Scheduling Multiple Trips for a Group in Spatial Databases

Roksana Jahan[1], Tanzima Hashem[2], Sukarna Barua[3]
[1,2,3]Department of CSE, Bangladesh University of Engineering and Technology, Bangladesh
{[1]munia_064@yahoo.com} {[2]tanzimahasem, [3]sukarnabarua@cse.buet.ac.bd}

ABSTRACT

Planning user trips in an effective and efficient manner has become an important topic in recent years. In this paper, we introduce Group Trip Scheduling (GTS) queries, a novel query type in spatial databases. Family members normally have many outdoor tasks to perform within a short time for the proper management of home. For example, the members of a family may need to go to a bank to withdraw or deposit money, a pharmacy to buy medicine, or a supermarket to buy groceries. Similarly, organizers of an event may need to visit different points of interests (POIs) such as restaurants and shopping centers to perform many tasks. Given source and destination locations of group members, a GTS query enables a group of n members to schedule n individual trips such that n trips together visit required types of POIs and the total trip distance of n group members is minimized. The trip distance of a group member is measured as the distance between her source to destination via the POIs. We develop an efficient approach to process GTS queries for both Euclidean space and road networks. The number of possible combinations of trips among group members increases with the increase of the number of POIs that in turn increases the query processing overhead. We exploit geometric properties to refine the POI search space and prune POIs to reduce the number of possible combinations of trips among group members. We propose a dynamic programming technique to eliminate the trip combinations that cannot be part of the query answer. We perform experiments using real and synthetic datasets and show that our approach outperforms a straightforward approach with a large margin.

1 Introduction

Family members normally have many outdoor tasks to perform within a short time for the proper management of their home. The members of a family may need to go to a bank to withdraw or deposit money, a pharmacy to buy medicine, or a supermarket to buy groceries. Similarly, organizers of an event may need to visit different points of interests (POIs) like supermarkets, banks, and restaurants to perform many tasks. In reality, all family or organizing members do not need to visit every POI and they can distribute the tasks among themselves. These scenarios motivate us to introduce a *group trip scheduling (GTS) query* that enables a group (e.g., a family) to schedule multiple trips among group members with the minimum total travel distance.

Users have some routine work like traveling from home to office or office to home, and they would prefer to visit other POIs on the way to office or returning home. Given source and destination locations of n group members, a GTS query returns n individual trips such that n trips together visit required types of POIs, each POI type is visited by a single member of the group, and the total trip distance of n group members is minimized. The trip distance of a group member is measured as the distance between her source to destination via the POIs that the group member visits. If the total travel distance is reduced, it will obviously cut down the cost for arranging an event or managing a set of tasks, which is very much desired. In this paper, we propose an efficient approach to process GTS queries for both Euclidean space and road networks.

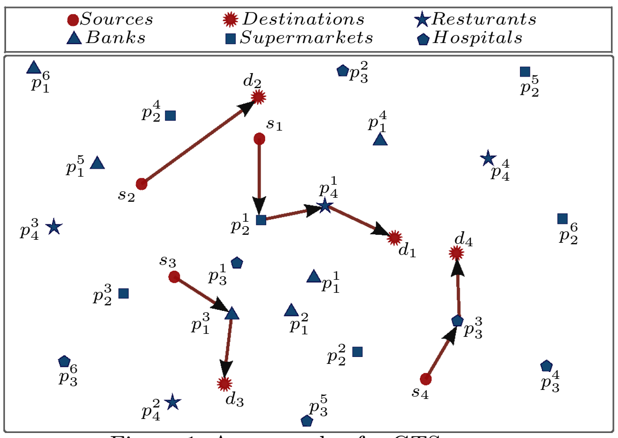


Figure 1: An example of a GTS query

In Figure 1, we consider a group or a family of four members. Every member has preplanned source and destination locations which may be home, office or any other place. Group members u_1, u_2, u_3, and u_4 have source destination pairs, $< s_1, d_1 >$, $< s_2, d_2 >$, $< s_3, d_3 >$, and $< s_4, d_4 >$, respectively. Here, p_j^k denotes a POI of type c_j with ID k. For example, POI p_1^2 in the figure is of type c_1, which represents a bank. The group has to visit four POI types: a bank (c_1), a supermarket (c_2), a hospital (c_3), and a restaurant (c_4). For each POI type, there are many options. For example, in real life, banks have many branches in different locations. A GTS query considers all options for each type of POIs, and returns four trips for four group members with the minimum total trip distance, where each POI type is included in a single trip. Figure 1 shows four scheduled trips: $s_1 \rightarrow p_2^1 \rightarrow p_4^1 \rightarrow d_1$, $s_2 \rightarrow d_2$, $s_3 \rightarrow p_1^3 \rightarrow d_3$ and $s_4 \rightarrow p_3^3 \rightarrow d_4$.

A major challenge of our problem is to find the set of POIs from a huge amount of candidate POI sets that provide the optimal answer in real time. For example, California City has about 87635 POIs with 63 different POI types [2]. For each POI type, there are on average 1300 POIs. If the required number of POI types is 4 then the number of candidate POI sets for a GTS query is $(1300) \times (1300) \times (1300) \times (1300) = (1300)^4 = 2.86e^{+12}$, a huge amount of candidate POI sets. We exploit elliptical properties to bound the POI search space, i.e., to prune POIs that cannot be part of the optimal answer. Though elliptical properties have been explored in the literature for processing other types of spatial queries [5, 7, 12, 13, 18] those pruning techniques are not directly applicable for GTS queries.

Furthermore, a GTS query needs to distribute the POIs of required types in a candidate set among group members. The candidate set contains exactly one POI from each of the m required POI types. The number of possible ways to distribute a candidate POI set of m POIs among n group members is n^m. Thus, the efficiency of a GTS query depends on the refinement of the POI search space and the technique to schedule trips among group members. We develop a dynamic programming technique to reduce the number of possible combinations while scheduling trips among group members. The technique eliminates the trip combinations that cannot be part of the optimal query answer.

