Effective and Efficient Reuse of Past Travel Behavior for Route Recommendation

Lisi Chen[1], Shuo Shang[2,*], Christian S. Jensen[3], Bin Yao[4], Zhiwei Zhang[5], Ling Shao[6]
[2]UESTC, China, [1,2,6]Inception Institute of Artificial Intelligence, UAE, [3]Aalborg University, Denmark
[4]Shanghai Jiao Tong University, China, [5]Hong Kong Baptist University
[1]lisi.chen@inceptioniai.org, [2]jedi.shang@gmail.com, [3]csj@cs.aau.dk
[4]yaobin@cs.sjtu.edu.cn, [5]cszwzhang@comp.hkbu.edu.hk, [6]ling.shao@inceptioniai.org

ABSTRACT

With the increasing availability of moving-object tracking data, use of this data for route search and recommendation is increasingly important. To this end, we propose a novel parallel split-and-combine approach to enable route search by locations (RSL-Psc). Given a set of routes, a set of places to visit O, and a threshold θ, we retrieve the route composed of sub-routes that (i) has similarity to O no less than θ and (ii) contains the minimum number of sub-route combinations. The resulting functionality targets a broad range of applications, including route planning and recommendation, ridesharing, and location-based services in general.

To enable efficient and effective RSL-Psc computation on massive route data, we develop novel search space pruning techniques and enable use of the parallel processing capabilities of modern processors. Specifically, we develop two parallel algorithms, Fully-Split Parallel Search (FSPS) and Group-Split Parallel Search (GSPS). We divide the route split-and-combine task into $\sum_{k=0}^{M} S(|O|, k+1)$ sub-tasks, where M is the maximum number of combinations and $S(\cdot)$ is the Stirling number of the second kind. In each sub-task, we use network expansion and exploit spatial similarity bounds for pruning. The algorithms split candidate routes into sub-routes and combine them to construct new routes. The sub-tasks are independent and are performed in parallel. Extensive experiments with real data offer insight into the performance of the algorithms, indicating that our RSL-Psc problem can generate high-quality results and that the two algorithms are capable of achieving high efficiency and scalability.

KEYWORDS

Route recommendation; Trajectory search

ACM Reference Format:
Lisi Chen, Shuo Shang, Christian S. Jensen, Bin Yao, Zhiwei Zhang, Ling Shao. 2019. Effective and Efficient Reuse of Past Travel Behavior for Route Recommendation. In *The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'19), August 4–8, 2019, Anchorage, AK, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3292500.3330835

* Corresponding author.

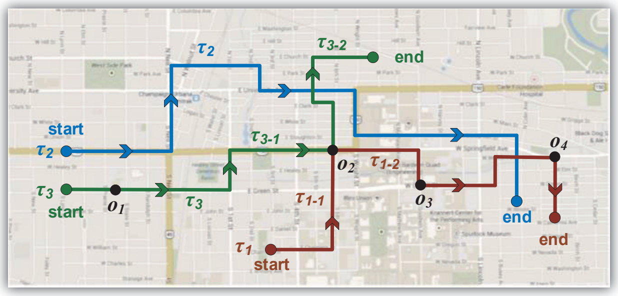


Figure 1: An example of the RSL-Psc problem

1 INTRODUCTION

The continued proliferation of GPS-equipped mobile devices (e.g., vehicle navigation systems and smart phones) and the proliferation of online map-based services (e.g., Bing Maps, Google Maps, and MapQuest) enable the collection and sharing of travel routes. Specialized sites, including Bikely, GPS-Way-Points, Share-My-Routes, and Microsoft Geolife [20], as well as general social network sites, including Twitter, Facebook, and Foursquare, are starting to support route sharing and search. The availability of massive route data enables novel mobile functionality, including route search by locations (RSL query [1, 12, 13]), which retrieves routes that are similar in some specific sense to a set of user-specified places (e.g., sightseeing places).

The RSL query is useful in a broad range of applications, including route planning and recommendation, ridesharing, and location based services in general [1, 12, 13]. For example, tourists can exploit the travel histories of other tourists to improve their own travel. Others with similar interests may have visited nearby landmarks that the tourist may not know, but may be interested in; or others may have avoided a specific road because it is unpleasant, although it may seem like a good choice in term of distance. Such experiences are captured in the routes shared by previous tourists. In addition, tourists may post their routes to attract potential ridesharing partners. The RSL query can identify such tourists with similar interests (i.e., the user-specified places are similar to the posted route) and can recommend them as ridesharing partners.

In most existing studies (e.g., [1, 12, 13]), the RSL query is defined as a top-k query. However, sometimes the quality of query results cannot be guaranteed due to insufficient data (e.g., the top-1 route is relatively far away from the user-specified places). Consider the example in Figure 1, where o_1, o_2, o_3, and o_4 are query locations (user-specified places) and τ_1, τ_2, and τ_3 are routes. Compared to τ_1

and τ_3, route τ_2 is spatially close to the query locations so it is returned as the top-1 result. However, this result is of low quality (i.e., relatively far away from the query locations), and it is not a useful result in real applications like travel planning and recommendation and ridesharing.

This motivates us to study a novel split-and-combine approach to solving the RSL problem. Given a set of routes, a set of user-specified places O, and a threshold θ, we retrieve the route τ that consists of several sub-routes that satisfy two conditions: (1) τ has the similarity to O no less than θ, and (2) τ contains the minimum number of sub-routes (the minimum number of transfers in ridesharing). Consider the example in Figure 1, where routes τ_1 and τ_3 split from o_2 into sub-routes τ_{1-1} (from $\tau_1.start$ to o_2), τ_{1-2} (from o_2 to $\tau_1.end$), τ_{3-1} (from $\tau_3.start$ to o_2) , and τ_{3-2} (from o_2 to $\tau_3.end$). Here, we combine sub-routes τ_{3-1} and τ_{1-2} to make up a new route $\tau = \langle \tau_{3-1}, \tau_{1-2} \rangle$. Compared to the original routes τ_1, τ_2, and τ_3, τ matches query locations o_1, o_2, o_3, and o_4 well while combining only two sub-routes.

The RSL-Psc problem is applied in spatial networks because in many practical scenarios, objects (e.g., commuters and vehicles) move in spatial networks [9, 11, 12] rather than in Euclidean space. In spatial networks, the most relevant distance notion when quantifying the distance between two objects is network distance; Euclidean distance may lead to errors. We adopt aggregate-distance matching (i.e., the sum of distances between query locations and routes) [1, 9, 10, 12] to match routes and query locations.

The RSL-Psc problem is challenging due to its high computation complexity. There exist $\sum_{k=0}^{M} S(|O|, k+1)$ possibilities when partitioning the set O of query locations, where M is the maximum number of combinations (e.g., the tolerance of transfer times for a tourist) and $S(\cdot)$ is the Stirling number of the second kind. The computations in different partitionings are independent of each other so can occur in parallel. We propose two parallel solutions to the RSL-Psc problem.

Fully-Split Parallel Search: In Fully-Split Parallel Search (FSPS), we first use network expansion [3] to explore the spatial network from each query location $o \in O$ and retrieve the route candidates that are spatially close to o. We define a distance lower bound and a similarity upper bound to prune the search space. Then we partition set O into $(k+1)$ subsets, where $k \in [0, M]$, and we refine the value of k from the minimum to the maximum (since we retrieve the routes with the minimum number of combinations, once we find a qualified route, it is unnecessary to consider larger values of k). For each possible subset $O_i \subseteq O$, we select the intersection of the route candidate sets of the corresponding query locations and generate the route candidate set at the subset level (i.e., the route candidate set of O_i). Next, we combine and evaluate candidate sets associated with every query location subset in each partitioning to obtain the query result. The computations in each subset and in each partitioning occur in parallel.

The advantage of the FSPS algorithm is that it only needs to conduct the route search once, after which it can reuse the search results for route combination. Its limitation lies in the tightness of its upper and lower bounds (it uses a single distance to prune an aggregate distance). As a result, each query location must maintain a large candidate set C. When combining route candidates (two steps: locations to subsets, and subsets to partitionings), a larger $|C|$ causes a very high (exponential) number of combination possibilities.

Group-Split Parallel Search: To further improve the efficiency of RSL-Psc processing, we propose the Group-Split Parallel Search (GSPS) algorithm that adopts a divide-and-conquer strategy. In the case of GSPS, we partition the set of query locations into $(k+1)$ subsets ($O = O_1 \cup \ldots \cup O_{k+1}$), where $k \in [0, M]$ and we refine the value of k from the minimum to the maximum. For each value of k, we have $S(|O|, k+1)$ possible partitionings. For each subset O_i ($i \in [1, k+1]$), we search the route candidates that are spatially close to O_i. Upper and lower bounds on the aggregate distance are defined in order to prune the search space. The route search in each subset is performed in parallel. Then we combine and evaluate the route candidates of the location subsets of the same partitioning, again performing the computation for each partitioning in parallel. Compared to FSPS, GSPS achieves tighter candidate sets and avoids the combination from locations to subsets.

