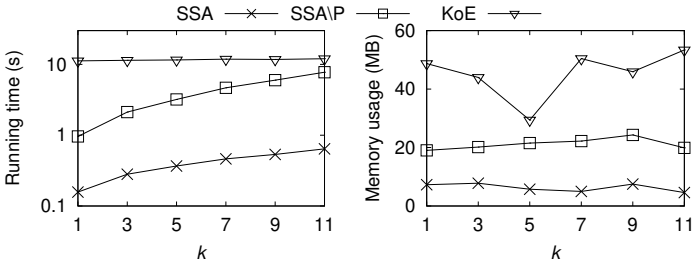
For the Session-based TIKRQ proposed in Section 6, we compare the performance of the four ResultUpdate (RU) algorithms and SSA, where SSA finds the result routes without using the results of the previous computations.

All algorithms are implemented in Java and run on a Mac with a 2GHz Quad-Core Intel i5 CPU and 16GB memory. The code and data are available in GitHub repository[8].

Performance Metrics. We measure the running time and the memory consumption. For each experimental setting, we generate 10 queries and report the average results. Note that the results are based on the queries with routes returned only. In case of no route is returned as the result, we simply re-generate a new query.

7.1.2 Efficiency Studies

Effect of k. Figure 4 shows the results of varying k. According to Figure 4(a), the running times of SSA and $SSA\backslash P$ increase slightly when k increases. This is because a larger k incurs more routes to be explored. KoE is insensitive to k while it always requires significantly long time to terminate. In general, our SSA runs faster than $SSA\backslash P$ and KoE by an order of magnitude, as contributed by the pruning techniques and computation strategy employed. According to Figure 4(b), the memory usages of all algorithms fluctuate with a varying k. SSA still consumes less than 10MB of memory in different k values, much less than its competitors.



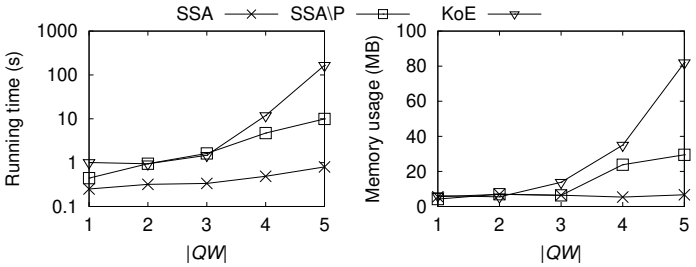
(a) Running time (b) Memory usage

Fig. 4: Effect of k

Effect of Query Size $|QW|$. Figure 5 shows the results of varying the number of query keywords from 1 to 5. According to Figure 5(a), the running times of all algorithms increase with an increasing $|QW|$. A larger $|QW|$ leads to more relevant partitions and thus more key partition sets need to be formed and considered. Moreover, each key partition set would be larger and therefore it takes more time to find the complete route for the set in each iteration. Our SSA runs consistently faster than $SSA\backslash P$ and KoE, and the gap enlarges

[8] https://github.com/harryckh/TIKRQ

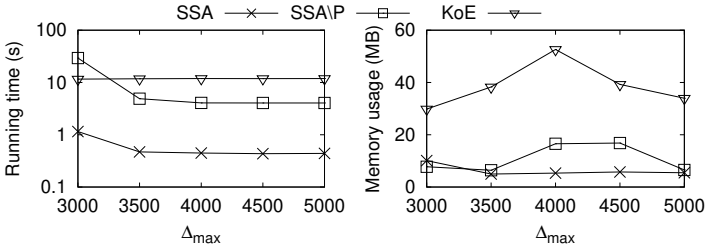
when $|QW|$ increases. This is because our pruning strategies are more effective when $|QW|$ is larger. According to Figure 5(b), the memory usages of all algorithms are similar and increase steadily with $|QW|$. However, SSA and $SSA\backslash P$ grow slower than KoE. Even with more pruning strategies employed, SSA consumes fewer memories than $SSA\backslash P$ and KoE for $|QW|$. This is due to the use of the cross-iteration computation strategy.



(a) Running time (b) Memory usage

Fig. 5: Effect of $|QW|$

Effect of Time Constraint Δ_{Max}. Figure 6 reports the results of varying Δ_{Max}. According to Figure 6(a), the running times of both SSA and $SSA\backslash P$ decreases when Δ_{Max} increases and our SSA always outperforms $SSA\backslash P$ and KoE. Note that a looser Δ_{Max} reduces the difficulty of finding the feasible routes and results in fewer key partition sets to explore. In this sense, the effectiveness of Pruning Rules 2 and 3 is amplified in a setting of a larger Δ_{Max}. Referring to Figure 6(b), the memory usage of SSA decreases when Δ_{Max} increases, since fewer paths and infeasible sets need to be stored when a larger Δ_{Max} is set. On the other hand, both $SSA\backslash P$ and KoE are insensitive to Δ_{Max}, and incur higher memory usages than SSA.

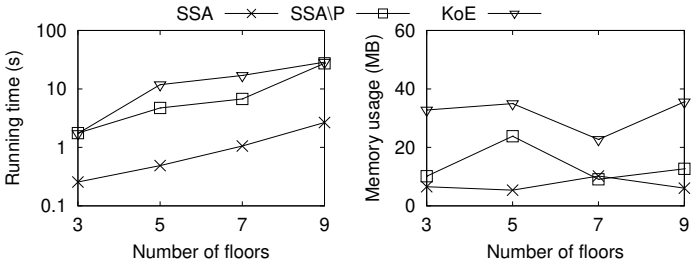


(a) Running time (b) Memory usage

Fig. 6: Effect of Δ_{Max}

Effect of Number of Floors n. To evaluate the scalability of our algorithm, we vary n in $\{3, 5, 7, 9\}$ and report the result in Figure 7. According to Figure 7(a), the running times of all algorithms increase with n increases. A higher n means a larger number of partitions and thus more relevant partitions need to be checked. Particularly, the elevators in our setting allow the route to pass different floors easily. Thus, the time constraint can barely help reducing the search space. Nevertheless, SSA runs consistently faster than its

competitors and it can finish within 3 seconds when n grows up to 9, showing its scalability.



(a) Running time (b) Memory usage

Fig. 7: Effect of n

Effect of α. Figure 8 reports the running time of the algorithms when varying α. Our SSA runs faster than $SSA\backslash P$ and KoE by orders of magnitude. Also, the running time of KoE decreases when α increases. For the route cost given in Definition 1, having a larger α puts less weight on routes' textual relevance, which is easier for a graph-based algorithm like KoE to find routes with smaller cost. The memory usages of all algorithms are insensitive to α, and SSA uses the least memory among all three.

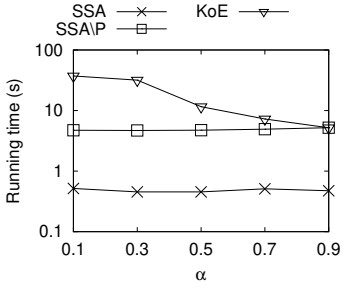


Fig. 8: Effect of α

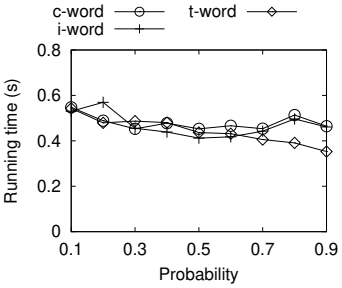


Fig. 9: Effect of $c/i/t$

Effect of Fraction of c/i/t-words. We vary the parameter p of each type of word (cf. Section 7.1.1) and report the result for SSA in Figure 9. In general, there is not much difference in the running time of different combinations of keywords.

7.1.3 Effectiveness Studies

Case Study. To show that TIKRQ is able to return desirable routes in practice, we perform a case study by comparing the TIKRQ result with that of minimizing the route's time cost. The query keywords are *apple* (an i-word) and *coffee* (a t-word), $\Delta_{Max} = 3600$ seconds (one hour), $k = 3$. We use $\alpha = 0.8$ to reflect and suit the needs of keyword-awareness in shopping. The routes returned by TIKRQ are listed in Table 6. The top-1 route (say R) has a total route cost $Cost(R) = 0.125$ and time cost $\gamma(R) = 2474.545$ ($\approx$ 41 minutes).

In particular, R has a textual relevance of 1, as it passes the partitions *apple* and *coffee day* (which sells coffee). In contrast, the route R' with the minimum time cost (i.e., $\gamma(R') = 1596.807 \approx 27$ minutes) incurs an overall route cost of 0.450. R' only has a textual relevance of 0.55, as it does not pass *apple* but another partition with category *electronics*. Although R has a longer time needed, its textual relevance and route cost meet the practical user needs. This demonstrates that our returned paths can better serve the users in the context of keyword awareness.

Table 6: Case Study

	k	Textual Relevance	Route Cost	Time Cost
TIKRQ returned path	1	1	0.05	2474.545
	2	1	0.08	1891.023
	3	1	0.13	2540.905
minimum time cost path	R'	0.55	0.45	1596.807

Effect of Unique Partition Set. To compare the results with and without adopting the concept unique partition set, we run $SSA\backslash UPS$ which removed the requirement of unique partition set. In other words, it allows different routes in the k resulting routes to have an identical key partition set.

We measure the identical rate as the fraction of routes with the identical key partition set with others. We ran 10 queries for each k and Figure 10 reports the average rate. It shows that the identical rate of $SSA\backslash UPS$ increases rapidly when k increases. More than 60% of the returned routes have identical key partition sets when $k \geq 5$. Such routes are not interesting to users and hinder the diversity of the results. This verifies that the unique partition set offers users more diversified combinations of partitions in the result.