Planning trips for a single user or a group in an effective and efficient manner has become an important topic in recent years. A trip planning (TP) query [13] for a single user finds the set of POIs of required types that minimize the trip distance with respect to the user's source and destination locations. To evaluate a GTS query, applying a trip planning algorithm for every user independently for all possible combinations of required POI types requires multiple traversal of the database and would be prohibitively expensive. A group trip planning (GTP) query [8] identifies the set of POIs of required types that minimize the total trip distance with respect to the source and destination locations of group members. In a GTP query, each required POI type is visited by all group members. On the other hand, in a GTS query, separate trips are planned for every group member and each required POI type is visited by only a single group member. For the example scenario mentioned in Figure 1, in Figure 2 we show the resultant trips for a GTP query, where the group members visit all required POI types together. A GTS query is also different from traveling salesman problem (TSP) [11] and its variants [4, 6, 15, 20]. The TSP and its variants assume a limited set of POIs and cannot handle a large dataset like a huge amount of POIs stored in a database.

To the best of our knowledge, we propose the first approach for GTS queries. In summary, the contributions of this paper are as follows:

- We introduce a new type of query, the group trip scheduling (GTS) query in spatial databases.

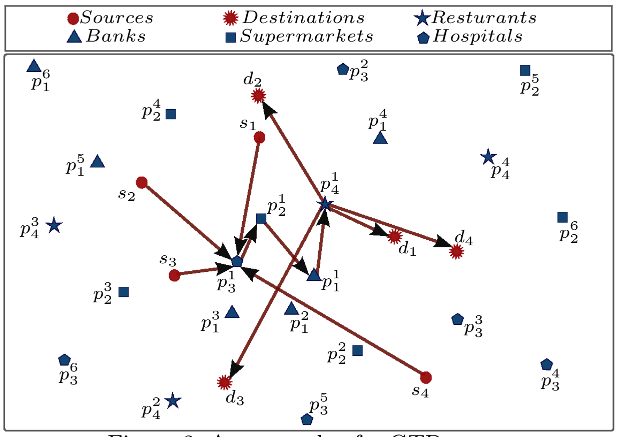


Figure 2: An example of a GTP query

- We present an efficient GTS query processing algorithm. Specifically, we refine the POI search space for processing GTS queries efficiently using elliptical properties and develop an efficient dynamic programming technique to schedule trips among group members.
- We perform extensive experimental evaluation of the proposed techniques and provide an comparative analysis of experimental results using both real and synthetic datasets.

2 Problem Definition

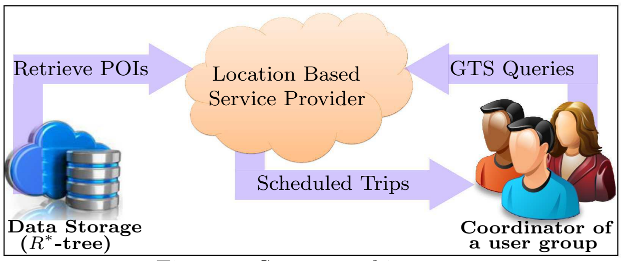
A GTS query for a group is formally defined as follows.
Definition 1.**[Group Trip Scheduling(GTS) Queries.]** Given a set $\mathbb{P}$ of POIs of different types in a 2-dimensional space, a set of n group members $U = \{u_1, u_2, \ldots, u_n\}$ with independent n source locations $S = \{s_1, s_2, \ldots, s_n\}$ and corresponding n destination locations $D = \{d_1, d_2, \ldots, d_n\}$, and a set of m POI types $\mathbb{C} = \{c_1, c_2, \ldots, c_m\}$, a GTS query returns a set of n trips, $T = \{T_1, T_2, \ldots, T_n\}$ that minimizes the total trip distance, $AggTripDist$ of group members, where a trip T_i corresponds to a group member u_i, group members together visit required types of POIs in $\mathbb{C}$, and a POI type in $\mathbb{C}$ is visited by a single member of the group.

For any two point locations x_1 and x_2 in a 2-dimensional space, let Function $Dist(x_1, x_2)$ return the distance between x_1 and x_2, where the distance can be measured either in the Euclidean space or road networks. The Euclidean distance is measured as the length of the direct line connecting x_1 and x_2. On the other hand, the road network distance is measured as the length of the shortest path between x_1 and x_2 on a given road network graph $\mathbb{G} = (\mathbb{V}, \mathbb{E}, \mathbb{W})$, where each vertex $v \in \mathbb{V}$ represents a road junction, each edge $(v, v') \in \mathbb{E}$ represents a direct path connecting vertices v and v' in $\mathbb{V}$, and each weight $w_{v,v'} \in \mathbb{W}$ represents the length of the direct path represented by the edge (v, v').

A trip T_i of group member u_i starts at s_i, ends at d_i, goes through POIs in A_i, where A_i includes at most m POIs of types specified in $\mathbb{C}$ and $m = |\mathbb{C}| = \sum_{i=1}^{n} |A_i|$. The total trip distance of group members is measured as $AggTripDist = \sum_{i=1}^{n} TripDist_i$. Let p_j denote a POI of type $c_j \in \mathbb{C}$. Without loss of generality, for $A_i = \{p_1, p_2, p_3\}$ and $\{c_1, c_2, c_3\} \in \mathbb{C}$, the trip distance $TripDist_i$ of T_i is computed as $Dist(s_i, p_1) + Dist(p_1, p_2) + Dist(p_2, p_3) + Dist(p_3, d_i)$, if the POI order $p_1 \rightarrow p_2 \rightarrow p_3$ gives the minimum value for $TripDist_i$.

3 System Architecture

Figure 3 shows an overview of the system architecture. The coordinator of the group sends a GTS query request to a location based service provider (LSP). The coordinator provides the source and destination locations of group members and the required POI types that the group members need to visit combinedly. The LSP incrementally retrieves POIs from the database, processes GTS queries and returns scheduled trips to the coordinator of the group that minimizes the total trip distance of the group members.



Retrieve POIs
Location Based Service Provider
GTS Queries
Scheduled Trips
Data Storage (R^*-tree)
Coordinator of a user group

Figure 3: System architecture

4 Related Work

Trip planning techniques exist for both single user and group in the literature. Trip planning (TP) queries have been introduced in [12] for a single user. TP queries allow a user to find an optimal route to visit POIs of different types while traveling from her source to destination location. In parallel to the work of TP queries, in [18], Sharifzadeh et al. addressed the optimal sequenced route (OSR) query that also focuses on planning a trip with the minimum travel distance for a single user for a fixed sequence of POI types (e.g., a user first visits a restaurant then a shopping center and a movie theater at the end). In [5], a generalization of the trip planning query, called the multi-rule partial sequenced route (MRPSR) query has been proposed that supports multiple constraints and a partial sequence ordering to visit POI types, and provides a uniform framework to evaluate both of the above mentioned variants [12, 18] of trip planning queries. In [16], the authors proposed an incremental algorithm to find the optimal sequenced route in the Euclidean space and then determine the optimal sequence route in road networks based on the incremental Euclidean restriction. A GTS query is different from TP and OSR queries as GTS queries schedule trips among group members.