Our contributions can be summarized as follows. First, we propose a novel parallel split-and-combine approach to tackling the problem of route search by locations (RSL-Psc) efficiently and effectively, thus targeting applications such as route planning and recommendation, ridesharing, and location-based services in general. Second, we develop two efficient algorithms, Fully-Split Parallel Search (FSPS) (Section 3) and Group-Split Parallel Search (GSPS) (Section 4), to process the RSL-Psc query efficiently. Third, we conduct extensive experiments on large real route data sets to study the performance of the algorithms (Section 5). Our experiment results show that the PSL-Psc query is much more likely to return a valid result compared with the PSL query without route combination.

2 PRELIMINARIES

2.1 Spatial Networks and Routes

A spatial network is modeled as a connected, undirected graph $G = (V, E, F, W)$, where V is a vertex set and $E \subseteq \{\{v_i, v_j\} | v_i, v_j \in V \wedge v_i \neq v_j\}$ is an edge set. A vertex $v_i \in V$ represents a road intersection or an end of a road, and an edge $e_k = \{v_i, v_j\} \in E$ represents a road segment that enables travel between vertices v_i and v_j. Function $F : V \cup E \longrightarrow Geometries$ maps a vertex to the point location of the corresponding road intersection and maps an edge to a polyline representing the corresponding road segment. Function $W : E \longrightarrow R$ assigns a real-valued weight $W(e)$ to an edge e that represents the corresponding road segment's length.

The shortest path between two vertices v_i and v_j is a sequence of edges linking v_i and v_j such that the sum of their edge weights is minimal. Such a path is denoted by $SP(v_i, v_j)$, and its length is denoted by $sd(v_i, v_j)$. Euclidean-space based spatial indices (e.g., the R-tree [6]) and accompanying techniques are relatively ineffective in network environments due to loose bounds. For simplicity, we assume that the data points considered (e.g., route sample points and query locations) are located on vertices.

<u>**Definition 1:**</u> (*Route*) A route τ of is a finite sequence $\langle p_1, p_2, \ldots, p_n \rangle$ that consists of at least 2 vertices, where p_i and p_{i+1} ($i \in [1, n-1]$) are adjacent vertices in V. □

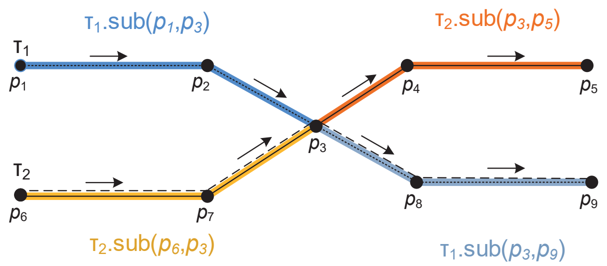


Figure 2: An example of route split-and-combine

Definition 2: (*Sub-route*) A sub-route of τ, denoted by $\tau.\text{sub}(p_i, p_j)$, is a segment of τ, where $p_i, p_j \in \tau$ and p_i and p_j are the start point and end point of the segment, respectively. □

Definition 3: (*Conflicting directions*) Assume that $\tau_a.\text{sub}(p_i, p_j)$ and $\tau_b.\text{sub}(p_k, p_l)$ are two sub-routes of τ_a and τ_b, respectively. If p_i and p_l are the same vertex or p_j and p_l are the same vertex, we say that they do not have conflicting directions; Otherwise, they have conflicting directions. □

If two routes τ_1 and τ_2 have an intersection point (i.e., $\tau_1 \cap \tau_2 \neq \emptyset$) that is neither the common start point of τ_1 and τ_2 nor the common end point of τ_1 and τ_2, they can be split and combined from that intersection. An example of route split-and-combine is shown in Figure 2, where $\tau_1 = \langle p_1, p_2, p_3, p_8, p_9\rangle$ and $\tau_2 = \langle p_6, p_7, p_3, p_4, p_5\rangle$. Specifically, p_3 is their intersection. We split τ_1 and τ_2 at p_3 into sub-routes $\tau_1.\text{sub}(p_1, p_3) = \langle p_1, p_2, p_3\rangle$ and $\tau_1.\text{sub}(p_3, p_9) = \langle p_3, p_8, p_9\rangle$, and $\tau_2.\text{sub}(p_6, p_3) = \langle p_6, p_7, p_3\rangle$ and $\tau_2.\text{sub}(p_3, p_5) = \langle p_3, p_4, p_5\rangle$, respectively. Then we combine $\tau_2.\text{sub}(p_6, p_3)$ and $\tau_1.\text{sub}(p_3, p_9)$ into a new route $\tau = \tau_2.\text{sub}(p_6, p_3) + \tau_1.\text{sub}(p_3, p_9) = \langle p_6, p_7, p_3, p_8, p_9\rangle$, which consists of one combination. Notice that sub-routes $\tau_1.\text{sub}(p_1, p_3)$ and $\tau_2.\text{sub}(p_6, p_3)$, and $\tau_1.\text{sub}(p_3, p_9)$ and $\tau_2.\text{sub}(p_3, p_5)$ cannot be combined because their directions conflict.

2.2 Distance Measures

Given a query location o and a route τ, the spatial network distance $d(o, \tau)$ between them is defined as follows.

$$d(o, \tau) = \min_{p_i \in \tau}\{sd(o, p_i)\}, \tag{1}$$

where $sd(o, p_i)$ denotes the shortest network distance between o and p_i. Given a set O of query locations and a route τ, the similarity $\text{Sim}(O, \tau)$ between them is defined according to aggregate distance [1, 9]:

$$\text{Sim}(O, \tau) = \sum_{o \in O} e^{-d(o,\tau)} \tag{2}$$

2.3 Problem Definition

We formally define the RSL-Psc problem in Definition 4.

Definition 4: (*RSL-Psc Problem*) Given a set O of query locations, a set T of routes, a threshold θ, and the maximum number of combinations M, RSL-Psc finds a route τ_r satisfying the following conditions:

(1) $\text{Sim}(O, \tau_r) \geq \theta$;

(2) $\tau_r = \tau_1.\text{sub}(p_1, p_2) + \tau_2.\text{sub}(p_2, p_3) + \ldots$
$+\tau_{m-1}.\text{sub}(p_{m-1}, p_m) + \tau_m.\text{sub}(p_m, p_{m+1})$, where
$m - 1 \leq M$ and $\tau_i \in T$ $(1 \leq i \leq m)$;

(3) for any k that $k < m$, we cannot find a route τ_u such that
$\text{Sim}(O, \tau_u) \geq \theta$ and
$\tau_u = \tau_1.\text{sub}(p_1, p_2) + \tau_2.\text{sub}(p_2, p_3) + \ldots$
$+\tau_{k-1}.\text{sub}(p_{k-1}, p_k) + \tau_k.\text{sub}(p_k, p_{k+1})$, where
$\tau_i \in T$ $(1 \leq i \leq k)$. □

3 FULLY-SPLIT PARALLEL SEARCH

3.1 Basic Idea

We propose the first solution to the RSL-Psc problem, the Fully-Split Parallel Search (FSPS) algorithm. Initially, we fully split the set O of query locations into $|O|$ individual expansion centers. Then we use network expansion [3] to explore the spatial network and to retrieve route candidates that are spatially close to each expansion center. We define an upper bound on network distance to prune the search space (Section 3.2). Next, we generate partitionings of the set O. Each partitioning consists of $(k + 1)$ disjoint subsets $(k \in [0, M])$, which are called *groups*. We refine the value of k from the minimum to the maximum. Since we retrieve the routes with the minimum number of combinations, once we find a qualified route, it is unnecessary to consider routes with a larger value of k. For each group in each partitioning, we combine and evaluate the candidates of the corresponding expansion centers; For each partitioning, we combine and evaluate the candidates of the corresponding subsets. The computations in each group and in each partitioning occur in parallel (Section 3.3). The FSPS algorithm is detailed in Section A.

3.2 Parallel Expansion Search

Consider the example in Figure 3, where $O = \{o_1, o_2, o_3\}$ is a set of query locations, and τ_1, τ_2, and τ_3 are routes. Set O is fully split and each query location is used as an expansion center. Network expansion is performed from each query location o_i $(i \in [1, 3])$ using Dijkstra's algorithm [3]. The exploration space of query location o_i is a region (o_i, rs_i), where radius rs_i is the network distance from the center o_i to the expansion boundary (i.e., the explore regions of o_1, o_2, and o_3 are depicted by the blue, red, and green regions, respectively). As Dijkstra's algorithm always selects the vertex with the minimum distance label for expansion, if $p \in \tau$ is the first vertex scanned by the expansion from o, p is the vertex closest to o, i.e.,

$$d(o, \tau) = sd(o, p). \tag{3}$$

For example, in Figure 3, $d(o_1, \tau_2) = sd(o_1, p_1)$, the shortest path between o_1 and p_1 is illustrated by the dashed line in the blue region. Notice that the expansions from query locations are independent of each other; thus, they occur in parallel.