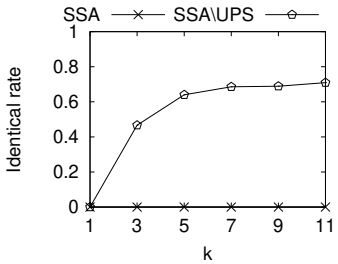


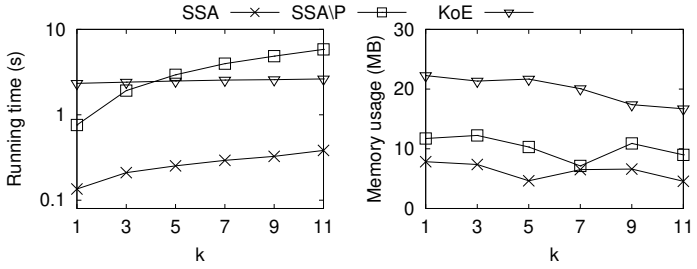
Fig. 10: Identical Rate

7.1.4 Variant 1: Preferred Visiting Order

We randomly generate a visiting order for the query keywords.

Effect of k. Figure 11 shows the results of varying k. According to Figure 11(a), the running times of both SSA and $SSA\backslash P$ increase when k

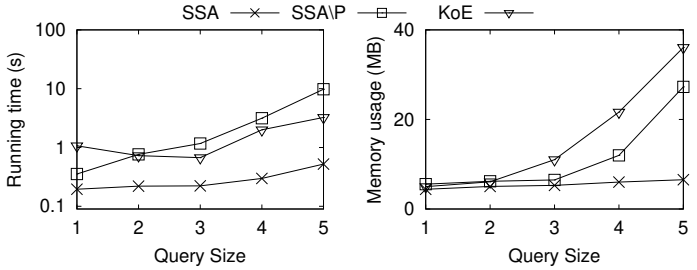
increases. While KoE is insensitive to k, it runs much slower than SSA. Similar to the case without a visiting order (Figure 4), our SSA runs faster than $SSA\backslash P$ and KoE by an order of magnitude. Moreover, all algorithms are faster than the counterparts without a visiting order. This is because ordering keywords reduces the search space for finding the feasible routes. According to Figure 4(b), the varied value of k does not cause big changes of the memory usages of the algorithms, and SSA always consumes less than 10MB of memory, clearly better than its competitors.



(a) Running time (b) Memory usage

Fig. 11: Effect of k (Variant 1: Preferred Visiting Order)

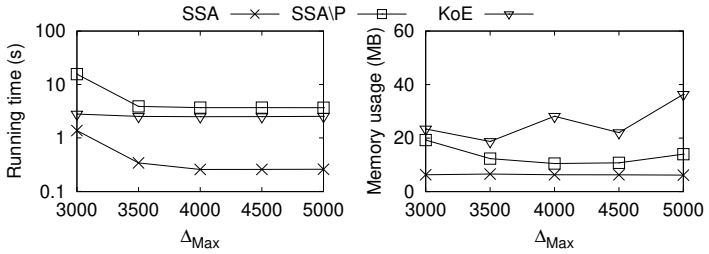
Effect of Query Size $|QW|$. Figure 12 shows the results of varying $|QW|$. According to Figure 12(a), SSA runs consistently faster than $SSA\backslash P$ and KoE, as the set-based search strategy and the pruning techniques in SSA prune the unqualified key partition sets and infeasible routes effectively. According to Figure 12(b), the memory usages of $SSA\backslash P$ and KoE increase steadily with $|QW|$. SSA always outperforms its competitors, using less than 10MB of memory. This is due to the reduced search space in SSA.



(a) Running time (b) Memory usage

Fig. 12: Effect of $|QW|$ (Variant 1: Preferred Visiting Order)

Effect of Δ_{Max}. Figure 13 shows the results of varying Δ_{Max}. According to Figure 13(a), the running times of SSA and $SSA\backslash P$ decrease when Δ_{Max} increases, and SSA always outperforms the competitors, similar to what are seen in Figure 6. This is because a larger Δ_{Max} reduces the difficulty of finding the feasible routes, and thus the number of key partition sets need to be explored is decreased. According to Figure 13(b), the memory usage of SSA is always the least, using less than 10MB of memory, since it has the smallest search space and thus fewer paths and infeasible sets need to be stored.

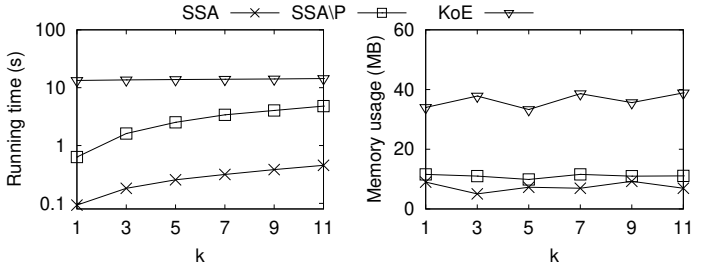


(a) Running time (b) Memory usage

Fig. 13: Effect of Δ_{Max} (Variant 1: Preferred Visiting Order)

7.1.5 Variant 2: Absence of Target Point

Effect of k**.** Figure 14 shows the results of varying k. According to Figure 14(a), the running times of SSA and $SSA\backslash P$ increase with an increasing k. The running time of KoE is insensitive to k, but is always larger than 10 seconds. Our SSA runs faster than $SSA\backslash P$ and KoE by at least an order of magnitude, as the effectiveness of the pruning techniques and computation strategies are not affected by the absence of a target point. According to Figure 14(b), the memory usages of all algorithms are insensitive to the value of k, and SSA always consumes the least memory.

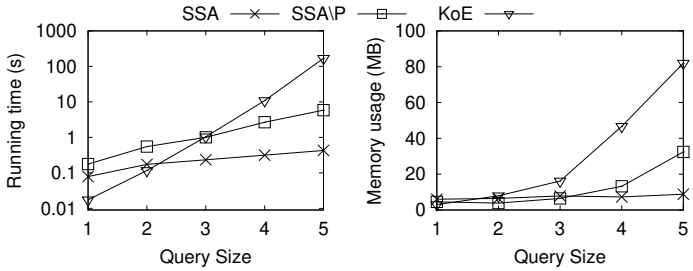


(a) Running time (b) Memory usage

Fig. 14: Effect of k (Variant 2: Absence of Target Point)

Effect of Query Size $|QW|$**.** Figure 15 shows the results of varying $|QW|$. According to Figure 15(a), the running times of all algorithms increase with query size, while that of KoE increases more rapidly. SSA runs faster than KoE when $|QW|$ is large (e.g., $|QW| \geq 3$). When $|QW| = 5$, KoE takes more than 100 seconds to run, while our SSA is more scalable to $|QW|$ and can finish within 1 second. It is because our set-based search strategy in SSA is more robust to the large query size, while pruning rules in KoE rely the target point heavily to reduce the search space. On the other hand, in the case that query size is small, the solution is usually trivial and easy to find, especially when the routes do not need to reach the target point. In this case, the time overhead of the prunings in SSA outweighs the performance gain, and thus SSA is slower than KoE. Still, SSA can terminate in real time, e.g., within

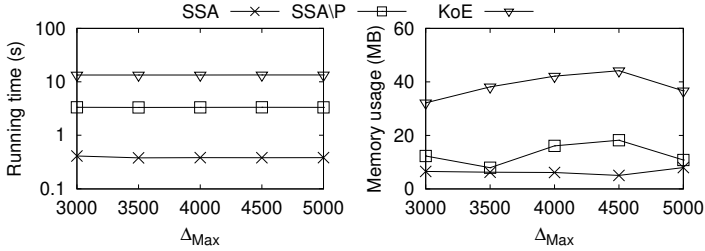
0.2 seconds. According to Figure 15(b), the memory usage of SSA is always less than 10MB, the least among all algorithms.



(a) Running time (b) Memory usage

Fig. 15: Effect of $|QW|$ (Variant 2: Absence of Target Point)

Effect of Δ_{Max}. Figure 16 shows the results of varying Δ_{Max}. According to Figure 16(a), the running times of all algorithms are insensitive to Δ_{Max}. This is because the result routes do not need to reach the target points, and thus it is easier to find the feasible routes. Moreover, SSA always performs the best, as contributed by the set-based search strategy and pruning techniques employed. According to Figure 16(b), the memory usages of the algorithms are insensitive to Δ_{Max}, and SSA is the least among the competitors, since it has a small search space to explore.



(a) Running time (b) Memory usage

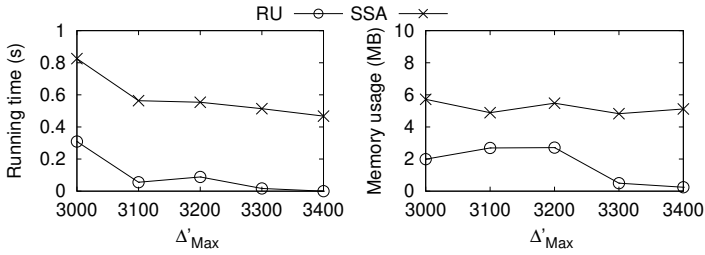
Fig. 16: Effect of Δ_{Max} (Variant 2: Absence of Target Point)

7.1.6 Session-based TIKRQ: Reduced Time Budget (Case (c))

We set $\Delta'_{Max} = \Delta_{Max} - 500$ seconds. Thus $\Delta'_{Max} = 3000$ seconds is used as the default value.