A group trip planning query that plans a trip with the minimum aggregate trip distance to visit POIs of different types with respect to source and destination locations of group members has been first proposed in [8]. In [3, 17], the authors proposed efficient algorithms to process GTP queries for a fixed sequence of visiting POI types. In [7], the authors developed an efficient algorithm to process GTP queries in both Euclidean space and road networks. In a GTP query, all group members visit all POI types in their trips, whereas in a GTS query, each POI type is visited by a single member in the group.

A traveling salesman problem (TSP) and variants that focus on planning routes with a limited set of locations are well studied problems in the literature. A generalized traveling salesman problem (GTSP) [6] and multiple traveling salesman problem (MTSP) [4] are well known variations of TSP. A GTSP assumes that from groups of given locations, a salesman visits a location from every group such that the travel distance for the route becomes the minimum. The MTSP allows more than one salesman to be involved in the solution. In MTSP, if the salesmen are initially based at different depots then this variation is known as the multiple depot multiple traveling salesman problem (MDMTSP). However, the limitation of the proposed solutions for TSP and its variants is that they cannot handle a large dataset (e.g., POI data) stored in the database, a scenario that is addressed by a GTS query.

Elliptical properties have been used in the literature to refine the search region for queries like group nearest neighbor queries [14], trip planning queries [12], group trip planning queries [7] and privacy preserving trip planning queries [19]. Though all of these refinement techniques present the refined search region with an ellipse, they differ on the way to set the foci and the length of the major axis of the ellipse. In this paper, we develop two novel techniques to refine the search region using ellipses for GTS queries.

5 Our Approach

In this section, we present our approach to process GTS queries in the Euclidean space and road networks. In a GTS query, the coordinator of a group sends the query request to the LSP and provides required information like group members' source and destination locations, and the required POI types. POI information is indexed using an R^*-tree [1] in the database. The LSP incrementally retrieves POIs from the database until it identifies the trips that minimize the total travel distance of the group members. The underlying idea of the efficiency of our approach is the POI search region refinement techniques using elliptical properties and the dynamic programming technique to schedule multiple trips among the group members.

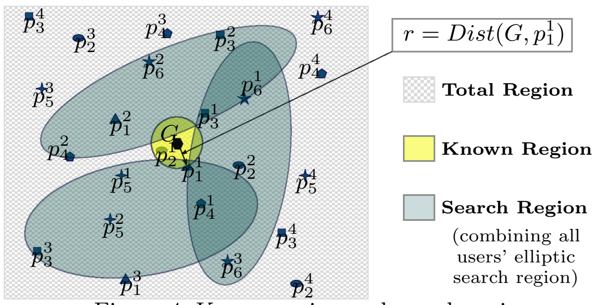


Figure 4: Known region and search region

We use the concept of known region and search region [7, 12] for the retrieval of POIs from the database. The known region represents the area which has already been explored, that means all POIs inside the known region have been retrieved from the database. The search region represents the refined space that we need to explore for the optimal solution. In Figure 4, suppose the LSP retrieves the nearest POIs p_2^1 and p_1^1 with respect to the geometric centroid G of source and destination locations of a group of three members, where p_1^1 is the farthest POI from G among POIs p_2^1 and p_1^1 that have been already retrieved. The circular region centered at G with radius equal to the distance between G and p_1^1 is the known region. We refine the POI search region with respect to the retrieved POIs in the known region using multiple ellipses, and call it simply a search region. In

Figure 4, based on current retrieved POIs, p_2^1 and p_1^1, the search region is the union of three ellipses.

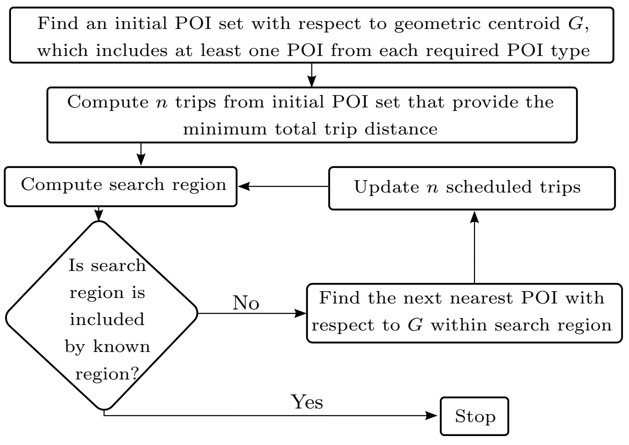

Find an initial POI set with respect to geometric centroid G, which includes at least one POI from each required POI type
Compute n trips from initial POI set that provide the minimum total trip distance
Compute search region
Update n scheduled trips
Is search region is included by known region?
No
Find the next nearest POI with respect to G within search region
Yes
Stop

Figure 5: Overview of our approach for GTS queries

Figure 5 shows an overview of our developed approach for processing GTS queries. Our approach initially incrementally retrieves the nearest POIs from G until at least one POI from each required POI type has been retrieved. Using the initial retrieved POI set, our approach schedules n trips that provide the minimum total travel distance for the group members, and refines the search region to prune POIs that cannot be the part of the query answer. Then the proposed approach checks whether the known region includes the search region. If yes, then our approach has retrieved all POIs that are required to find the optimal answer and the approach terminates the search. Otherwise, our approach continues to incrementally retrieve the next nearest POIs within the search region, updates scheduled n trips, refines the search region, and checks the termination condition of the search until the condition becomes true. In the following sections, we elaborate the steps of our approach for processing GTS queries.