Upper bound: For each expansion center o_i, if a vertex p is scanned by the expansion from o_i and route τ passes p, we can derive the similarity upper bound of τ as follow:

$$\text{Sim}(O, \tau) = \sum_{o_j \in O} e^{-d(o_j,\tau)}$$
$$< |O| - 1 + e^{-sd(o_i,p)} = \text{Sim}(O, \tau).ub \tag{4}$$

Here, $e^{-d(o_j,\tau)} < 1$, so we use 1 as the upper bound for all $o_j \in O \setminus \{o_i\}$. For each expansion center o_i, if its similarity upper bound is smaller than the threshold θ, the expansion from o_i terminates, and all unscanned routes are pruned safely. We place all scanned routes (e.g., in Figure 3, τ_2 and τ_3, passing through the red region,

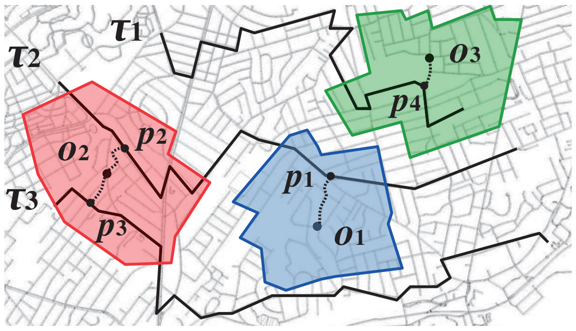


Figure 3: An example of network expansion

are scanned by the expansion from o_2) in a candidate set $C(o_i)$ for expansion center o_i, and we also maintain the network distance $d(o_i, \tau)$ for each candidate (it is derived during network expansion, see Equation 3).

Now we are ready to present the FSPS algorithm. Specifically, FSPS consists of two steps: (1) Generating route candidate sets (Section 3.3); (2) Combining route candidates from different query location subsets (Section 3.4).

3.3 Generating Route Candidate Sets

Before presenting the algorithm, we first define *route tuple*, which will be used to record the route candidates and corresponding labels for each query location. Route tuples will be retrieved and updated when combining route candidates from different query location subsets.

Definition 5: (*Route Tuple*) A route tuple of route τ associated with query location o_i and point p is denoted by $o_i(\tau, p) = \langle e, p, d\rangle$, which consists of three elements: an entry e (identifier) of route τ, an *expansion point* p in τ scanned by network expansion, and the shortest network distance d between p and o_i. □

Algorithm 1 presents the pseudo code for generating a route candidate set. The inputs are a set of query location O, the route dataset T, and the similarity threshold θ. The output is a route candidate tuple set of each query location (i.e., $C = \{C_1, C_2, ..., C_{|O|}\}$). Note that each C_i is maintained as a priority queue and contains route tuples associated with query location o_i. Specifically, the route tuples in C_i is sorted in ascending order of $o_i(\tau).d$.

We first initialize the route (candidate) tuple set associated with each query location (lines 1–2). Next, we find the route candidate tuple set of each query location $o_i \in O$. Specifically, we perform a network expansion from query location o_i. If an unvisited vertex exists (line 4), we retrieve the next unvisited vertex p (line 5). Then we update the upper bound of route candidate tuple set associated with o_i (i.e., ub_i) to be $|O| - 1 + e^{-sd(o_i,p)}$ (Equation 4) (line 6). If ub_i is no less than the similarity threshold θ, we regard all routes whose vertices contain p as candidates, so we add their route candidate tuples to C_i (lines 8–9). If the value of ub_i is lower than θ, the expansion from o_i terminates (lines 10–11). Having searched all query locations, we combine their results and get the result C of route candidates.

Algorithm 1: FSPSCandidates

```
Data: Query location set $O$, trajectory set $T$, similarity threshold $\theta$
Result: $C = \{C_1, C_2, ..., C_{|O|}\}$
1  for each $C_i$ in $C$ do
2      $C_i \leftarrow \emptyset$;
3  for each query location $o_i$ in $O$ do
4      while DijkstraExpansion($o_i$).hasNext() do
5          $p \leftarrow$DijkstraExpansion($o_i$).next();
6          $ub_i \leftarrow |O| - 1 + e^{-sd(o_i,p)}$;
7          if $ub_i \geq \theta$ then
8              for each $\tau$ s.t. $p \in \tau$ do
9                  $C_i$.add($o_i(\tau, p)$);
10         else
11             break;
12 return $C$;
```

3.4 Combining Route Candidates

Now we have a route candidate tuple set of each query location. The next step is to combine route candidates and acquire the final route. In particular, we need to: (1) Generate partitionings for query locations; (2) For each partitioning, retrieve route candidates associated with each query location group; (3) Combine route candidates in each group and generate the final route. Before presenting the algorithm, we introduce the concepts of partitioning, query location group, and relevant data structures for maintaining partitionings and route candidates associated with each query location group.

3.4.1 Partitioning and query location group. We present the definition of query locations partitioning in Definition 6.

Definition 6: (*Partitioning of Query Locations*) A partitioning of query locations is denoted by $\mathcal{P}$. It consists of a set of disjoint query location groups $\{G_1, ..., G_n\}$. Each location group contains a subset of the query locations in O. We use $\mathbb{P}_{(k)}$ to denote k-set of query location partitionings. In particular, we have:

$$\mathbb{P}_{(k)} = \{\mathcal{P}_i \mid |\mathcal{P}_i| = k\} \quad (5)$$

□

Recall that once we find a qualified final route, which is generated from a set of qualified route candidates from each group (i.e., *group-wise route candidates*) in a particular partitioning, the algorithm terminates immediately. To enhance the search efficiency, we need to find a qualified final route as early as possible. To achieve this, we use a priority queue to maintain group-wise route candidates in each query location group $G_i \in \mathcal{P}$. In particular, it stores *group-wise route tuples* that are sorted in descending order of the similarity upper bound. Using this data structure, route candidates with high similarity scores, which are more likely to produce a qualified final route, are evaluated at first. Group-wise route candidates are stored as group-wise route tuples (Definition 7).

Definition 7: (*Group-wise route tuple*) A group-wise route tuple of route τ associated with query location group G is denoted by $G(\tau, P) = \langle e, P, ub\rangle$, which consists of three elements: an entry (identifier) e of route τ, a set of key-value pairs where the key is a query location (expansion center) in G and the value is an expansion point of τ, and the similarity upper bound ub of τ. Specifically, each pair $\langle o_i, p_j\rangle \in P$ satisfies the following conditions: (1) $o_i \in G$; (2) $o_i(\tau, p_j) \in C_i$; (3) $\forall(\langle o, p\rangle, \langle o', p'\rangle \in P)$ $(o \neq o')$. The value of ub is computed as follows:

$$G(\tau, P).ub = |O| - |G| + \sum_{\langle o_i, p_j\rangle \in P} e^{-d(o_i, p_j)}$$

□

We say that $G(\tau, P)$ is a *valid group-wise route tuple* if $G(\tau, P)$ satisfies Definition 7 and $G(\tau, P).ub \geq \theta$, where θ is the similarity threshold.

3.4.2 Route combination. We proceed to present how to combine two route candidates from two different query location groups as a route candidate in the next level (i.e., a route candidate associated with the union of the two query location groups). Specifically, given group-wise route tuples $G(\tau, P)$ and $G'(\tau', P')$, we need to be able to determine whether we can generate a qualified group-wise route tuple of group $G \cup G'$ based on $G(\tau, P)$ and $G'(\tau', P')$. We use $\tau^-(p)$ to denote the sub-route of τ starting from the beginning of τ and terminating at point p (p is a point in τ). Likewise $\tau^+(p)$ denotes the sub-route of τ starting from point p and terminating at the end of τ.

Intuitively, two route candidates τ and τ' can be combined into a route candidate in the next level if they have an intersection point (i.e., $\tau \cap \tau' \neq \emptyset$) and the intersection is a "transfer point", which lies between the expansion points of query locations in the one group and the expansion points of query locations in the other group. A next-level route candidate tuple is defined as follows.

Definition 8: (*Next-level route tuple*) Let $G(\tau, P)$ and $G'(\tau', P')$ be two group-wise route tuples of G and G', respectively. Routes τ and τ' intersect at point p_{in} and O denotes a set of query locations ($G \subseteq O, G' \subseteq O$). Route $\tau_s = \tau^-(p_{in}) + \tau'^+(p_{in})$ is a next-level route candidate for $S = G \cup G'$ and $S(\tau_s, P \cup P')$ is a next-level route tuple for G and G' if:

$$\forall(\langle o_i, p_j\rangle \in P, \langle o'_i, p'_j\rangle \in P')\ (p_j \in \tau^-(p_{in}) \wedge p'_j \in \tau'^+(p_{in}))$$

□

Having derived a next-level route tuple $S(\tau_s, P \cup P')$, we regard its corresponding route candidate τ_s as a new group-wise route candidate associated with $G \cup G'$. The route combination processing is performed iteratively in a bottom-up fashion until we find a qualified final route associated with group O.