Effect of Δ'_{Max}. Figure 17 shows the results of varying Δ'_{Max}, while keeping $\Delta_{Max} = 3500$. According to Figure 17(a), the running times of both RU and SSA decrease when Δ'_{Max} increases. Moreover, RU runs much faster than SSA, which shows the advantage of reusing the results. In particular, the running times of RU are close to 0 when $\Delta_{Max} \geq 3300$. When the time budget change is small, more result routes can be reused, and thus lowering the total computation time. Besides, RU runs slower when $\Delta'_{Max} = 3200$ than $\Delta'_{Max} = 3100$. It is probably because when $\Delta'_{Max} = 3200$, it takes a longer time to find

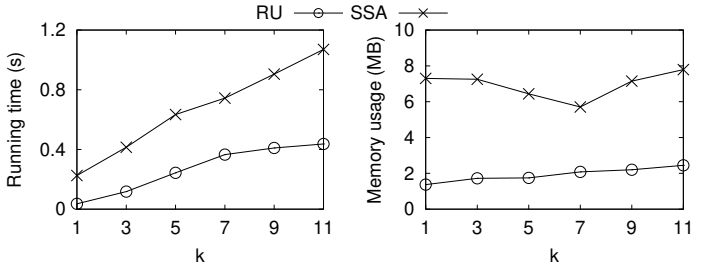
the feasible route of $KPS(R)$ for each route R in $TopKRoutes$. When $\Delta'_{Max} = 3100$, it is easier to verify the infeasibilities of the routes in $TopKRoutes$ (as the time budget is smaller), and thus it proceeds to find routes from other key partition sets quickly. According to Figure 17(b), both algorithms consume less than 10MB memory, and RU uses less memory than SSA.



(a) Running time (b) Memory usage

Fig. 17: Effect of Δ'_{Max} (Case (c))

Effect of k. Figure 18 shows the results of varying k. According to Figure 18(a), the running times of both algorithms increase when k increases. RU runs faster than SSA, and the difference become larger when k enlarges. It is because when k increases, the number of original results also increases, which helps to reduce the search space in RU. According to Figure 18(b), the memory usage of RU is much smaller than that of SSA, since the search space is smaller in RU.

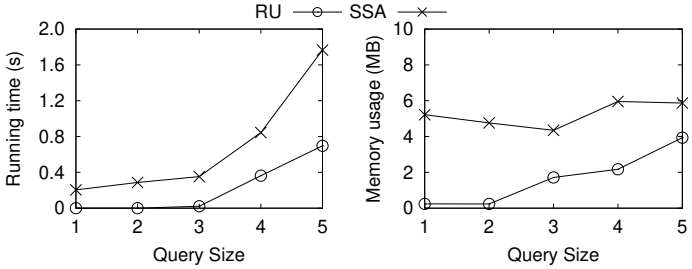


(a) Running time (b) Memory usage

Fig. 18: Effect of k (Case (c))

Effect of Query Size $|QW|$. Figure 19 shows the results of varying $|QW|$. According to Figure 19(a), the running times of both algorithms increase with query size, since more key partitions need to be considered and thus longer routes are formed. Moreover, RU runs consistently faster than SSA. It is because the result reuse phase saved the expensive computation in the search phase. According to Figure 19(b), the memory usage of RU increases with the query size, since more key partitions and longer routes need to be stored. Still, RU consumes less memory than SSA.

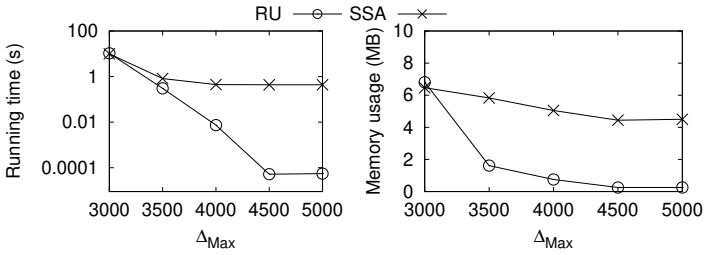
Effect of Δ_{Max}. Figure 20 shows the results of varying Δ_{Max}. According to Figure 20(a), the running times of both algorithms decrease when Δ_{Max} increases, as the difficulty of finding the feasible routes decreases. When Δ_{Max}



(a) Running time (b) Memory usage

Fig. 19: Effect of $|QW|$ (Case (c))

is large (e.g., $\Delta_{Max} \geq 4000$), *RU* can terminate very quickly (e.g., within 0.01s), since most of the results can be reused as the time budget is loose. According to Figure 20(b), the memory usages of both algorithms are small (less than 10MB), and *RU* has a lower memory consumption than *SSA*.



(a) Running time (b) Memory usage

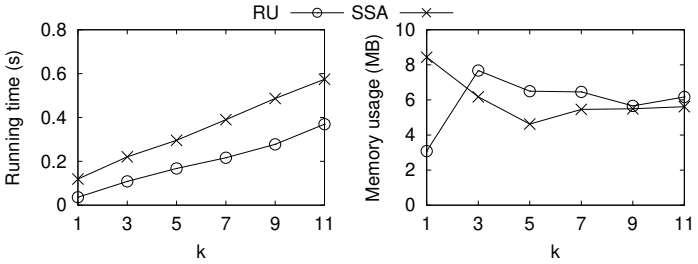
Fig. 20: Effect of Δ_{Max} (Case (c))

7.1.7 Session-based TIKRQ: Changed source point (Case (d))

Effect of k. Figure 21 shows the results of varying k. According to Figure 21(a), the running times of both algorithms increase with k, and *RU* runs slightly faster than *SSA*, which shows that our result reusing strategy can reduce the search space. According to Figure 21(b), the memory usages of both *RU* and *SSA* fluctuate with a varying k, but both algorithms always use less than 10MB of memory.

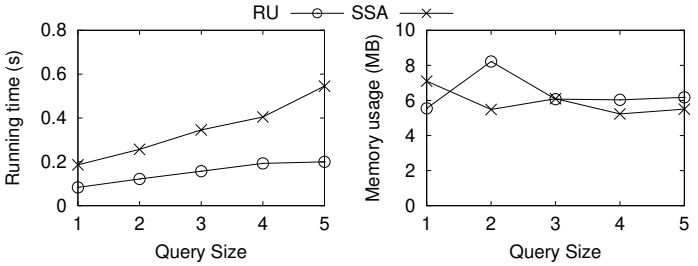
Effect of Query Size $|QW|$. Figure 22 shows the results of varying $|QW|$. According to Figure 22(a), the running times of both *RU* and *SSA* increase with $|QW|$, while *RU* runs consistently faster than *SSA* since the *RU* reuses some feasible routes. According to Figure 22(b), the memory usage of both algorithms are similar, and are less than 10MB.

Effect of Δ_{Max}. Figure 23 shows the results of varying Δ_{Max}. According to Figure 23(a), the running times of both algorithms decrease when Δ_{Max} increases, which is because the larger Δ_{Max} makes the algorithms easier to find feasible routes. Moreover, *RU* always runs faster than *SSA*. This again shows the computation time is reduced by the result reuse strategy. According



(a) Running time (b) Memory usage

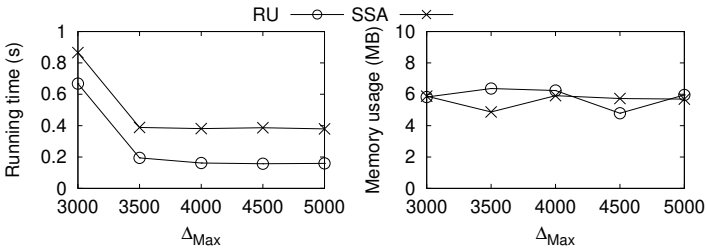
Fig. 21: Effect of k (Case (d))



(a) Running time (b) Memory usage

Fig. 22: Effect of $|QW|$ (Case (d))

to Figure 23(b), the memory usage of RU and SSA are similar, and both algorithms consume less than 10MB of memory in different Δ_{Max} values.



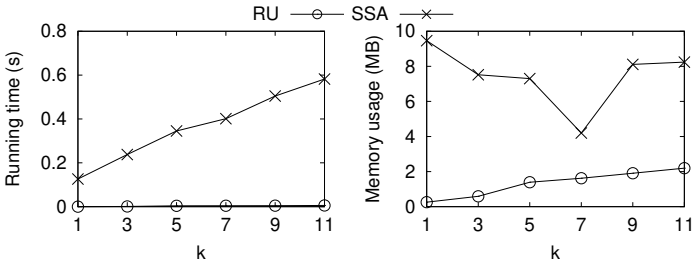
(a) Running time (b) Memory usage

Fig. 23: Effect of Δ_{Max} (Case (d))

7.1.8 Session-based TIKRQ: Changed target point (Case (e))

Effect of k. Figure 24 shows the results of varying k. According to Figure 24(a), the running time of SSA increases with k, while that of RU remains almost the same, and RU runs consistently faster than SSA. This shows the advantage of our result reuse strategy employed in RU. Comparing with Figure 21(a), we find that the running time of RU in case (d) is larger. The reason is that the algorithm finds feasible route starting from p_s, and

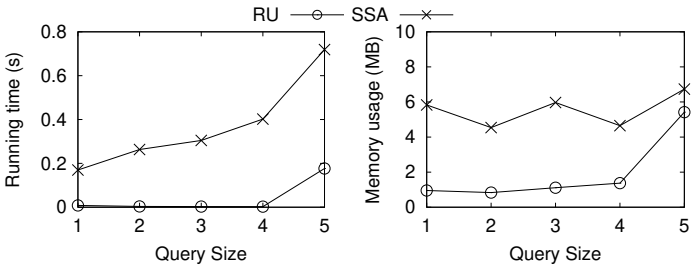
thus changing the source point leads to a smaller chance of information reuse, which thus incurs a higher running time. According to Figure 24(b), the memory usages of both algorithms fluctuate with a varying k, but are always less than 10MB of memory. RU consumes fewer memory than SSA, since RU has a smaller search space.