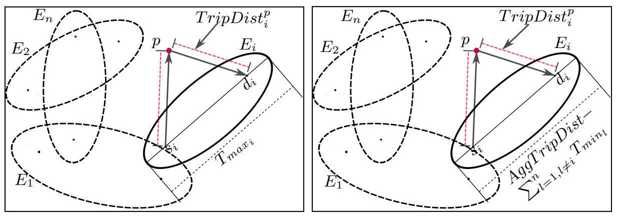
5.1 Computing the known region

For both Euclidean and road network spaces, our approach incrementally retrieves the Euclidean nearest POIs with respect to the geometric centroid G of n source-destination pairs of group members. It uses the best-first search (BFS) to find the POIs of required POI types that are assumed to be indexed using an R^*-tree [1] in the database. The BFS search also prunes the POIs whose types do not match with the required POI types and returns the remaining POIs.

Let the BFS discover p_j as the first nearest POI with respect to G. The circular region centered at G with radius r equal to the Euclidean distance between G and p_j is the known region. With the retrieval of the next nearest POI, r is updated with the Euclidean distance from G to the last retrieved nearest POI from the database.

5.2 Refinement of the search region

The key idea of our search region refinement techniques is based on elliptical properties. A smaller search region decreases the number of POIs retrieved from the database, avoids unnecessary trip computations, and reduces I/O access and computational overhead significantly. We present two novel techniques in Theorems 1 and 2 to refine the search region using multiple ellipses, and based on these two refinement techniques, we develop our algorithm to process GTS queries in Section 5.5. The notations that we use in our theorems are summarized below:



(a) Proof of Theorem 1 (b) Proof of Theorem 2

Figure 6: Search region refinement

- T_{min_i}: the minimum trip distance for a group member u_i, i.e., the distance between s_i and d_i without visiting any POI type.
- T_{max_i}: the maximum trip distance for a group member u_i, i.e., the trip distance from s_i to d_i via required m POI types.
- $TripDist_i$: the current trip distance of a group member u_i among the scheduled trips.
- $AggTripDist$: the current minimum total trip distance of the group.

Above notations are measured in terms of Euclidean distances if a GTS query is evaluated in the Euclidean space, and in terms of road network distances if a GTS query is evaluated in the road networks. Theorems 1 and 2 show two ways to refine the search region for a GTS query in the Euclidean space and road networks.

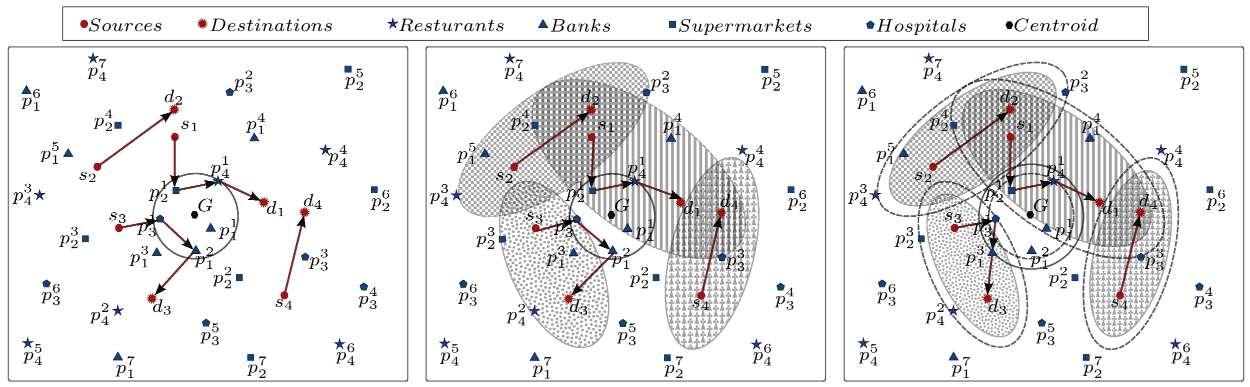
THEOREM 1. *The search region can be refined as the union of n ellipses $E_1 \cup E_2 \cup \ldots \cup E_n$, where the foci of ellipse E_i are at s_i and d_i, and the major axis of the ellipse E_i is equal to T_{max_i}.*

PROOF. Let a POI p lie outside the search region, $E_1 \cup E_2 \cup \ldots \cup E_n$, and $AggTripDist^p$ be the total trip distance of the group, where a group member u_i's trip includes POI p as shown in Figure 6(a). We have to prove that POI p can not be a part of the optimal solution, i.e., $AggTripDist^p > AggTripDist$. Let $TripDist_i^p$ be the trip distance for the group member u_i whose trip includes POI p. An elliptical property states that the Euclidean distance between two foci via a point outside the ellipse is greater than the length of the major axis. Since the road network distance is greater than or equal to the Euclidean distance, the road network distance between two foci via a point outside the ellipse is also greater than the length of the major axis. As POI p lies outside the ellipse E_i, for both Euclidean and road network spaces we have,

$$TripDist_i^p > T_{max_i} \qquad (1)$$

T_{max_i} represents the trip distance of user u_i for visiting m POI types. Any trip passing through the POI p outside the ellipse E_i can not give better trip distance for user u_i. Thus, any POI outside the union of ellipses $E_1, E_2, \ldots, E_n$ can not improve the total trip distance $AggTripDist$ for the group and can not be a part of an optimally scheduled group of trips. Thus, $AggTripDist^p > AggTripDist$. □

THEOREM 2. *The search region can be refined as the union of n ellipses $E_1 \cup E_2 \cup \ldots \cup E_n$, where the foci of ellipse E_i are at s_i and d_i, and the major axis of the ellipse is equal to $AggTripDist - \sum_{l=1,l\neq i}^{n} T_{min_l}$.*



(a) Initial known region (the circle with center G) and scheduled trips calculated using initial POIs

(b) Refined search region

(c) Known region expands (outer circle) and search region shrinks (inner ellipses)

Figure 7: Steps of our approach

PROOF. Let a POI p lie outside the search region, $E_1 \cup E_2 \cup \ldots \cup E_n$, and $AggTripDist^p$ be the total trip distance of the group, where a group member u_i's trip includes POI p as shown in Figure 6(b). We have to prove that POI p can not be a part of the optimal solution, i.e., $AggTripDist^p > AggTripDist$.