Detailed algorithms and complexity analyses for generating group-wise route tuples, deriving a qualified final route by combining route candidates, are presented in Appendix, Section A.

4 GROUP-SPLIT PARALLEL SEARCH

4.1 Basic Idea

The FSPS algorithm maintains a set of route candidates for each query location. Because the similarity upper bound of each route candidate in FSPS only takes one query location into consideration (cf. Equation 4), which has low pruning power, the number of route candidates associated with each query location can be large. When combining route candidates in Algorithm 2, a large $|C_i|$ results in a very large (exponential) number of combination possibilities, which makes the combination process computationally expensive.

To improve the efficiency of RSL-Psc search, we need to decrease the number of route candidates and route candidate tuples. Hence, a more effective pruning strategy is required for the filtering of unqualified candidates. To achieve this, we propose the Group-Split Parallel Search (GSPS) algorithm. Unlike the FSPS algorithm, the GSPS algorithm does not need to maintain route candidates and tuples for each query location. Instead, we generate route candidates associated with each group directly. Thus, a similarity upper bound between a route and a query location set can be derived by computing the aggregate distances to query locations in a group rather than to a single query location. Consequently, the pruning power provided by GSPS is much larger than that provided by FSPS.

The high-level idea of the GSPS algorithm is as follows. First, we partition the set of query locations into $k+1$ groups, where $k \in [0, M]$, and we refine the value of k from the minimum to the maximum. For each group G_i ($i \in [1, k+1]$), we directly generate the route candidates and route tuples that are spatially close to G_i. This involves two steps: (1) Group-based network expansion (cf. Section 4.2) and (2) route candidate filtering (cf. Section B.2). The route candidate search in each group are performed in parallel. After that, we combine and evaluate the route candidates associated with location groups of the same partitioning. Note that the computation for each partitioning is also performed in parallel. Compared to FSPS, GSPS produces much fewer candidate sets, and it avoids the route candidate combination from query-location level to group level.

4.2 Group-based Network Expansion

Recall that in FSPS, network expansion is performed individually for each query location. When we parallelize the network expansion, we may only consider the minimum distance between a query location o and its nearest vertex p (i.e., $sd(o, p)$ in Equation 3) when calculating the similarity upper bound, which is a static value and has limited pruning effect. In GSPS, we introduce group-based network expansion that performs expansion for all query locations in a group simultaneously. In addition, given a group G, instead of storing a comparably loose and static similarity upper bound for each query location, we maintain a dynamic upper bound for each query location $o \in G$ that takes an aggregated distance between all query locations in G and their corresponding nearest vertices into consideration. With the group-based dynamic similarity upper bounds, we generate route candidates for G directly.

Next, we explain how to compute the group-based similarity upper bounds and how to perform route candidate pruning based on the upper bounds. Consider the example in Figure 4, where $G = \{o_1, o_2, o_3\}$ is a query location group in a partitioning $\mathcal{P}$, and τ_1, τ_2, τ_3, and τ_4 are routes. In each network expansion iteration, we select one of the query locations in G as an expansion center. For each query location o, we maintain its network distance to its current expansion point, which is denoted by $o.sd$. The location which has the minimum $o.sd$ value will be selected as the expansion center in the current iteration.

Table 1 presents the values of $o.sd$ in each iteration. At the beginning (Iter 0), the values of $o.sd$ for all $o \in G$ are 0. Assume that we select o_1 as the expansion center in the first iteration and p_1 is the first vertex scanned by the expansion from o_1. We update $o_1.sd$

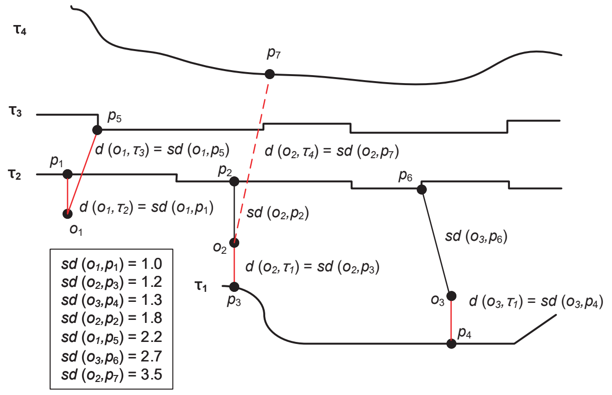


Figure 4: An example of group-based network expansion

Table 1: Update of $o.sd$

	Iter 1	Iter 2	Iter 3	Iter 4	Iter 5	Iter 6	Iter 7
$o_1.sd$	**1.0**	1.0	1.0	**2.2**	2.2	2.2	2.2
$o_2.sd$	0	**1.2**	1.2	1.2	**1.8**	1.8	**3.5**
$o_3.sd$	0	0	**1.3**	1.3	1.3	**2.7**	2.7

Table 2: Route Label Hash Map

Key	Value (route label set - $\langle o, p, o.sd \rangle$)
τ_1	$\langle o_2, p_3, 1.2\rangle$, $\langle o_3, p_4, 1.3\rangle$
τ_2	$\langle o_1, p_1, 1.0\rangle$, $\langle o_2, p_2, 1.8\rangle$,$\langle o_3, p_6, 2.7\rangle$
τ_3	$\langle o_1, p_5, 2.2\rangle$

to be 1.0. Because $p_1 \in \tau_2$, we generate the label $\langle o_1, p_1, 1.0\rangle$ for τ_2 and add it to the *route label hash map* maintained by group G (cf. Table 2). The route label hash map is used during route candidate filtering (cf. Section B.2). In the second iteration, we select o_2 as the expansion center (after the 1st iteration, $o_2.sd = o_3.sd$, so either o_2 or o_3 can be selected as the expansion center), and p_3 is the first vertex scanned by the expansion from o_2. Thus, we set $o_2.sd = 1.2$. Likewise, since $p_3 \in \tau_1$, we generate a label $\langle o_2, p_3, 1.2\rangle$ for τ_1 and add it to the route label hash map. We continue the iterative process until we reach the similarity upper bound of G. Theorem 1 explains that the resulting pruning is safe.

Theorem 1: Given a query location set O, a similarity threshold θ, and a group of query locations G where $G \subset O$, group-based network expansion can be stopped and all unexplored routes can be safely pruned when:

$$|O| - |G| + \sum_{o \in G} e^{-o.sd} < \theta \tag{6}$$

Proof. Let τ_u be a route that is unexplored during group-based network expansion. The Dijkstra expansion has the property that, $\forall (o_i \in G)\ (d(o_i, \tau_u) \geq o_i.sd)$. As a result, $\forall (o_i \in G)\ (e^{-d(o_i,\tau_u)} \leq e^{-o_i.sd})$. Because $d(o, \tau_u)$ is non-negative, $e^{-d(o,\tau_u)}$ cannot exceed 1. Then we have: $\forall (o_j \in O \setminus G)\ (e^{d(o_j,\tau_u)} \leq 1)$. Consequently, if $|O| - |G| + \sum_{o \in G} e^{-o.sd}$ is smaller than θ, $\sum_{o \in O} e^{-d(o,\tau_u)}$ must be smaller than θ. This completes the proof. □

Detailed algorithms for group-based network expansion and route candidate filtering are presented in Sections B.1 and B.2, respectively.

5 EXPERIMENTAL STUDY

We report on experiments with real road networks routes and Points-of-Interest (POI) data sets that offer insight into the efficiency and scalability of the proposed algorithms.

5.1 Experiment Settings

5.1.1 Data sets. We use two road networks, namely the Beijing Road Network (BRN) and the New York Road Network (NRN)[1], which contain 28,342 vertices and 27,690 edges, and 95,581 vertices and 260,855 edges, respectively. The corresponding network graphs are stored and indexed by adjacency lists. In BRN, we use a real taxi trajectory data set collected by the T-drive project [19], while in NRN, we use a real taxi data set from New York [10]. Each item in the data set contains pick-up and drop-off locations of a taxi. We derive the shortest path from the pick-up location to the drop-off location and regard it as a route. The T-drive taxi trajectory data set contains 800K trajectories and 300K POIs (each POI has a spatial coordinate with latitude and longitude), while the New York taxi data set contains 700M routes. In NRN, we use a real POI data set that contains 19,918 POIs in New York City [2]. For NRN, the POIs may not match the trajectory points. So we map each POI in NRN to its nearest road network vertex.