(a) Running time (b) Memory usage

Fig. 24: Effect of k (Case (e))

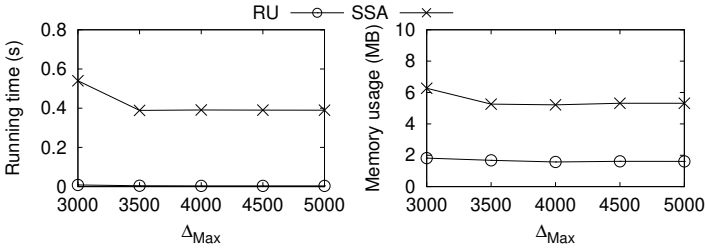
Effect of Query Size $|QW|$. Figure 25 shows the results of varying $|QW|$. According to Figure 25(a), the running times of both RU and SSA increase when $|QW|$ increases, while RU runs faster than SSA, since the result reuse phase can identify some feasible routes quickly. According to Figure 25(b), RU uses fewer memory than SSA, while both algorithms use smaller than 10MB in different query sizes. The increase in running time and memory usage of RU when $|QW| = 5$ is probably because the large number of keywords increased the difficulty to find feasible routes in the result reusing phase. Thus, more computation is needed to find new feasible routes in the searching phase.



(a) Running time (b) Memory usage

Fig. 25: Effect of $|QW|$ (Case (e))

Effect of Δ_{Max}. Figure 26 shows the results of varying Δ_{Max}. According to Figure 26(a), the running times of both algorithms decrease when Δ_{Max} increases, and RU always runs faster than SSA. This again shows the computation time is reduced by the result reuse strategy. According to Figure 26(b), the memory usage of RU is smaller than that of SSA, and both algorithms consume less than 10MB of memory in different Δ_{Max} values.



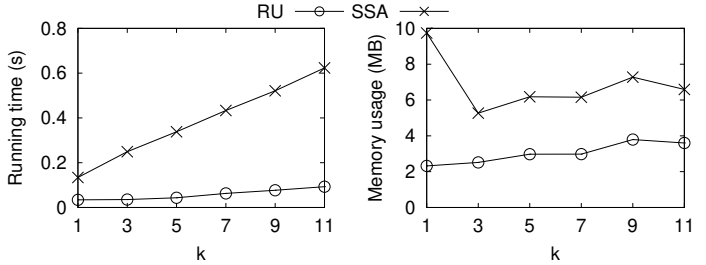
(a) Running time (b) Memory usage

Fig. 26: Effect of Δ_{Max} (Case (f))

7.1.9 Session-based TIKRQ: Keyword Removal (Case (f))

Given QW, we randomly remove a keyword in QW to obtain QW'.

Effect of k. Figure 27 shows the results of varying k. According to Figure 27(a), the running times of both algorithms increase with k, and RU runs consistently faster than SSA, since the result reuse strategy can greatly reduce the search space. According to Figure 27(b), the memory usages of both algorithms fluctuate with a varying k, but are always less than 10MB of memory. RU consumes fewer memory than SSA, since RU has a smaller search space.

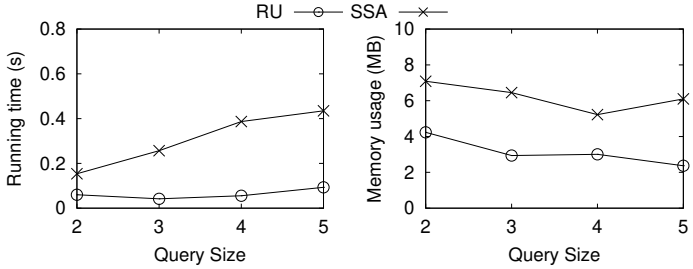


(a) Running time (b) Memory usage

Fig. 27: Effect of k (Case (f))

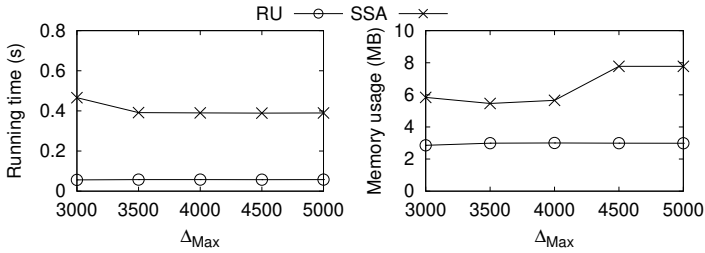
Effect of Query Size $|QW|$. Figure 28 shows the results of varying $|QW|$. The results of $|QW| = 1$ is not shown because $|QW'| = 0$ after the keyword removal. According to Figure 28(a), RU runs faster than SSA since the result reuse phase can identify some feasible routes quickly. According to Figure 28(b), the memory usage of RU is much smaller than SSA, because of the smaller search space in RU.

Effect of Δ_{Max}. Figure 29 shows the results of varying Δ_{Max}. According to Figure 29(a), the running times of both algorithms are insensitive to Δ_{Max}, and RU always runs faster than SSA. This again shows the computation time is reduced by the result reuse strategy. According to Figure 29(b), the memory usage of RU is smaller than that of SSA, and both algorithms consume less than 10MB of memory in different Δ_{Max} values.



(a) Running time (b) Memory usage

Fig. 28: Effect of $|QW|$ (Case (f))



(a) Running time (b) Memory usage

Fig. 29: Effect of Δ_{Max} (Case (f))

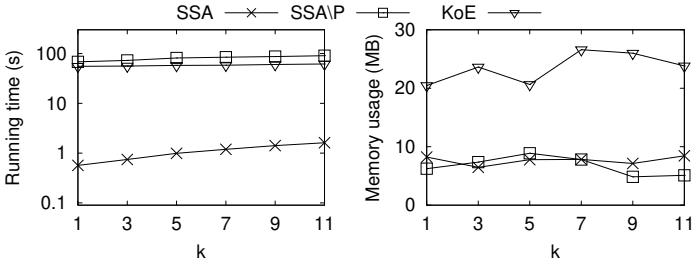
7.2 Experiment on Real Dataset

7.2.1 Set-up

Following [12], we conduct our experiment on real dataset of a shopping mall in Hangzhou, China. The building has 7 floors each approximately of 108m × 80m, and contains 10 staircases each being 20m long. It contains 639 stores, those of the same category are usually located on the same floor. The keywords of the stores are extracted from the mall's website. In total, we obtain 5036 t-words and 533 i-words. We categorize them into the 11 categories as in the synthetic dataset. Also, we use the same parameter settings as in Table 5.

7.2.2 Results on TIKRQ

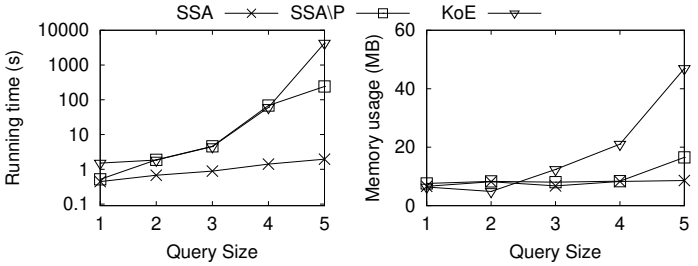
Effect of k. Figure 30 shows the results of varying k. According to Figure 30(a), the running times of both SSA and $SSA\backslash P$ increase slowly when k increases. This could be explained by the fact that the mall usually has shops of the same category (and thus having similar keywords) in the same floor. Thus, only small extra computation is needed to explore more results (i.e., a larger k). Besides, our SSA runs faster than its competitors by approximately two orders of magnitude, since our pruning techniques reduced the search space effectively. According to Figure 30(b), SSA consumes less than 10MB of memory in different values of k, similar to $SSA\backslash P$ and much less than KoE.



(a) Running time (b) Memory usage

Fig. 30: Effect of k

Effect of Query Size $|QW|$**.** Figure 31 shows the results of varying $|QW|$. According to Figure 31(a), the running times of all algorithms increase when query size increases, because larger key partition sets need to be formed. Our SSA consistently runs faster than $SSA\backslash P$ and KoE, and the gap enlarges when the query size increases, as contributed by the pruning techniques employed in SSA. In particular, when $|QW| = 5$, SSA takes less than 2 seconds to terminate, while KoE takes more than an hour (3600 seconds). According to Figure 31(b), the memory usage of SSA is usually smaller than the competitors, and is less than 10MB in different values of $|QW|$.