Let $TripDist_i^p$ be the trip distance for the group member u_i whose trip includes POI p. An elliptical property states that the Euclidean distance between two foci via a point outside the ellipse is greater than the length of the major axis. Since the road network distance is greater than or equal to the Euclidean distance, the road network distance between two foci via a point outside the ellipse is also greater than the length of the major axis. As the POI p lies outside the ellipse E_i, for both Euclidean and road network spaces we have,

$$TripDist_i^p > AggTripDist - \sum_{l=1, l\neq i}^{n} T_{min_l}$$

Rearranging the equation we get,

$$TripDist_i^p + \sum_{l=1, l\neq i}^{n} T_{min_l} > AggTripDist \tag{2}$$

By definition we know,

$$AggTripDist^p = TripDist_i^p + \sum_{l=1, l\neq i}^{n} TripDist_l^p \tag{3}$$

and

$$\sum_{l=1, l\neq i}^{n} TripDist_l^p \geq \sum_{l=1, l\neq i}^{n} T_{min_l} \tag{4}$$

From Equations 3 and 4, we get,

$$AggTripDist^p \geq TripDist_i^p + \sum_{l=1, l\neq i}^{n} T_{min_l} \tag{5}$$

Combining inequalities of 2 and 5,

$$AggTripDist^p > AggTripDist$$

Thus, any POI outside the search region $E_1 \cup E_2 \cup \ldots \cup E_n$ can not improve the total trip distance for the group and can not be a part of an optimally scheduled group of trips. □

Our approach refines the ellipses of every group member independently using both bounds proposed in Theorems 1 and 2, and selects the bound that provides the minimum length for the major axis of the ellipse. For the same foci, the smaller major axis represents a smaller ellipse. It may happen that for an ellipse of a member, Theorem 1 provides the minimum length of the major axis and for another member's ellipse, Theorem 2 provides the minimum length of the major axis. The refined search region is computed as the union of the smaller ellipses of all group members.

For a GTS query, our approach retrieves an initial set of nearest POIs that includes at least one POI of each required type. From the initial set of POIs, our approach schedules trips with the minimum total trip distance for the group using the dynamic programming technique shown in Section 5.4, and refines the search region using Theorems 1 and 2. With the incremental retrieval of the nearest POIs from G within the refined search region, our approach checks and updates the scheduled trips, if the newly discovered POIs improve the current scheduled trips. The newly updated trips may improve the bound T_{max_i} for a group member or the total trip distance of the group $AggTripDist$, which can further refine the search region.

Figure 7(a) shows the initial set of retrieved POIs $p_1^1, p_1^2, p_2^1, p_3^1, p_4^1$, the known region, and four scheduled trips using the initial POI set for a group of four members. Note that the initial set may include more than one POIs of same POI type (e.g., p_1^1 and p_1^2) because the incremental nearest POI retrieval continues until the initial set includes at least one POI from every required POI type. Using bounds from Theorem 1 and 2, we compute and refine the search region. Figure 7(b) shows the refined search region as the union of four ellipses. After retrieving the next nearest POI p_1^3, the known region expands, which has the radius equal to $Dist(G, p_1^3)$. Our approach checks whether this new POI can improve the current solution. In this example, the new POI p_1^3 decreases the trip distance for group member u_3 and thus, the updated trip for u_3 is $s_3 \to p_3^1 \to p_1^3 \to d_3$. It also improves the total trip distance and shrinks the search region for all group members. In Figure 7(c), the dotted lines show the scenario before retrieving POI p_1^3 and the shaded areas with solid lines show the updated scenario after retrieving the POI p_1^3. With the retrieval of the nearest POIs from the database, the known region expands and the search region shrinks or remains same.

5.3 Terminating condition for POI retrieval

When the known region covers the search region, no more minimization in the total trip distance is further possible. At this point, we can terminate traversing R^*-tree and retrieving POIs. Figure 8 shows that the known region covers the search region.

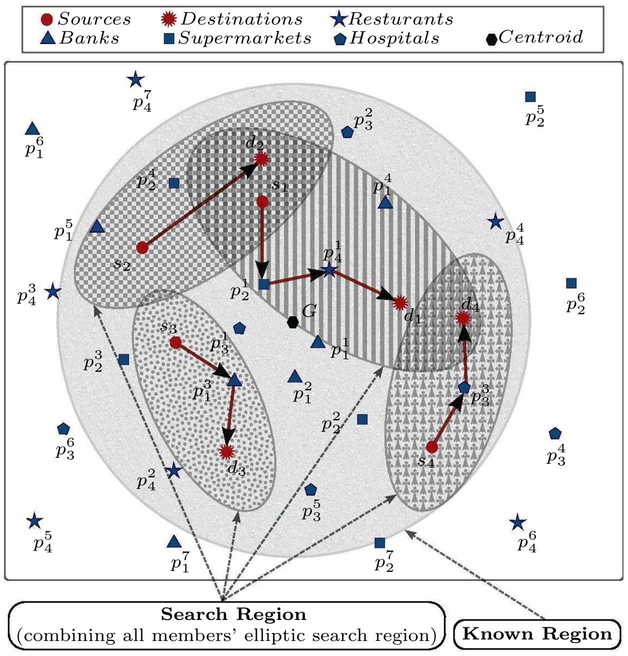


Figure 8: Terminating condition: the known region includes the search region

5.4 Dynamic programming technique for scheduling trips

Scheduling the trips among the group members is an essential component of GTS query processing approach. After retrieving the initial POI set, our approach schedules the trips among the group members such that the total trip distance of the group is minimized. Each time our approach retrieves new POIs, it again schedules trips using new POIs, if the new trips improve the total trip distance of the group. Thus, the efficiency of our approach largely depends on the computational cost of scheduling trips among the group members. We propose a dynamic programming technique to schedule the trips among the group members. The technique reduces the number of trip combinations that we need to consider to find the set of trips with the minimum total trip distance. The distances computed in our dynamic programming technique are Euclidean distances, if a GTS query is processed in the Euclidean space, and the distances are road network distances, otherwise.