5.1.2 Query location sets. A query location set O is generated as follows: First, we plot n circular query selector regions with radius r and place each selector region at a random position in the underlying space. Next, we randomly select $|O|/n$ POIs from each selector region. The selected POIs constitute the query location set. In the experiments, we evaluate the parameters n and r.

5.1.3 Implementations. In the experiments, the road network graphs, routes, and POIs are memory resident. All algorithms are implemented in Java and run on a cluster with 10 data nodes. Each node is equipped with two Intel Xeon Processors E5-2620 v3 (2.4GHz) and 128GB RAM. Unless stated otherwise, experimental results are averaged over 200 and 50 independent trials using different query location sets for effectiveness (Section 5.2) and efficiency evaluations, respectively. The performance metrics are runtime and the number of route visits. The number of route visits is used as a metric because it reflects the number of data accesses. In multi-threaded executions, the total runtime is the maximum runtime among all individual threads.

Trajectories in T are selected randomly from the real data sets. We evaluate the following three methods:

- FSPS: Fully-Split Parallel Search (Section 3);
- GSE+CTF: Group-Split Parallel Search (GSPSExpansion + CTFilter) (Section 4);
- GSE: Group-Split Parallel Search without CTFilter (GSPSExpansion only) (Section 4.2).

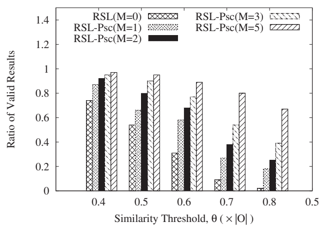
When evaluating the number of route visits, we do not report the performance of GSE+CTF because GSE and GSE+CTF incur the same numbers of route visits. The parameter settings are listed in Table 3.

[1]https://publish.illinois.edu/dbwork/open-data/
[2]https://data.cityofnewyork.us/City-Government/Points-Of-Interest/rxuy-2muj

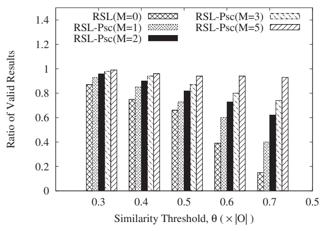
Table 3: Parameter Settings

	BRN	**NRN**
Cardinality of routes $\|T\|$	100,000–500,000 / default 300,000	2,000,000–10,000,000 / default 2,000,000
Cardinality of query location set $\|O\|$	8–12 / default 10	8–12 / default 10
Similarity threshold θ	0.4–0.8 ($\times\|O\|$) / default 0.6 ($\times\|O\|$)	0.3–0.7 ($\times\|O\|$) / default 0.5 ($\times\|O\|$)
Maximum number of transfers M	1–5 / default 3	1–5 / default 3
Radius of query location selector region r	1–9 km / default: detailed in experiments	1–9 km / default: detailed in experiments
Number of query location selector regions n	1–4 / default 2	1–4 / default 3
Thread count m	24–120 / default 24	24–120 / default 24

5.2 Result Quality Analysis



(a) BRN



(b) NRN

Figure 5: Ratio of valid results

First, we evaluate the ratio of queries that return valid results (i.e., qualified final routes) as we vary the similarity threshold θ. Here, RSL (M=0) in Figure 5 denotes an RSL query without a route combination. RSL-Psc (M=x) denotes an RSL-Psc query with x route combinations. We can see that all methods exhibit a decreasing trend regarding the ration of valid results as we increase the similarity threshold. In particular, the performance of RSL decreases significantly as we increase the similarity threshold. We also find that the decreasing trend becomes less significant when the maximum number of combinations (M) becomes larger. Specifically, when we set θ to 0.8 in BRN, RSL only has 4 out of 200 queries that return valid results. In contrast, RSL-Psc (M=3) and RSL-Psc (M=5) have 78 and 135 out of 200 queries that return valid results, respectively. Similarly, when we set θ to 0.7 in NRN, RSL has 31 out of 200 queries that return valid results. In contrast, RSL-Psc (M=3) and RSL-Psc (M=5) have 149 and 186 out of 200 queries that return valid results, respectively. As a result, the RSL-Psc query, even with a small number of route combinations, demonstrates superiority over the RSL query (without route combination) with regards to the probability of returning a valid result route.

5.3 Pruning Effectiveness

First, we investigate the pruning achieved by the three methods with default parameter settings. The experimental results are shown in Tables 4. Specifically, the *pruning rate* (*PR*) and *proportion of candidates* (*PC*) are defined as follows.

Table 4: Pruning Effect

	FSPS	GSE	GSE+CTF
PR in BRN	0.29	0.81	0.91
PC in BRN	0.71	0.19	0.09
PR in NRN	0.34	0.84	0.93
PC in NRN	0.66	0.26	0.07

$$PC_{FSPS} = \frac{\#DistinctR(C)}{|T|} \qquad PC_{GSPS} = \frac{\sum_{\mathcal{P}_i \in \mathbb{P}} \#DistinctR(\mathcal{P}_i)}{|\mathbb{P}| \times |T|}$$

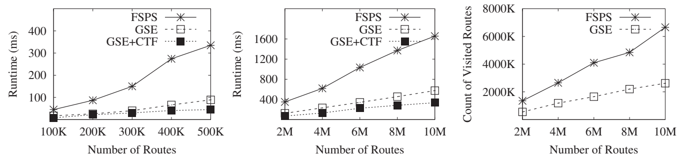
$$PR_{FSPS} = 1 - PC_{FSPS} \quad PR_{GSPS} = 1 - PC_{FSPS}$$

Here, the pruning effectiveness of FSPS is computed by PC_{FSPS} and PR_{FSPS}, while the pruning effectivenesses of GSE and GSE+CTF are computed by PC_{GSPS} and PR_{GSPS}. The numerator of PC_{FSPS} (i.e., $\#DisdinctR(C)$) denotes the number of distinct routes in C, and $|T|$ is the cardinality of the route set. The value of $\#DistinctR(\mathcal{P}_i)$) in PC_{GSPS} is the number of distinct route candidates in all groups of partition $\mathcal{P}_i$. By comparing the pruning rate of GSE+CTF to that of FSPS, we see that the pruning effectiveness is improved by approximately a factor of 3 when using the group-based dynamic similarity upper bounds based on the aggregated distance between all query locations in each group (cf. Section 4.2). In addition, the comparison between GSE+CTF and GSE suggests that the route candidate filtering algorithm (CTFilter) is capable of improving the pruning effectiveness by a factor of 1.1.

5.4 Evaluation of Query Performance

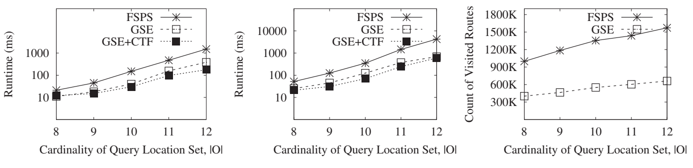
Effect of the number of routes: Figure 6 presents the performance of the algorithms when varying the number of routes $|T|$. Intuitively, a larger $|T|$ causes more routes to be processed and yields a larger search space. As a result, both the CPU time and the count of visited routes are expected to increase for all three algorithms. Figures 6(a) and 6(b) shows that GSE+CTF outperforms FSPS by approximately a factor of 6 in both BRN and NRN regarding the CPU time. In particular, the group-based network expansion algorithm (cf. Section 4.2) in GSE is able to improve the CPU-time performance by a factor of 3–4 compared with FSPS, and the route candidate filtering algorithm (CTFilter) can further improve the efficiency of GSE by a factor of 1.5. It is worth noting that the CPU time is not fully aligned with the count of visited routes. Specifically, the performance discrepancy between FSPS and GSE in terms of route counts is less significant than that in terms of CPU time. The reason can be explained as follows. In FSPS, each visited route is regarded as a route candidate of a query location, and it is evaluated in the combination process. In GSE, a large number of visited routes are pruned by the group-based dynamic similarity upper bounds, so they are not evaluated in the combination process.

Effect of query location set cardinality: Figure 7 shows the effect of varying the size of query location set $|O|$ on the efficiency of the algorithms. A larger $|O|$ implies: (1) A larger search space with more routes to be accessed and evaluated; (2) A larger number of possible partitions and split-and-combine sub-tasks (i.e., the number of split-and-combine sub-tasks is $\sum_{k=0}^{M} S(|O|, k+1)$). Thus, when we increase the number of query locations in O, more CPU time and route visits occur. It is worth mentioning that when $|O|$ is increased



(a) BRN-time (b) NRN-time (c) NRN-count

Figure 6: Effect of the number of routes



(a) BRN-time (b) NRN-time (c) NRN-count

Figure 7: Effect of $|O|$

from 8 to 12, the value of $\sum_{k=0}^{M} S(|O|, k+1)$ $(M = 3)$ is increased by more than 300 times. However, according to Figures 7(a) and 7(b), the CPU time only increases by 30–100 times for all algorithms. This is because some qualified routes require only 0, 1, or 2 transfers, and that once we find a qualified route, the search process terminates.