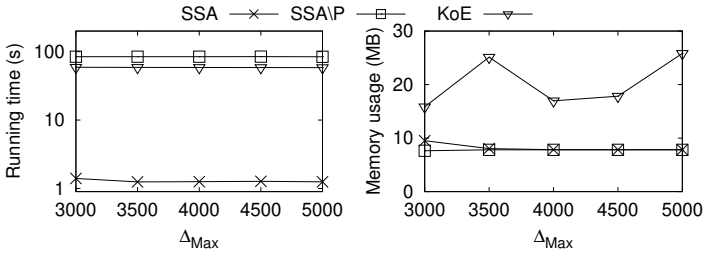
(a) Running time (b) Memory usage

Fig. 31: Effect of $|QW|$

Effect of Δ_{Max}**.** Figure 32 shows the results of varying Δ_{Max}. According to Figure 32(a), SSA runs faster than $SSA\backslash P$ and KoE by approximately two orders of magnitude. This is because our set-based search strategy and pruning techniques can find the result feasible routes quickly. According to Figure 32(b), the memory usages of SSA and $SSA\backslash P$ are similar, and both are smaller than that of KoE.

7.2.3 Variant 1: Preferred Visiting Order

Effect of k**.** Figure 33 shows the results of varying k. According to Figure 33(a), the running times of the algorithms increase when k increases, and SSA runs faster than both KoE and $SSA\backslash P$. This is because our pruning techniques and computation strategy are still effective when there is a visiting



(a) Running time (b) Memory usage

Fig. 32: Effect of Δ_{Max}

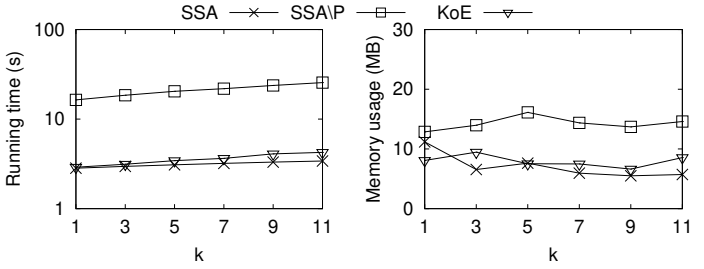
order requirement. According to Figure 33(b), the memory usages of SSA and KoE are similar, and both algorithms consume less memory than $SSA\backslash P$.

Comparing Figure 30(a) and Figure 33(a), we observe performance improvements on all three algorithms when visiting order is imposed. This is because the visiting order requirement helps to reduce the search space. Interestingly, we also observe that the improvement of KoE is much larger than that of SSA (and $SSA\backslash P$), which is likely because of the shops' location distribution in the dataset.

Specifically, the real dataset is a shopping mall, where the shops in the same category (and having similar i-words and t-words) are usually located closely, e.g., in a cluster on the same floor. Thus, when we do not have visiting order requirement, KoE has a huge search space and is slow, because it needs to iterate a lot of candidate routes that are feasible in order to find the best k routes (i.e., with the k smallest route costs).

When the visiting order is imposed, the search space in KoE is shrinked much more (compared to SSA), since there is only one possible visiting order of keywords for the routes. The improvement in SSA is less significant, since the original search space in SSA is much smaller, as contributed by our set-based search strategy and pruning techniques.

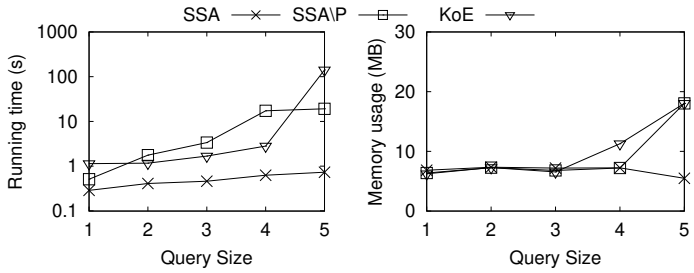
It is noteworthy that this significant improvement in KoE are not observed in the synthetic data (e.g., Figure 27(a) and Figure 11(a)), because the synthetic dataset does not exhibit such spatial proximity of the shops in the same category.



(a) Running time (b) Memory usage

Fig. 33: Effect of k (Variant 1: Preferred Visiting Order)

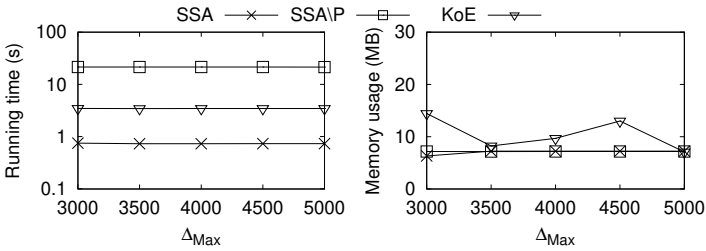
Effect of Query Size $|QW|$. Figure 34 shows the results of varying $|QW|$. According to Figure 34(a), the running times of all algorithms increase when the query size increases, and SSA always outperforms its competitors. This is because our pruning strategies are effective. In particular, KoE needs more than 100s to terminate when $|QW| = 5$, which shows that KoE is not scalable to query sizes. According to Figure 34(b), the memory usages of all algorithms are similar, and SSA is able to keep a low memory consumption in all cases.



(a) Running time (b) Memory usage

Fig. 34: Effect of $|QW|$ (Variant 1: Preferred Visiting Order)

Effect of Δ_{Max}. Figure 35 shows the results of varying Δ_{Max}. According to Figure 35(a), the running times of all algorithms are not affected by Δ_{Max}, which is probably because the layout of the mall allows shops to be easily accessed, and thus reducing the difficulty of finding the feasible route. Moreover, our SSA outperforms its competitors. According to Figure 35(b), the memory usages of all algorithms are similar, and SSA is always among the best due to its small search space.

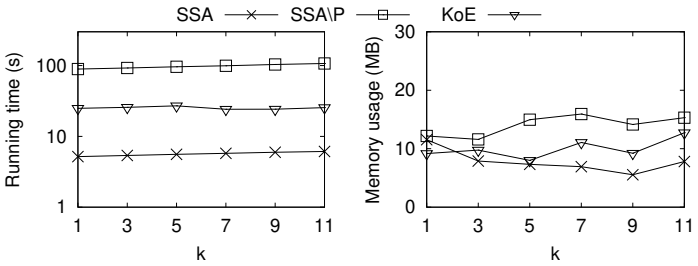


(a) Running time (b) Memory usage

Fig. 35: Effect of Δ_{Max} (Variant 1: Preferred Visiting Order)

7.2.4 Variant 2: Absence of Target Point

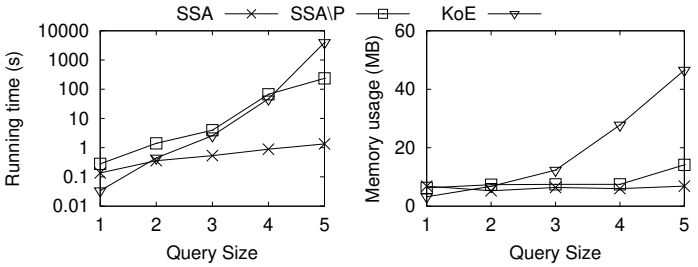
Effect of k. Figure 36 shows the results of varying k. According to Figure 36(a), the running times of all algorithms increase slightly when k increases. Our SSA runs faster than both $SSA\backslash P$ and KoE. This is because our pruning strategies are effective even in the case of without the target point. According to Figure 36(b), the memory usages of the algorithms are similar, and SSA usually consumes the least memory.



(a) Running time (b) Memory usage

Fig. 36: Effect of k (Variant 2: Absence of Target Point)

Effect of Query Size $|QW|$. Figure 37 shows the results of varying $|QW|$. According to Figure 37(a), the running times of all algorithms increase with query size, and SSA is more scalable to the query size. This is because our set-based search strategy is more robust to increases of query size. Similar to the case in synthetic dataset (i.e., Figure 15), when query size is small and target point is absence, the solution is usually trivial and easy to find. The time overhead of the pruning techniques in SSA outweighs the performance gain, and thus runs slower than KoE in such case. According to Figure 37(b), the memory usage of SSA is always the least, consumes less than 10MB of memory.



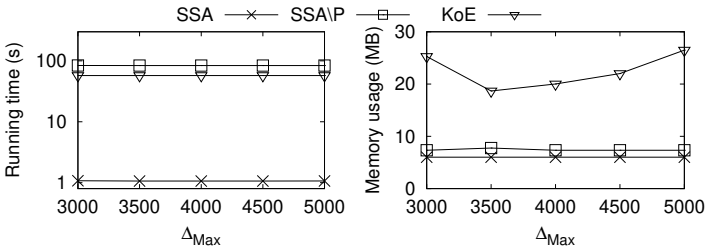
(a) Running time (b) Memory usage

Fig. 37: Effect of $|QW|$ (Variant 2: Absence of Target Point)

Effect of Δ_{Max}. Figure 38 shows the results of varying Δ_{Max}. According to Figure 38(a), the running time of SSA is faster than $SSA\backslash P$ and KoE by approximately two orders of magnitude, since the prunnings reduced the search space effectively. According to Figure 38(b), the memory usage of SSA is the least among the competitors.