Our dynamic programming technique minimizes the following objective function:

$$\sum_{i=1}^{n} TripDist_i$$

satisfying constraints that a group of n members together visit m different POI types and each POI type is visited by a single group member. Let $\mathbb{C}_{T_i}$ be the set of POI types visited by trip T_i of user u_i, where $0 \leq |\mathbb{C}_{T_i}| \leq m$. Formal representation of the constraints are as follows. The dynamic programming technique satisfies,

$$\sum_{i=1}^{n} |\mathbb{C}_{T_i}| = m, \quad \bigcup_{i=1}^{n} \mathbb{C}_{T_i} = \mathbb{C} \text{ and } \forall_{i,j}(\mathbb{C}_{T_i} \cap \mathbb{C}_{T_j}) = \emptyset$$

For the GTS query, we have a set of m POI types

Table 1: Structure of dynamic table ν_y, where $0 \leq y \leq (m-1)$

	$\{u_1\}$	$\dots$	$\{u_n\}$	$\{u_1u_2\}$	$\dots$	$\{u_1u_2\dots u_{n-1}\}$
$\{c_1, c_2, \dots, c_y\}$						
$\{c_1, c_3, \dots, c_y\}$						
$\vdots$						

Table 2: Structure of dynamic table ν_m

	$\{u_1\}$	$\dots$	$\{u_n\}$	$\{u_1u_2\}$	$\dots$	$\{u_1u_2\dots u_n\}$
$\{c_1, c_2, \dots, c_m\}$						

$\mathbb{C}=\{c_1, c_2, \dots, c_m\}$, where a group member visits any number of POI types from 0 to m. Thus, there are $\sum_{y=0}^{m}(^mC_y)$ ways to choose any y POI types from $m(= |\mathbb{C}|)$ different POI types, where $0 \leq y \leq m$. Suppose $^{\mathbb{C}}C_y$ denotes the set of all possible y chooses from the set of POI types $\mathbb{C}$. Let $(^{\mathbb{C}}C_y)^j$ represent the jth member of the set $^{\mathbb{C}}C_y$. Suppose we have a set of $m = |\mathbb{C}| = 4$ POI types, $\mathbb{C} = \{c_1, c_2, c_3, c_4\}$. For $y = 2$, the number of ways to choose y POI types from $m(= |\mathbb{C}|)$ POI types is $^{|\mathbb{C}|}C_y = {}^4C_2 = 6$ and the set all possible y chooses from the set $\mathbb{C}$ is $^{\mathbb{C}}C_y = \{\{c_1, c_2\}, \{c_1, c_3\}, \{c_1, c_4\}, \{c_2, c_3\}, \{c_2, c_4\}, \{c_3, c_4\}\}$, where $(^{\mathbb{C}}C_y)^1 = \{c_1, c_2\}$, $(^{\mathbb{C}}C_y)^2 = \{c_1, c_3\}, \dots, (^{\mathbb{C}}C_y)^6 = \{c_3, c_4\}$.

For each member of the set $^{\mathbb{C}}C_y$, we calculate optimal trips for each group member in $U = \{u_1, u_2, u_3, \dots, u_n\}$ and store trip distances for future computations. This is the initial step for our dynamic programming technique. We define $m+1$ dynamic tables, $\nu_0, \nu_1, \nu_2, \dots \nu_m$ to store the trip distances of every group member and the combined trip distances of the group members. Table ν_y has ^mC_y rows, where jth row corresponds to jth member of the set $^{\mathbb{C}}C_y$, i.e., $(^{\mathbb{C}}C_y)^j$.

Each table has two types of columns : ***single member columns*** and ***combined member columns***. Each table has n single member columns, where each column corresponds to a member of the group $U = \{u_1, u_2, u_3, \dots, u_n\}$. The cells of these columns store the minimum trip distances for the corresponding column's member to visit the POI types of the corresponding rows. Each dynamic table except ν_m has $(n-2)$ combined member columns $u_1u_2, u_1u_2u_3, \dots, u_1u_2..u_{n-1}$, where the cells of the corresponding columns store the combined trip distances of the corresponding column's multiple members. For example, each cell of the column u_1u_2 stores the minimum combined trip distance of user u_1 and u_2 to visit the POI types of the corresponding row, where a POI type is visited either by u_1 or u_2. Table 1 shows the structure of ν_y where $0 \leq y \leq (m-1)$. Table 2 shows the structure of ν_m that has an extra column $u_1u_2 \dots u_n$ to store the minimum total trip distance for n scheduled trips, where n trips together visit m required POI types and every POI type is visited by a single trip. The table has only one row which contains all m POI types.

In addition to storing the minimum trip distance, each cell of the dynamic tables stores the set of POIs for which

Table 3: Possible number of POI type distributions between u_1 and u_2

u_1	u_2
3	0
2	1
1	2
0	3

Table 4: Candidate trips with trip distances for cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$

Candidate trips	**Distances**
$s_1 \rightarrow p_2^1 \rightarrow p_1^1 \rightarrow d_1$	65.55
$s_1 \rightarrow p_2^1 \rightarrow p_1^2 \rightarrow d_1$	61.72
$s_1 \rightarrow p_1^1 \rightarrow p_2^1 \rightarrow d_1$	60.44
$s_1 \rightarrow p_1^2 \rightarrow p_2^1 \rightarrow d_1$	51.58

Table 5: Dynamic tables for the example scenario

(a) Dynamic table ν_0

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\emptyset$	51.55	93.33	68.84	81.78	144.88	213.72

(b) Dynamic table ν_1

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1\}$	51.57	96.22	123.61	90.67	144.90	213.74
$\{c_2\}$	51.56	93.33	68.84	81.78	144.88	213.72
$\{c_3\}$	51.55	93.97	78.31	81.79	144.88	213.72
$\{c_4\}$	51.55	93.33	68.84	81.78	144.88	213.72

(c) Dynamic table ν_2

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1, c_2\}$	51.58	96.22	123.61	90.67	144.90	213.74
$\{c_1, c_3\}$	51.86	96.26	123.68	90.70	145.19	214.03
$\{c_1, c_4\}$	51.57	96.23	123.61	90.67	144.90	213.74
$\{c_2, c_3\}$	51.57	93.97	78.34	81.81	144.88	213.72
$\{c_2, c_4\}$	51.56	93.34	68.84	81.78	144.88	213.72
$\{c_3, c_4\}$	51.55	93.97	78.32	81.79	144.88	213.72

(d) Dynamic table ν_3

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1, c_2, c_3\}$	51.90	96.26	123.68	90.70	145.19	214.03
$\{c_1, c_2, c_4\}$	51.59	96.23	123.61	90.67	144.90	213.74
$\{c_1, c_3, c_4\}$	51.88	96.28	123.68	90.71	145.19	214.03
$\{c_2, c_3, c_4\}$	51.57	93.97	78.34	81.81	144.88	213.72

(e) Dynamic table ν_4

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$	$\{u_1u_2u_3u_4\}$
$\{c_1, c_2, c_3, c_4\}$	51.90	96.28	123.69	90.71	145.20	214.03	295.53

the minimum trip distance is obtained. For example, cell $\nu_3[\{c_1, c_3, c_4\}][\{u_1\}]$ stores the minimum trip distance and the POI set $< p_3, p_1, p_4 >$, for which u_1 obtains the minimum trip distance.