Effect of similarity threshold θ: This set of experiments investigates the effect of similarity threshold θ. Figure 8 shows the results when we vary the similarity threshold θ. Increasing the value of θ has the following two effects on the performance: (1) Based on Equation 4 and Theorem 1, a larger value of θ leads to higher pruning effectiveness, which may improve the efficiency. (2) A larger value of θ may postpone the termination of the algorithms. Recall that while combining routes, we first generate the routes with the minimum number of combinations (transfers), and that once we find a qualified route, it is unnecessary to consider larger numbers of combinations. Hence, when we increase θ, it is less likely that we will be able to generate a qualified route with few combinations. Such effect may deteriorate the efficiency. Compared to Effect (2), Effect (1) is negligible. As a result, all algorithms exhibit increasing CPU time and route visits as we increase the value of θ.

Effect of maximum number of transfers M: We proceed to evaluate the effect of varying the maximum number of route transfers (M). From Figures 9(a) and 9(b), we find that when we increase M, the CPU time increases for all algorithms. Specifically, when the value of M reaches 4 in NRN, the subsequent increase in CPU time is modest. The reason is that when M is set to be 5 in NRN, most of the trials are returned with qualified final routes with no more than 4 transfers. Additionally, Figure 9(c) suggests that the performance of route counts is relatively consistent as we increase M.

Effect of the radius of query location selector region: Figure 10 shows the effect of varying the radius of the query location selector region. We find that both CPU time and route visits exhibit slight or moderate increasing trends for all algorithms when we increase the radius of the selector region from 1 km to 9 km. The reason is that when we apply a large query location selector region, the query locations in O are distributed increasingly widely in the underlying space, which increases the number of route visits during network expansion.

Effect of the number of query location selector regions: Figure 11 covers the effect of varying the number of query location selector regions. More query regions implies that more expansion centers must be processed, which increases the search space and the number of route visits. Thus, the CPU time and the count of visited routes for all three algorithms increase with the number of query location selector regions.

Effect of thread counts: We study the effect of thread count on the efficiency of the algorithms using large route data sets ($|T|$ = 500K for BRN and $|T|$ = 10M for NRN). The results in Figure 12 show that GSE+CTF outperforms FSPS by a factor of 6–8 in term of runtime and outperforms GSE by 30%–60% in term of runtime. In BRN, when we set the thread counts to 120, GSE+CTF is able to solve the RSL-Psc problem over a collection of 500K routes in 15 milliseconds, while in NRN, GSE+CTF is able to solve the RSL-Psc problem with 10M routes in 100 milliseconds. When we increase the thread count from 24 to 120 (5 times), the runtime of GSE+CTF and GSE are improved by a factor of around 3, while the runtime performance of FSPS is improved by 2.8 and 2.1 in BRN and NRN, respectively.

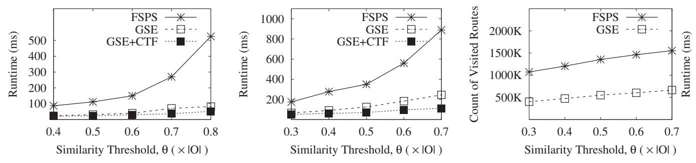
6 RELATED WORK

Existing studies related to the RSL-Psc problem can be classified into two categories: Location-to-trajectory search and location-based route recommendations.

Location-to-route search: Location-to-route search aims at retrieving trajectories who have the highest relevances to query arguments [4, 12, 15, 21]. In particular, the relevancy functions may contain spatial [1], temporal [8], textual [12][21], and density elements. The resulting queries are useful in many popular applications including travel planning, carpooling, friend recommendation in social networks, and location-based services in general.

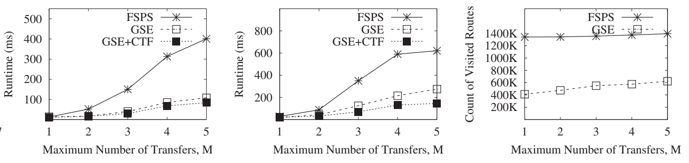
According to the types of trajectory query arguments, we further classify existing studies regarding location-to-trajectory search into two sub-categories: (1) Trajectory search based on a single location; (2) trajectory search based on multiple locations. Zheng et al. [22] extend the single-point trajectory query to cover spatial and textual domains and propose the TkSK query, which retrieves the trajectories that are spatially close to the query point and also meet semantic requirements defined by the query. For trajectory search based on multiple locations, the query takes a set of locations as argument and returns a trajectory that connects or is close to the query locations according to specific metrics. The concept of trajectory search by locations (TSL) was first proposed by Chen et al. [1]. The main difference between RSL-Psc and the problem of location-based trajectory search studied by existing work is that RSL-Psc returns a route by combining a set of connected trajectories. In contrast, existing location-based trajectory queries return a single pre-existing trajectory or a list of pre-existing trajectories.

Location-based route recommendations: Given a set of locations (e.g., POIs, taxi locations), the location-based route recommendation problem aims to derive a new route based on the locations and user preferences. Ge et al. [5] and Ye et al. [16, 17] study the mobile sequential recommendation problem that outputs an optimal routes with minimum potential travel distance to a taxi driver's next potential passenger. In particular, Ye et al. [18] first



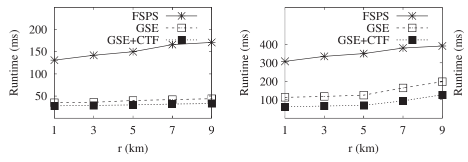
(a) BRN-time (b) NRN-time (c) NRN-count

Figure 8: Effect of θ



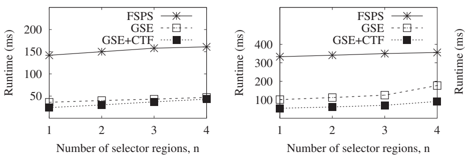
(a) BRN-time (b) NRN-time (c) NRN-count

Figure 9: Effect of M



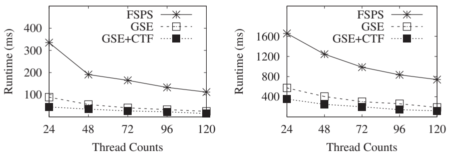
(a) BRN-time (b) NRN-time

Figure 10: Effect of r



(a) BRN-time (b) NRN-time

Figure 11: Effect of n



(a) BRN-time (b) NRN-time

Figure 12: Effect of thread counts

study the multiple mobile sequential recommendation problem that generates optimal routes for a group of taxis with different locations. Another area of related studies is travel itinerary recommendation (e.g., [7, 14]). Specifically, it takes a set of user-specified POIs and constraints as input to generate an itinerary through a subset of POIs with a specific starting and ending POI that can be completed within a certain time. Additionally, Yang et al. [2] recommend the shortest route to users based on existing trajectories by considering multiple costs. However, the results generated by these proposals are new individual routes, while the results of RSL-Psc are combinations of existing trajectories.

7 CONCLUSIONS

We propose and study RSL-Psc problem, namely parallel split-and-combine approach to enable route search by locations. To answer the RSL-Psc query, we develop two parallel search algorithms: Fully-Split Parallel Search (FSPS) and Group-Split Parallel Search (GSPS). Specifically, we divide the route split-and-combine task into $\sum_{k=0}^{M} S(|O|, k+1)$ sub-tasks, where M is the maximum number of combinations and $S(\cdot)$ is the Stirling number of the second kind. In each sub-task, we use network expansion to explore the spatial network and exploit spatial similarity bounds for pruning. The algorithms split candidate routes into sub-routes and combine them to construct new routes. The sub-tasks are independent of each other and are performed in parallel. Extensive experiment with real data demonstrates that our proposed RSL-Psc query is much more likely to return a valid result compared with the PSL query without route combination. In addition, FSPS and GSPS algorithms are capable of achieving high efficiency and scalability on massive route data.

Acknowledgements: This work (Bin Yao) was supported by the NSFC (61872235, 61729202, 61832017, U1636210) and the National Key Research and Development Program of China (2018YFC1504504). Additionally, Zhiwei Zhang is supported by GRF (12201518, 12232716, 12258116) and NSFC (61602395).