7.2.5 Session-based TIKRQ: Reduced Time Budget (Case (c))

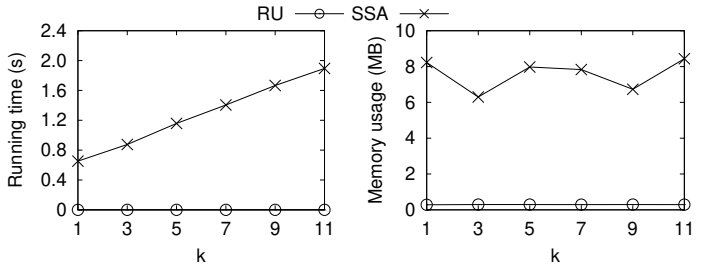
Effect of k. Figure 39 shows the results of varying k. According to Figure 39(a), the running time of SSA increases when k increases, while RU keeps its running time close to 0. This is because RU can reuse most of the results to avoid the time-consuming feasible route finding. According to



(a) Running time (b) Memory usage

Fig. 38: Effect of Δ_{Max} (Variant 2: Absence of Target Point)

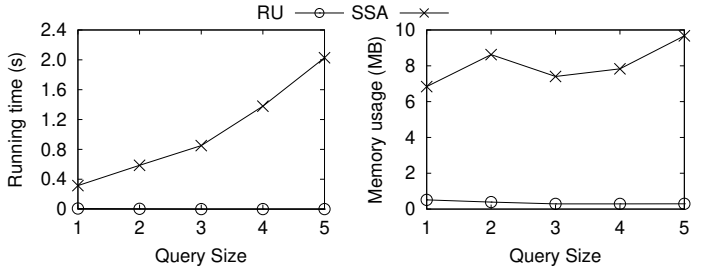
Figure 39(b), the memory usage of *RU* is also close to 0, much smaller than that of *SSA*.



(a) Running time (b) Memory usage

Fig. 39: Effect of k (Case (c))

Effect of Query Size $|QW|$. Figure 40 shows the results of varying $|QW|$. According to Figure 40(a), the running time of *SSA* increases with large query sizes, and *RU* terminates in close to 0 second. This is because our result reuse phase can reduce the number of new key partition sets that need to be explored. According to Figure 40(b), the memory usage of *RU* is much less than *SSA*. Still, both algorithms consume less than 10MB of memory.

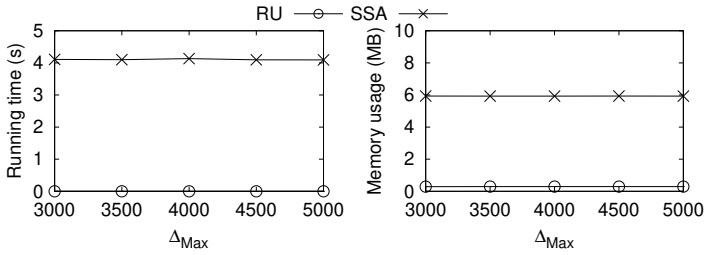


(a) Running time (b) Memory usage

Fig. 40: Effect of Query Size $|QW|$ (Case (c))

Effect of Δ_{Max}. Figure 41 shows the results of varying Δ_{Max}. According to Figure 41(a), the running times of both algorithms are insensitive to the value of Δ_{Max}. *RU* terminates much faster than *SSA*, since most of the results can

be reused. According to Figure 41(b), the memory usages of both algorithms are small (less than 10MB), and *RU* always consumes a much smaller amount of memory.

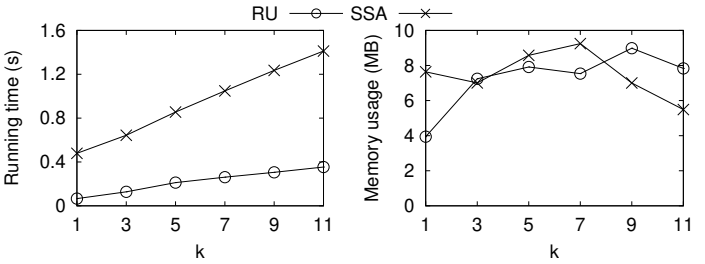


(a) Running time (b) Memory usage

Fig. 41: Effect of Δ_{Max} (Case (c))

7.2.6 Session-based TIKRQ: Changed source point (Case (d))

Effect of k. Figure 42 shows the results of varying k. According to Figure 42(a), the running times of both algorithms increase when k increases, and *RU* runs consistently faster than *SSA*. According to Figure 42(b), the memory usages of both algorithms fluctuate with k, and both are less than 10MB of memory.

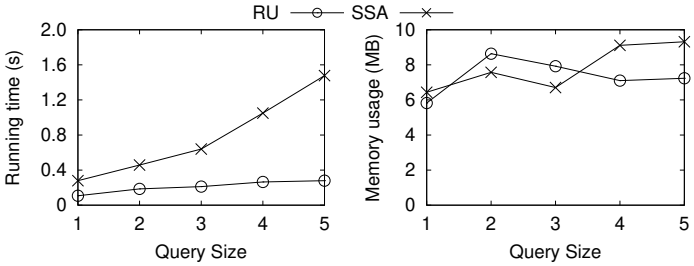


(a) Running time (b) Memory usage

Fig. 42: Effect of k (Case (d))

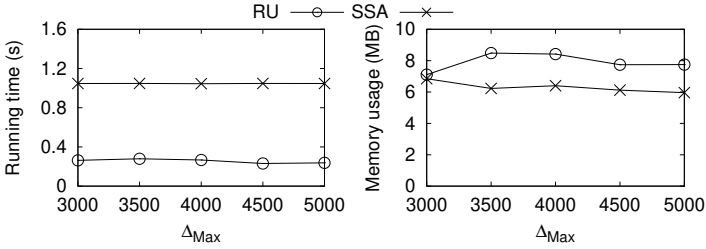
Effect of Query Size $|QW|$. Figure 43 shows the results of varying $|QW|$. According to Figure 43(a), *RU* runs faster than *SSA* consistently, and the differences increase with $|QW|$. This again show the advantage of our result reuse strategy. According to Figure 43(b), the memory usage of *RU* and *SSA* are similar, and are smaller than 10MB.

Effect of Δ_{Max}. Figure 44 shows the results of varying Δ_{Max}. According to Figure 44(a), while the running times of both algorithms are insensitive to Δ_{Max}, and *RU* consistently runs faster than *SSA*. This again is contributed by the result reuse phase in *RU*. According to Figure 44(b), the memory usage of *RU* is smaller than *SSA*, and both are less than 10MB.



(a) Running time (b) Memory usage

Fig. 43: Effect of $|QW|$ (Case (d))

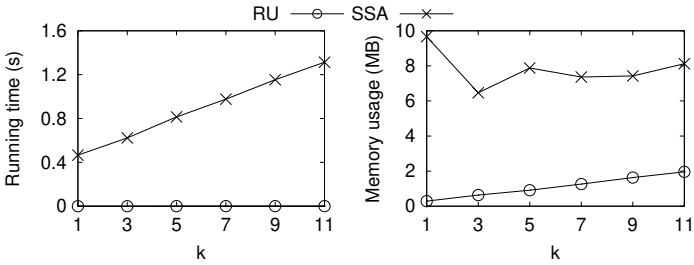


(a) Running time (b) Memory usage

Fig. 44: Effect of Δ_{Max} (Case (d))

7.2.7 Session-based TIKRQ: Changed target point (Case (e))

Effect of k**.** Figure 45 shows the results of varying k. According to Figure 45(a), the running time of SSA increases with k, and that of RU is always close to 0, consistently faster than SSA. In particular, RU can terminate within 0.1 seconds, which again is contributed by the result reuse. According to Figure 45(b), the memory usages of RU is smaller than SSA, and both algorithms use less than 10MB of memory in different k values.

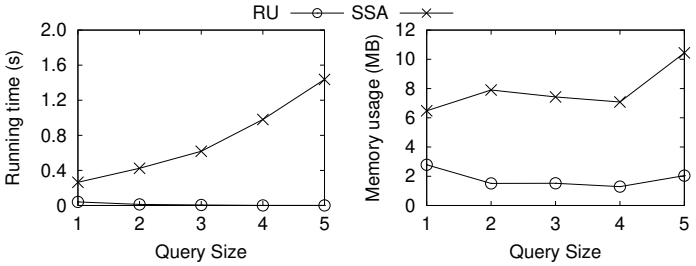


(a) Running time (b) Memory usage

Fig. 45: Effect of k (Case (e))

Effect of Query Size $|QW|$**.** Figure 46 shows the results of varying $|QW|$. According to Figure 46(a), RU runs consistently faster than SSA. Also, we observed that running time of RU decreases slightly when $|QW|$ increases. It

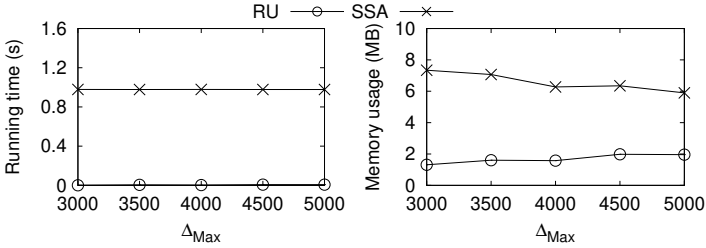
is because when $|QW|$ is large, more routes have been explored during the original search, and thus more information can be re-used, which lowered the running time. According to Figure 46(b), the memory usage of RU is smaller than SSA due to the reduced search space.



(a) Running time (b) Memory usage

Fig. 46: Effect of $|QW|$ (Case (e))

Effect of Δ_{Max}. Figure 47 shows the results of varying Δ_{Max}. According to Figure 47(a), while the running times of both algorithms are insensitive to Δ_{Max}, and RU terminates within 0.1 seconds, consistently runs faster than SSA. This again is contributed by the result reuse phase in RU. According to Figure 47(b), the memory usage of RU is smaller than SSA, and both are less than 10MB.