The size of a dynamic table ν_y is : ${}^mC_y \times (n + (n-2))$, where $0 \leq y \leq (m-1)$, and the size of table ν_m is ${}^mC_m \times (n + (n-2)+1)$. Thus, the total space required for dynamic tables is $\sum_{y=0}^{(m-1)}({}^mC_y \times (n+(n-2))) + ({}^mC_m \times (n+(n-2)+1)) = (2^{m+1} \times (n-1)+1)$ units. Similarly, the processing time of the dynamic programming technique is proportional to the number of the dynamic tables and the number of cells in a dynamic table, which vary with the values of m and n.

Contents of cells of the single member columns of a dynamic table are computed using already retrieved POIs from the database. To compute the contents of cells of the combined member columns of a dynamic table ν_y, we use the single member columns of the same table, and both single and combined member columns of $\nu_0, \nu_1, \ldots, \nu_{y-1}$. For example, for computing each cell of combined member column u_1u_2 of ν_4, we use the already calculated single member columns of ν_4, and both single and combined member columns of ν_0, ν_1, ν_2 and ν_3 based on possible number of POI type distributions between members u_1 and u_2 of that corresponding column. For the example scenario, to visit 3 POI types, possible ways to distribute the number of POI types between u_1 and u_2 are listed in Table 3. Formally, the minimum trip distance stored in a cell (e.g., $\nu_y[\{c_1, c_2, \ldots, c_y\}][\{u_1u_2\}]$ of table ν_y) is computed as $\min_{g=0}^{y}\{\min_{j=1}^{{}^mC_g}\{\min_{k=1}^{{}^mC_{y-g}}(\nu_g[({}^{\mathbb{C}}C_g)^j][\{u_1\}] + \nu_{(y-g)}[({}^{\mathbb{C}}C_{(y-g)})^k][\{u_2\}])\}\}$, where $({}^{\mathbb{C}}C_g)^j \cap ({}^{\mathbb{C}}C_{(y-g)})^k = \emptyset$. The constraint guarantees that no POI type is considered twice while computing the minimum trip distance.

Similar to the combined member column u_1u_2, for computing each cell of combined member column $u_1u_2u_3$ of ν_4, we use the same dynamic tables, and similar distribution listed in Table 3 between combined members u_1u_2 (instead of u_1) and single member u_3 (instead of u_2). Thus, we incrementally compute dynamic tables $\nu_0, \nu_1, \nu_2, \ldots, \nu_m$, one by one and finally we get our desired result for a GTS query.

We elaborate our dynamic programming technique with an example. Suppose a group of 4 members, $\{u_1, u_2, u_3, u_4\}$, together want to visit 4 POI types $\{c_1, c_2, c_3, c_4\}$ with the minimum total trip distance, and each POI type is visited by a single member. Here, $n = 4$, $m = 4$, and a group member can visit any number of POI types between 0 to m.

Figure 7(a) shows the initial set of retrieved POIs: $p_1^1, p_1^2, p_2^1, p_3^1, p_4^1$ and the known region. The initial set includes at least a POI from every POI type. Using these POIs, we first compute all possible trips for the group members and then schedule the trips using our proposed dynamic programming technique.

We define $(m+1)$, i.e., 5 tables, ν_0, ν_1, ν_2, ν_3 and ν_4 to store the computed trip distances and combined trip distances of the group members. Each dynamic table ν_y has ${}^{m=4}C_y$ rows, where each row corresponds to a member of the set ${}^{\mathbb{C}}C_y$. Each table has $n = 4$ single member columns, where a column corresponds to a group member in $\{u_1, u_2, u_3, u_4\}$, and $n-2 = 2$ combined member columns, u_1u_2 and $u_1u_2u_3$. Table ν_4 contains an extra column $u_1u_2u_3u_4$ to store the minimum total trip distance of the 4 scheduled trips for 4 users that together visit 4 POI types, where each POI type is visited by a single user. Tables 5 (a-e) show ν_0, ν_1, ν_2, ν_3 and ν_4 for the considered example.

Computing single member columns: In the dynamic tables, columns u_1, u_2, u_3 and u_4 are the single member columns. Each cell of these columns of a table stores the minimum trip distance for the corresponding column's user passing through POI types of the corresponding row of that table. For example, in Table 5(c), cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$ contains the minimum trip distance for user u_1 passing through POI types c_1 and c_2. For computing this trip distance, we consider user, u_1's source (s_1) and destination (d_1) locations along with candidate POIs in the initial set: $\{p_1^1, p_1^2\}$ and $\{p_2^1\}$ with POI types c_1 and c_2, respectively. All candidate trips for cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$ using these POIs with the corresponding trip distances are listed in Table 4.

Among the candidate trips listed in this table, the minimum trip distance 51.58 for trip $s_1 \rightarrow p_1^2 \rightarrow p_2^1 \rightarrow d_1$ is stored in cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$. Similarly, our dynamic programming technique populates all cells of the single member columns of ν_1, ν_2, ν_3 and ν_4. Table ν_0 is a trivial one that stores trip distances for particular user's trip from her source to destination location only.

Computing combined member columns: Using the single member columns and already calculated combined member columns, we dynamically calculate the combined member columns of ν_0, ν_1, ν_2, ν_3 and ν_4 one by one.

In ν_0, cell $\nu_0[\emptyset][\{u_1u_2\}]$ contains the minimum total trip distance of trips T_1 and T_2, where the trips correspond to users u_1 and u_2, respectively, and visit no POI type. Table 6 shows the candidate combinations that are used to compute the cell value, where trip distances are for users' trips from their source to destination locations.