REFERENCES

[1] Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, Yu Zheng, and Xing Xie. 2010. Searching trajectories by locations: an efficiency study. In *SIGMOD*. 255–266.
[2] Jian Dai, Bin Yang, Chenjuan Guo, and Zhiming Ding. 2015. Personalized route recommendation using big trajectory data. In *ICDE*. 543–554.
[3] E. W. Dijkstra. 1959. A note on two problems in connection with graphs. *Numerische Math* 1 (1959), 269–271.
[4] E. Frentzos, K. Gratsias, and Y. Theodoridis. 2007. Index-based most similar trajectory search. In *ICDE*. 816–825.
[5] Yong Ge, Hui Xiong, Alexander Tuzhilin, Keli Xiao, Marco Gruteser, and Michael J. Pazzani. 2010. An energy-efficient mobile recommender system. In *KDD*. 899–908.
[6] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. 47–57.
[7] Kwan Hui Lim, Jeffrey Chan, Shanika Karunasekera, and Christopher Leckie. 2017. Personalized Itinerary Recommendation with Queuing Time Awareness. In *SIGIR*. 325–334.
[8] Shuo Shang, Lisi Chen, Zhewei Wei, Christian S. Jensen, Ji-Rong Wen, and Panos Kalnis. 2016. Collective Travel Planning in Spatial Networks. *IEEE Trans. Knowl. Data Eng.* 28, 5 (2016), 1132–1146.
[9] Shuo Shang, Lisi Chen, Zhewei Wei, Christian S. Jensen, Kai Zheng, and Panos Kalnis. 2017. Trajectory Similarity Join in Spatial Networks. *PVLDB* 10, 11 (2017), 1178–1189.
[10] Shuo Shang, Lisi Chen, Zhewei Wei, Christian S. Jensen, Kai Zheng, and Panos Kalnis. 2018. Parallel trajectory similarity joins in spatial networks. *VLDB J.* 27, 3 (2018), 395–420.
[11] Shuo Shang, Lisi Chen, Kai Zheng, Christian S. Jensen, Zhewei Wei, and Panos Kalnis. 2019. Parallel Trajectory-to-Location Join. *IEEE Trans. Knowl. Data Eng.* 31, 6 (2019), 1194–1207.
[12] Shuo Shang, Ruogu Ding, Bo Yuan, Kexin Xie, Kai Zheng, and Panos Kalnis. 2012. User oriented trajectory search for trip recommendation. In *EDBT*. 156–167.
[13] Shuo Shang, Ruogu Ding, Kai Zheng, Christian S. Jensen, Panos Kalnis, and Xiaofang Zhou. 2014. Personalized trajectory matching in spatial networks. *VLDB J.* 23, 3 (2014), 449–468.
[14] Kendall Taylor, Kwan Hui Lim, and Jeffrey Chan. 2018. Travel Itinerary Recommendations with Must-see Points-of-Interest. In *WWW*. 1198–1205.
[15] Kexin Xie, Ke Deng, and Xiaofang Zhou. 2009. From trajectories to activities: a spatio-temporal join approach. In *LBSN*. 25–32.
[16] Zeyang Ye, Keli Xiao, and Yuefan Deng. 2018. A Unified Theory of the Mobile Sequential Recommendation Problem. In *ICDM*. 1380–1385.
[17] Zeyang Ye, Keli Xiao, Yong Ge, and Yuefan Deng. 2019. Applying Simulated Annealing and Parallel Computing to the Mobile Sequential Recommendation. *IEEE Trans. Knowl. Data Eng.* 31, 2 (2019), 243–256.
[18] Zeyang Ye, Lihao Zhang, Keli Xiao, Wenjun Zhou, Yong Ge, and Yuefan Deng. 2018. Multi-User Mobile Sequential Recommendation: An Efficient Parallel Computing Paradigm. In *KDD*. 2624–2633.
[19] Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. 2013. T-Drive: Enhancing Driving Directions with Taxi Drivers' Intelligence. *IEEE Trans. Knowl. Data Eng.* 25, 1 (2013), 220–232.
[20] Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang. 2010. T-drive: driving directions based on taxi trajectories. In *ACM SIGSPATIAL*. 99–108.
[21] Kai Zheng, Shuo Shang, Nicholas Jing Yuan, and Yi Yang. 2013. Towards efficient search for activity trajectories. In *ICDE*. 230–241.
[22] Kai Zheng, Bolong Zheng, Jiajie Xu, Guanfeng Liu, An Liu, and Zhixu Li. 2016. Popularity-aware spatial keyword search on activity trajectories. *World Wide Web* 19, 6 (2016), 1–25, online first.

A ALGORITHM FOR COMBINING ROUTE CANDIDATES IN FSPS

Algorithm 2: FSPSCombination

```
   Data: Route candidate tuple set $C$, similarity threshold $\theta$, combination
         threshold $M$
   Result: Final result route
1  $k \leftarrow 0$;
2  while $k \leq M$ do
3      for each $\mathcal{P}$ in $\mathbb{P}_{(k+1)}$ do
4          for each $G$ in $\mathcal{P}$ do
5              $G.candidates \leftarrow \emptyset$;
6              for each $\tau$ s.t. $\forall(o_i \in G)\ (o_i(\tau, \cdot) \in C_i)$ do
7                  for each valid $G(\tau, P)$ do
8                      $G.candidates$.add($G(\tau, P)$); // Definition 7

9          for each group sequence $\mathcal{X}$ of $\mathcal{P}$ do
10             for each $G_i^{(0)}$ in $\mathcal{X}$ do
11                 Initialize $G_i^{(0)}$;
12             $j \leftarrow 0$;
13             while $|\mathcal{X}| > 1$ do
14                 $l, s \leftarrow 1$;
15                 while $l \leq k$ do
16                     if $l = k$ then
17                         $G_s^{(j+1)} \leftarrow G_l^{(j)}$;
18                     else
19                         $G_s^{(j+1)} \leftarrow G_l^{(j)} \cup G_{l+1}^{(j)}$;
20                         $G_s^{(j+1)}.candidates \leftarrow$ NextLevelRoute($G_l^{(j)}$,$G_{l+1}^{(j)}$);
21                         if $|G_s^{(j+1)}| = |O|$ and $G_s^{(j+1)}.candidates \neq \emptyset$
                           then
22                             return $G_s^{(j+1)}.candidates$.top();
23                         Replace $G_l^{(j)}$ and $G_{l+1}^{(j)}$ by $G_s^{(j+1)}$ in $\mathcal{X}$;
24                         $l \leftarrow l + 2$;
25                         $s \leftarrow s + 1$;
26                 $j \leftarrow j + 1$;
27     $k \leftarrow k + 1$;
```

Algorithm 3: NextLevelRoute(G, G')

```
   Data: Groups $G$ and $G'$, similarity threshold $\theta$
   Result: Next-level route tuples of $G \cup G'$
1  $S \leftarrow G \cup G'$;
2  $S.candidates \leftarrow \emptyset$;
3  for each $G(\tau, P) \in G.candidates$, $G(\tau', P') \in G'.candidates$ do
4      if $\tau \cap \tau' \neq \emptyset$ then
5          for each $p_i \in \tau \cap \tau'$ do
6              $\tau_l \leftarrow \tau^-(p_i) + \tau'^+(p_i)$;
7              $\tau_r \leftarrow \tau'^-(p_i) + \tau^+(p_i)$;
8              if $\tau_l$ is a next-level route and $|G \cup G'| < |O|$ then
9                  $S.candidates$.add($S(\tau_l, P \cup P')$);
10             else if $\tau_l$ is a next-level route and $Sim(O, \tau_l) \geq \theta$ then
11                 $S.candidates$.add($S(\tau_l, P \cup P')$);
12                 return $S.candidates$;
13             if $\tau_r$ is a next-level route and $|G \cup G'| < |O|$ then
14                 $S.candidates$.add($S(\tau_r, P \cup P')$);
15             else if $\tau_r$ is a next-level route and $Sim(O, \tau_r) \geq \theta$ then
16                 $S.candidates$.add($S(\tau_r, P \cup P')$);
17                 return $S.candidates$;
18 return $S.candidates$;
```

Algorithm 2 presents the pseudo code of generating group-wise route tuples and deriving a qualified final route by combining route candidates. We partition the query location set O into $k + 1$ subsets, where $k \in [0, M]$, and we evaluate each possible partitioning starting from partitioning with one group to partitionings with $k + 1$ groups. For each possible partitioning $\mathcal{P}$, we generate the group-wise route tuples associated with each group G (lines 4–8). First, we initialize $G.candidates$, which stores all group-wise route tuples of G (line 5). Next, for each τ that were scanned by every query locations in G during Dijkstra expansion, we generate its group-wise route tuples based on Definition 7 (lines 6–8). After having group-wise route tuples of all groups in partitioning $\mathcal{P}$, we evaluate each possible sequence of groups $\mathcal{X}$ and for each sequence we combine route candidates in each group in a bottom-up fashion until all groups are combined (lines 9–26). Here, i in $G_i^{(j)}$ is the group index and j in $G_i^{(j)}$ denotes the level of the group. In particular, for each sequence of group we initialize the group-wise route tuples of each group $G_i^{(0)}$ from level 0 (lines 10–11). If $|\mathcal{X}| > 1$, which means that the groups in sequence $\mathcal{X}$ can be combined, we proceed to combine route candidates in groups from the lowest level. Starting from level $j = 0$, we combine route candidates associated with two adjacent groups (i.e., $G_l^{(j)}$ and $G_{l+1}^{(j)}$) and generate corresponding next-level route candidate tuples (lines 19–20). Specifically, we derive the union of groups $G_l^{(j)}$ and $G_{l+1}^{(j)}$ (i.e., $G_s^{(j+1)}$) (line 19) and generate route candidates associated with $G_s^{(j+1)}$ by calling NextLevelRoute function (Algorithm 3) (line 20). If the cardinality of $G_s^{(j+1)}$ equals the cardinality of the query location set O, this means that we have combined all groups and the route candidates are associated with all locations in the query location set. Thus, we can return any route candidate as a result (lines 21–22). Otherwise, we need to update $\mathcal{X}$ by merging $G_l^{(j)}$ and $G_{l+1}^{(j)}$ (line 23).