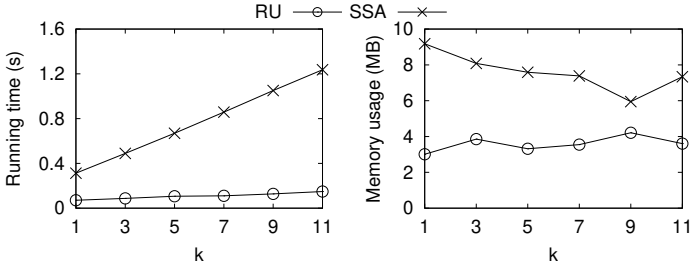
(a) Running time (b) Memory usage

Fig. 47: Effect of Δ_{Max} (Case (e))

7.2.8 Session-based TIKRQ: Keyword Removal (Case (f))

Effect of k. Figure 48 shows the results of varying k. According to Figure 48(a), the running times of both algorithms increase when k increases, and RU runs consistently faster than SSA. In particular, RU can terminate within 0.1 seconds, which again is contributed by the result reuse. According to Figure 48(b), the memory usages of both algorithms are not affected by the value of k, and are less than 10MB of memory.

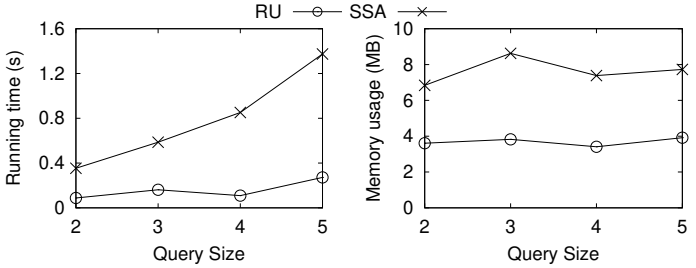
Effect of Query Size $|QW|$. Figure 49 shows the results of varying $|QW|$. The results of $|QW| = 1$ is not shown because $|QW'| = 0$ after the keyword removal. According to Figure 49(a), RU runs faster than SSA, as the result reuse phase found part of the result routes and thus reduced the computation needed to



(a) Running time (b) Memory usage

Fig. 48: Effect of k (Case (f))

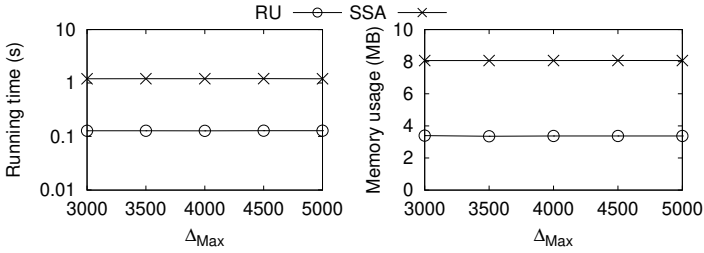
find feasible routes on new key partition sets. According to Figure 49(b), the memory usage of RU is smaller than SSA due to the reduced search space.



(a) Running time (b) Memory usage

Fig. 49: Effect of $|QW|$ (Case (f))

Effect of Δ_{Max}. Figure 50 shows the results of varying Δ_{Max}. According to Figure 50(a), while the running times of both algorithms are not affected by Δ_{Max}, RU consistently runs faster than SSA by an order of magnitude. This again is contributed by the result reuse phase in RU. According to Figure 50(b), the memory usage of RU is smaller than SSA, and both are less than 10MB.



(a) Running time (b) Memory usage

Fig. 50: Effect of Δ_{Max} (Case (f))

8 Related Work

Query Processing in Indoor Space. Efficient indoor query processing has received significant attention in recent years. Some works [28, 42, 44, 45] studied the indoor spatial queries such as range queries, kNN queries, and shortest path queries under various settings. Lu et al. [27] designed an indoor space model that facilitates indoor shortest path finding. Shao et al. [37, 38] proposed the VIP-Tree and KP-Tree that allows efficient processing of indoor shortest path queries and spatial keyword queries. However, they did not consider the keyword-aware routing queries, which is the focus of this work. Luo et al. [29] studied the time-constrained sequence route query in an indoor space. Their work considers the stay-time and partitions' category, but not the objects' static cost and thus their solution is not applicable to our problem. Liu et al. [25] studied the indoor temporal-aware shortest path query, which considers the current time stamp and the opening hours of the doors. The temporal-aware setting is orthogonal to our problem, and it can be integrated into our problem to model the case that partitions and doors have different opening hours. Liu et al. [26] proposed two indoor crowd-aware route planning queries, which find the fastest route and least crowded route. They developed a time-evolving population estimator to derive room populations for a future time. Both exact and approximate algorithms were proposed to answer the two queries. Chan et al. [6] studied the social distance monitoring query in the indoor space. They proposed a framework for monitoring and predicting the pairwise distances between users in an online setting. Li et al. [23] developed an indoor spatial data management system that supports k nearest neighbor query and shortest path query in an indoor space. Li et al. [24] proposed the indoor reverse k nearest neighbor query. They proposed an indoor influence computation algorithm, which utilizes the VIP-Tree [38] as the underlying index. Sun et al. [40] studied the indoor nearest neighbor search based on crowdsourced Received Signal Strength Indication (RSSI), which find an approximate nearest neighbor.

Keyword-aware Routing in Indoor Space. Salgado [34] studied the keyword-aware skyline route (KSR) search in indoor venues that considers the number of objects in the routes and the route distances. While KSR assumes that each partition contains one keyword, our setting allows a partition to have multiple keywords. Salgado et al. [35] studied the category-aware multi-criteria indoor route planning queries (CAM queries) that consider the objects' keyword and static cost. Shao et al. [36] studied the indoor trip planning queries (iTPQ) and developed a solution called VIP-tree neighbor expansion that exploits the features of indoor space, such as room and hallways. Recently, Feng et al. [12] studied the IKRQ problem, which finds k s-to-t routes with the highest scores that consider both the keyword relevance and spatial distance. Each route has a distance satisfying a pre-defined distance constraint. They defined prime route to return diverse results, and developed two algorithms for answering the query. All of these works fail to capture different criteria

at the same time and thus their solutions can not be directly applied to our problem. Table 7 shows the summary and comparison of all those proposals and our TIKRQ.

Table 7: Existing Indoor Keyword-aware Routing Queries

	Textual Constraint	**Spatial Constraint**	**Static Cost**	**Result Diversification**
KSR [34]	boolean	distance	-	Skyline
CAM [35]	boolean	distance	√	-
*i*TPQ [36]	boolean	distance	-	-
IKRQ [12]	i/t-word	distance	-	Prime route
TIKRQ	i/t/c-word	time budget	√	Unique partition set

Outdoor Keyword-aware Query and Routing. Some works [3, 9, 10, 31] studied the outdoor spatial keyword queries that retrieve a single object close to the query location and relevant to the query keywords, while others [4, 14, 43] find an object set as a solution.

Given a source point s, a target point t, and a category set C, the trip planning query [17] finds the shortest s-to-t route that passes at least one object from each category in C. For routing query with visiting order constraint, the optimal sequenced route query [39] finds the shortest route that passes the categories in the user-specified sequence, while others [8, 21] consider partial order. The keyword-aware optimal route query [2] finds a route that covers all query keywords, has the minimum objective score, and meets the given budget constraint. The optimal route search [46] finds a route that has maximum query keywords coverage and satisfies the budget constraint. The route of interest query [22] also searches a route that has maximum query keywords coverage and satisfies the budget constraint, but using a different equation to calculate the coverage. The clue-based route search [48] allows users to specify the order of keywords to cover and the distance range from one matched keyword to the next one. All these works fall short for indoor topology considered in our TIKRQ problem. Also, none of them organize the keywords according to their semantics.

Session-based Query and Routing. Kanza et al. [15] studied the interactive route search in which the route changes based on the user's feedback en route. Later this work was extended [16] to consider an additional complete or partial ordering requirement. Roy et al. [33] proposed the interactive itinerary planning problem, which allows users the give feedback on POI selected by the system. Chen at al. [7] proposed the TripPlanner that iteratively adds user preferred venues into candidate routes with specified venues. Zheng et al. [49] proposed interactive top-k spatial keyword queries that utilizes the users' feedback to improves the results of previous round. All these works only consider part of the parameter changes. Also, none of them work in indoor spaces as in our TIKRQ problem setting.

9 Conclusion and Future Work

In this paper, we studied the problem of time-constrained indoor keyword-aware routing, which find routes with minimum route costs while satisfying a time constraint. In our setting, the route costs capture both static cost and textual relevance of the route to meet the practical user needs. We developed a set-based search algorithm SSA to answer the query. Moreover, we discussed two variants of TIKRQ, namely preferred visiting order and absence of a target point. In addition, we studied the session-based TIKRQ and developed algorithms for four cases. Extensive experiments were conducted on both real and synthetic datasets, which verified the efficiency and sclability of our algorithms.

For future work, we can take the opening hours of partitions and doors into account. It is also interesting to study the hierarchical word organization for partitions and to provide suggestions to refine query keywords when no route is found. Moreover, it is relevant to consider additional constraints like prohibiting staircases in a route.

10 Declarations

Funding. This work was supported by Independent Research Fund Denmark (Grant No. 8022-00366B).