Algorithm 3 presents the pseudo code for combining the route candidate tuples associated with two adjacent groups and generating corresponding next-level route candidate tuples. After initialization (lines 1–2), we evaluate each pair of routes from G and G', respectively, and check if we can generate a next-level route candidate from the route pair τ and τ'. Based on Definition 8, if τ and τ' can generate a next-level route candidate, they must have at least one intersection point (line 4). For each intersection point p_i, we consider two potential combinations of sub-routes, $\tau_{p_i}^- + \tau'^+(p_i)$ and $\tau'^-(p_i) + \tau^+(p_i)$, and we check whether they are qualified for combination based on Definition 8 (lines 8–17). Qualified next-level route candidate tuples are added to the route candidate tuple set associated with the union of G and G'. In particular, if the union of G and G' equals to the query location set O, each route candidate associated with $S = G \cup G'$ is considered to be a final route candidate. Here, we check whether the similarity between the route and query location set satisfies the similarity threshold θ. If so, we regard the route as a qualified final route and return $S.candidates$.

Time Complexity: The time complexity of generating route tuples of each query location in O (i.e., FSPSCandidates) is $O((|V|\log|V| + |E|) \cdot |O| \cdot |T_v|)$. Specifically, $(|V|\log|V| + |E|)$ is the complexity of the Dijkstra expansion for each query location, $|V|$ is the number of vertices, and $|E|$ is the number of edges. Further, $|T_v|$ denotes the average number of routes passing each vertex. The time complexity of running FSPSCombination on each partitioning can be approximated as $O(k \cdot |\tau_o|^{\frac{|O|}{k}} + |T_G|^2 \cdot k!)$, where k denotes the number of groups in the partitioning, $|\tau_o|$ is the average number of route tuples of each route associated with each query location, and $|T_G|$ is the average number of group-wise route tuples of each group.[3] Here, k and $|O|$ are much smaller than $|T_v|$ and $|T_G|$.

B ALGORITHM OF GSPS

B.1 Algorithm of Network Expansion

Algorithm 4 presents the pseudo code of group-based network expansion. First, we initialize route label hash map H and the value of $o_i.sd$ for each $o_i \in G$ (lines 1–3). Next, we check whether the expansion can be terminated based on Theorem 1 (line 4). If not, we start the current iteration of network expansion (lines 5–13). Specifically, we select the query location that has the minimum value of $o.sd$ in G for performing expansion (i.e., o_{min}) (line 5). Here, p is the next vertex scanned by the expansion from o_{min} (line 6). Then we update $o_{min}.sd$ to be $sd(o_{min}, p)$, which is the shortest network distance between o_{min} and p (line 7). After that, we scan and evaluate the routes that pass through p (lines 8–13). In particular, for each route τ passing through p, we retrieve its route label set (i.e., L) in H (line 9). If L is null, τ has never been scanned. So we need to add a new key-value pair of τ into H. The key is the entry of τ and the value is a new route label set containing label $\{\langle o_{min}, p, o_{min}.sd\rangle\}$ (lines 10–11). If L is not null, it denotes that τ was scanned before. Here, we just insert a new label ($\{\langle o_{min}, p, o_{min}.sd\rangle\}$) into L (lines 12–13). When the expansion terminates, we return the hash map H as the result (line 14). Elements in H are considered to be route candidates associated with G.

Algorithm 4: GSPSExpansion

```
   Data: Query location group $G$, cardinality of query location set $|O|$, route set $T$,
         similarity threshold $\theta$
   Result: Route label hash map $H$
1  $H \leftarrow \emptyset$;
2  for each $o_i$ in $G$ do
3      $o_i.sd \leftarrow 0$;
4  while $|O| - |G| + \sum_{o_i \in G} e^{-o_i.sd} \geq \theta$ do
5      $o_{min} \leftarrow$ the $o$ with the minimum $o.sd$ in $G$;
6      $p \leftarrow$DijkstraExpansion($o_{min}$).next();
7      $o_{min}.sd \leftarrow sd(o_{min}, p)$;
8      for each $\tau$ s.t. $p \in \tau$ do
9          $L \leftarrow H$.get($\tau$);
10         if $L$ is null then
11             $H$.put($\tau$, $\{\langle o_{min}, p, o_{min}.sd\rangle\}$);
12         else
13             $L$.add($\{\langle o_{min}, p, o_{min}.sd\rangle\}$);
14 return $H$;
```

B.2 Route Candidate Filtering

Recall that the GSPSExpansion algorithm generates route candidates (H) for group G. However, some routes in H can be eliminated based on the route label set associated with each route in H. In particular, while evaluating whether a new route τ is a qualified route candidate for G during iteration i, the corresponding similarity upper bound (i.e., $|O| - |G| + \sum_{o_i \in G} e^{-o_i.sd}$) is computed based on the value of $o_i.sd$ in iteration i. As we continue the iterative process, the value of $o_i.sd$ will increase, which makes the value of $|O| - |G| + \sum_{o_i \in G} e^{-o_i.sd}$ decrease. Therefore, it is possible that a route τ added to H during iteration i can be pruned based on Theorem 1 during iteration i+n.

To address the problem, we develop a route candidate filtering algorithm, CTFilter, that takes H as input and evaluates each $\tau \in H$ based on the final value of $o_i.sd$ produced by the GSPSExpansion algorithm.

Algorithm 5: CTFilter

```
   Data: Route label hash map $H$, the final values of $o_i.sd$ for each $o_i \in G$,
         cardinality of query location set $|O|$, route set $T$, similarity threshold $\theta$
   Result: $G.candidates$
1  $G.candidates \leftarrow \emptyset$;
2  for each $\tau$ in $H$ do
3      $L \leftarrow H$.get($\tau$);
4      $S \leftarrow \emptyset$;   $P \leftarrow \emptyset$;
5      $a \leftarrow 0$;
6      for each label $l \in L$ do
7          $o \leftarrow$ query location in $l$;
8          $sd \leftarrow o.sd$ in $l$;
9          $p \leftarrow p$ in $l$;
10         $S$.add($o$);
11         $P$.add($\langle o, p\rangle$);
12         $a \leftarrow a + e^{-sd}$;
13     if $|O| - |G| + a + \sum_{o_i \in G \setminus S} e^{-o_i.sd} \geq \theta$ then
14         $G.candidates$.add($G(\tau, P)$);
15 return $G.candidates$;
```

After initializing $G.candidates$, we evaluate each τ in H. First, we retrieve the route label set L associated with τ (line 3). Next, sets S, P, and variable a are initialized (lines 4–5). Specifically, S stores the query locations in L, P stores the $\langle o, p\rangle$ pairs (cf. P in Definition 7), and a records the aggregated value of $o_i.sd$ ($o_i \in S$). Then we visit each label l in L and acquire the query location o and its corresponding p and $o.sd$ in l (lines 7–9). We update S and P by inserting o and $\langle o, p\rangle$, respectively, and update a by adding the value of $o.sd$ (lines 10–12). Next, we calculate the up-to-date similarity upper bound of τ. In expression $|O| - |G| + a + \sum_{o_i \in G \setminus S} e^{-o_i.sd}$, a denotes the aggregated similarity score contributed by route labels of τ (i.e., L), and $\sum_{o_i \in G \setminus S} e^{-o_i.sd}$ computes the similarity score contributed by the query locations in G that are not stored in L. If the upper bound is no less than θ, we add the group-wise route tuples of τ into $G.candidates$. When having completed the scan of H, we return $G.candidates$ as the result.

The algorithm for combining route candidates in GSPS is similar to Algorithm 2. The only difference is that we do not need to generate the group-wise route tuples associated with each group G (cf. Algorithm 2 lines 4–8) because we have done it in Algorithm 4.

Time Complexity: The time complexity of generating route tuples of each query location in O (i.e., GSPSExpansion) is $O(2^{|O|} \cdot (|V| \cdot \log|V| + |E|) \cdot |T_v|)$, where $2^{|O|}$ denotes the number of unique groups that can be generated from query location set O. The time complexity of route combination on each partitioning can be approximated as $O(|T_G|^2 \cdot k!)$. The notation was explained at the end of Section A.

[3]We omit the detailed reduction and approximation of the time complexity due to the space limitation.