Conflict of Interest. Harry Kai-Ho Chan is affiliated to the Information School, University of Sheffield, United Kingdom. Partial of his work was done at the Department of People and Technology, Roskilde University, Denmark. Tiantian Liu and Hua Lu are affiliated to the Department of People and Technology, Roskilde University, Denmark. Huan Li is affiliated to the Department of Computer Science, Aalborg University, Denmark.

Data Availability Statement. The code used and datasets generated during and/or analysed during the current study are available in the GitHub repository, https://github.com/harryckh/TIKRQ.

References

[1] Basiri, A., Lohan, E.S., Moore, T., Winstanley, A., Peltola, P., Hill, C., Amirian, P., e Silva, P.F.: Indoor location based services challenges, requirements and usability of current solutions. Computer Science Review **24**, 1–12 (2017)

[2] Cao, X., Chen, L., Cong, G., Xiao, X.: Keyword-aware optimal route search. PVLDB **5**(11), 1136–1147 (2012)

[3] Cary, A., Wolfson, O., Rishe, N.: Efficient and scalable method for processing top-k spatial boolean queries. In: SSDBM, pp. 87–95 (2010). Springer

[4] Chan, H.K.-H., Long, C., Wong, R.C.-W.: On generalizing collective spatial keyword queries. TKDE **30**(9), 1712–1726 (2018)

[5] Chan, H.K.-H., Liu, T., Li, H., Lu, H.: Time-constrained indoor keyword-aware routing. In: SSTD, pp. 74–84 (2021)

[6] Chan, H.K.-H., Li, H., Li, X., Lu, H.: Continuous social distance monitoring in indoor space. PVLDB **15**(7), 1390–1402 (2022)

[7] Chen, C., Zhang, D., Guo, B., Ma, X., Pan, G., Wu, Z.: Tripplanner: Personalized trip planning leveraging heterogeneous crowdsourced digital footprints. IEEE Transactions on Intelligent Transportation Systems **16**(3), 1259–1273 (2014)

[8] Chen, H., Ku, W.-S., Sun, M.-T., Zimmermann, R.: The multi-rule partial sequenced route query. In: SIGSPATIAL, p. 10 (2008). ACM

[9] Cong, G., Jensen, C.S., Wu, D.: Efficient retrieval of the top-k most relevant spatial web objects. PVLDB **2**(1), 337–348 (2009)

[10] De Felipe, I., Hristidis, V., Rishe, N.: Keyword search on spatial databases. In: ICDE, pp. 656–665 (2008). IEEE

[11] Fakas, G.J., Cai, Y., Cai, Z., Mamoulis, N.: Thematic ranking of object summaries for keyword search. DKE **113**, 1–17 (2018)

[12] Feng, Z., Liu, T., Li, H., Lu, H., Shou, L., Xu, J.: Indoor top-k keyword-aware routing query. In: ICDE, pp. 1213–1224 (2020). IEEE

[13] Golden, B.L., Levy, L., Vohra, R.: The orienteering problem. Naval Research Logistics (NRL) **34**(3), 307–318 (1987)

[14] Guo, T., Cao, X., Cong, G.: Efficient algorithms for answering the m-closest keywords query. In: SIGMOD, pp. 405–418 (2015). ACM

[15] Kanza, Y., Levin, R., Safra, E., Sagiv, Y.: An interactive approach to route search. In: SIGSPATIAL, pp. 408–411 (2009)

[16] Kanza, Y., Levin, R., Safra, E., Sagiv, Y.: Interactive route search in the presence of order constraints. PVLDB **3**(1-2), 117–128 (2010)

[17] Li, F., Cheng, D., Hadjieleftheriou, M., Kollios, G., Teng, S.-H.: On trip planning queries in spatial databases. In: SSTD, pp. 273–290 (2005). Springer

[18] Li, H., Lu, H., Shou, L., Chen, G., Chen, K.: Finding most popular indoor semantic locations using uncertain mobility data. TKDE **31**(11), 2108–2123 (2018a)

[19] Li, H., Lu, H., Shou, L., Chen, G., Chen, K.: In search of indoor dense regions: An approach using indoor positioning data. TKDE **30**(8), 1481–1495 (2018b)

[20] Li, H., Lu, H., Cheema, M.A., Shou, L., Chen, G.: Indoor mobility semantics annotation using coupled conditional markov networks. In: ICDE, pp. 1441–1452 (2020). IEEE

[21] Li, J., Yang, Y.D., Mamoulis, N.: Optimal route queries with arbitrary order constraints. TKDE **25**(5), 1097–1110 (2012)

[22] Li, W., Cao, J., Guan, J., Yiu, M.L., Zhou, S.: Retrieving routes of interest over road networks. In: WAIM, pp. 109–123 (2016). Springer

[23] Li, Y., Yang, S., Cheema, M.A., Shao, Z., Lin, X.: Indoorviz: A demonstration system for indoor spatial data management. In: SIGMOD, pp. 2755–2759 (2021)

[24] Li, Y., Ma, G., Yang, S., Wang, L., Zhang, J.: Influence computation for indoor spatial objects. In: DASFAA, pp. 259–267 (2022). Springer

[25] Liu, T., Feng, Z., Li, H., Lu, H., Cheema, M.A., Cheng, H., Xu, J.: Shortest path queries for indoor venues with temporal variations. In: ICDE, pp. 2014–2017 (2020). IEEE

[26] Liu, T., Li, H., Lu, H., Cheema, M.A., Shou, L.: Towards crowd-aware indoor path planning. PVLDB **14**(8), 1365–1377 (2021)

[27] Lu, H., Cao, X., Jensen, C.S.: A foundation for efficient indoor distance-aware query processing. In: ICDE, pp. 438–449 (2012). IEEE

[28] Lu, H., Yang, B., Jensen, C.S.: Spatio-temporal joins on symbolic indoor tracking data. In: ICDE, pp. 816–827 (2011). IEEE

[29] Luo, W., Jin, P., Yue, L.: Time-constrained sequenced route query in indoor spaces. In: APWeb, pp. 129–140 (2016). Springer

[30] Qin, L., Yu, J.X., Chang, L.: Diversifying top-k results. PVLDB **5**(11), 1124–1135 (2012)

[31] Rocha-Junior, J.B., Gkorgkas, O., Jonassen, S., Nørvåg, K.: Efficient processing of top-k spatial keyword queries. In: SSTD, pp. 205–222 (2011). Springer

[32] Rose, S., Engel, D., Cramer, N., Cowley, W.: Automatic keyword extraction from individual documents. Text Mining: Applications and Theory **1**, 1–20 (2010)

[33] Roy, S.B., Das, G., Amer-Yahia, S., Yu, C.: Interactive itinerary planning. In: 2011 IEEE 27th International Conference on Data Engineering, pp. 15–26 (2011). IEEE

[34] Salgado, C.: Keyword-aware skyline routes search in indoor venues. In: SIGSPATIAL-ISA, pp. 25–31 (2018)

[35] Salgado, C., Cheema, M.A., Taniar, D.: An efficient approximation algorithm for multi-criteria indoor route planning queries. In: SIGSPATIAL, pp. 448–451 (2018)

[36] Shao, Z., Cheema, M.A., Taniar, D.: Trip planning queries in indoor venues. The Computer Journal **61**(3), 409–426 (2018)

[37] Shao, Z., Cheema, M.A., Taniar, D., Lu, H.: VIP-tree: an effective index for indoor spatial queries. PVLDB **10**(4), 325–336 (2016)

[38] Shao, Z., Cheema, M.A., Taniar, D., Lu, H., Yang, S.: Efficiently processing spatial and keyword queries in indoor venues. TKDE (2020)

[39] Sharifzadeh, M., Kolahdouzan, M., Shahabi, C.: The optimal sequenced route query. VLDBJ **17**(4), 765–787 (2008)

[40] Sun, J., Wang, B., Yang, X.: Practical approximate indoor nearest neighbour locating with crowdsourced rssis. World Wide Web **24**(3), 747–779 (2021)

[41] Xie, X., Lu, H., Pedersen, T.B.: Efficient distance-aware query evaluation on indoor moving objects. In: ICDE, pp. 434–445 (2013). IEEE

[42] Xie, X., Lu, H., Pedersen, T.B.: Distance-aware join for indoor moving objects. TKDE **27**(2), 428–442 (2014)

[43] Xu, H., Gu, Y., Sun, Y., Qi, J., Yu, G., Zhang, R.: Efficient processing of moving collective spatial keyword queries. VLDBJ **29**(4), 841–865 (2020)

[44] Yang, B., Lu, H., Jensen, C.S.: Scalable continuous range monitoring of moving objects in symbolic indoor space. In: CIKM, pp. 671–680 (2009)

[45] Yuan, W., Schneider, M.: Supporting continuous range queries in indoor space. In: MDM, pp. 209–214 (2010). IEEE

[46] Zeng, Y., Chen, X., Cao, X., Qin, S., Cavazza, M., Xiang, Y.: Optimal route search with the coverage of users' preferences. In: IJCAI, pp. 2118–2124 (2015)

[47] Zhang, C., Zhang, Y., Zhang, W., Lin, X., Cheema, M.A., Wang, X.:

Diversified spatial keyword search on road networks. In: EDBT, pp. 367–378 (2014)

[48] Zheng, B., Su, H., Hua, W., Zheng, K., Zhou, X., Li, G.: Efficient clue-based route search on road networks. TKDE **29**(9), 1846–1859 (2017)

[49] Zheng, K., Su, H., Zheng, B., Shang, S., Xu, J., Liu, J., Zhou, X.: Interactive top-k spatial keyword queries. In: ICDE, pp. 423–434 (2015). IEEE