Table 6: Candidate combined combinations with trip distances for cell $\nu_0[\emptyset][\{u_1u_2\}]$

Combined Combinations	Distances	Total
$\nu_0[\emptyset][\{u_1\}]$ + $\nu_0[\emptyset][\{u_2\}]$	51.55 + 93.33	144.88

Table 7: Candidate combined combinations with trip distances for cell $\nu_1[\{c_1\}][\{u_1u_2\}]$

Combined Combinations	Distances	Total
$\nu_1[\{c_1\}][\{u_1\}]$ + $\nu_0[\emptyset][\{u_2\}]$	51.57 + 93.33	144.90
$\nu_0[\emptyset][\{u_1\}]$ + $\nu_1[\{c_1\}][\{u_2\}]$	51.55 + 96.22	147.77

To compute the cells of the combined member columns for other table ν_y, we need to consider all dynamic tables from ν_0 to ν_y. For example, in ν_2, cell $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$ stores the minimum total trip distance of trips T_1 and T_2, where the trips correspond to users u_1 and u_2, respectively. Here a user (u_1 or u_2) can visit any number (0 or 1 or 2) of POI types, but u_1 and u_2 together visit the POI types $\{c_1, c_2\}$, and each POI type is either visited by u_1 or u_2. For computing the cell value, we use stored single member trip distances and multiple member trip distances in ν_0, ν_1 and ν_2. Using ν_0, ν_1 and ν_2 (Tables 5(a-c)), Table 8 shows the candidate combinations of POI types for u_1 and u_2 along with the trip distances for computing the value for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$ in ν_2 (Table 5(c)). Among candidate combinations listed in Table 8, the minimum total trip distance 144.90 is stored in cell $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$.

Table 8: Candidate combined combinations with trip distances for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$

Combined Combinations	Distances	Total
$\nu_2[\{c_1, c_2\}][\{u_1\}]$ + $\nu_0[\emptyset][\{u_2\}]$	51.58 + 93.33	144.91
$\nu_1[\{c_1\}][\{u_1\}]$ + $\nu_1[\{c_2\}][\{u_2\}]$	51.57 + 93.33	144.90
$\nu_1[\{c_2\}][\{u_1\}]$ + $\nu_1[\{c_1\}][\{u_2\}]$	51.56 + 96.22	147.78
$\nu_0[\emptyset][\{u_1\}]$ + $\nu_2[\{c_1, c_2\}][\{u_2\}]$	51.55 + 96.22	147.77

Similarly, our dynamic programming technique populates all cells of the combined member columns of ν_0, ν_1, ν_2, ν_3 and ν_4. Candidate combinations with trip distances for cell $\nu_1[\{c_1\}][\{u_1u_2\}]$, $\nu_3[\{c_1, c_2, c_3\}][\{u_1u_2\}]$ and $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1u_2\}]$ are listed in Table 7, Table 9 and Table 10, respectively.

Table 9: Candidate combined combinations with trip distances for cell $\nu_3[\{c_1, c_2, c_3\}][\{u_1u_2\}]$

Combined Combinations	Distances	Total
$\nu_3[\{c_1, c_2, c_3\}][\{u_1\}]$ + $\nu_0[\emptyset][\{u_2\}]$	51.90 + 93.33	145.23
$\nu_2[\{c_1, c_2\}][\{u_1\}]$ + $\nu_1[\{c_3\}][\{u_2\}]$	51.58 + 93.97	145.55
$\nu_2[\{c_1, c_3\}][\{u_1\}]$ + $\nu_1[\{c_2\}][\{u_2\}]$	51.86 + 93.33	145.19
$\nu_2[\{c_2, c_3\}][\{u_1\}]$ + $\nu_1[\{c_1\}][\{u_2\}]$	51.57 + 96.22	147.79
$\nu_1[\{c_1\}][\{u_1\}]$ + $\nu_2[\{c_2, c_3\}][\{u_2\}]$	51.55 + 96.22	147.77
$\nu_1[\{c_2\}][\{u_1\}]$ + $\nu_2[\{c_1, c_3\}][\{u_2\}]$	51.56 + 96.26	147.82
$\nu_1[\{c_3\}][\{u_1\}]$ + $\nu_2[\{c_1, c_2\}][\{u_2\}]$	51.57 + 93.97	145.54
$\nu_0[\emptyset][\{u_1\}]$ + $\nu_3[\{c_1, c_2, c_3\}][\{u_2\}]$	51.55 + 96.26	147.81

Table 10: Candidate combined combinations with trip distances for cell $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1u_2\}]$

Combined Combinations	Distances	Total
$\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1\}]$+$\nu_0[\emptyset][\{u_2\}]$	51.90+93.33	145.23
$\nu_3[\{c_1, c_2, c_3\}][\{u_1\}]$+$\nu_1[\{c_4\}][\{u_2\}]$	51.90+93.33	145.23
$\nu_3[\{c_1, c_2, c_4\}][\{u_1\}]$+$\nu_1[\{c_3\}][\{u_2\}]$	51.59+93.97	145.56
$\nu_3[\{c_1, c_3, c_4\}][\{u_1\}]$+$\nu_1[\{c_2\}][\{u_2\}]$	51.88+93.33	145.21
$\nu_3[\{c_2, c_3, c_4\}][\{u_1\}]$+$\nu_1[\{c_1\}][\{u_2\}]$	51.57+96.22	147.79
$\nu_2[\{c_1, c_2\}][\{u_1\}]$+$\nu_2[\{c_3, c_4\}][\{u_2\}]$	51.58+93.97	145.55
$\nu_2[\{c_1, c_3\}][\{u_1\}]$+$\nu_2[\{c_2, c_4\}][\{u_2\}]$	51.86+93.34	145.20
$\nu_2[\{c_1, c_4\}][\{u_1\}]$+$\nu_2[\{c_2, c_3\}][\{u_2\}]$	51.57+93.97	145.54
$\nu_2[\{c_2, c_3\}][\{u_1\}]$+$\nu_2[\{c_1, c_3\}][\{u_2\}]$	51.57+96.23	147.80
$\nu_2[\{c_2, c_4\}][\{u_1\}]$+$\nu_2[\{c_1, c_3\}][\{u_2\}]$	51.56+96.26	147.82
$\nu_2[\{c_3, c_4\}][\{u_1\}]$+$\nu_2[\{c_1, c_2\}][\{u_2\}]$	51.55+96.22	147.77
$\nu_1[\{c_1\}][\{u_1\}]$+$\nu_3[\{c_2, c_3, c_4\}][\{u_2\}]$	51.55+96.26	147.81
$\nu_1[\{c_2\}][\{u_1\}]$+$\nu_3[\{c_1, c_3, c_4\}][\{u_2\}]$	51.55+96.23	147.78
$\nu_1[\{c_3\}][\{u_1\}]$+$\nu_3[\{c_1, c_2, c_4\}][\{u_2\}]$	51.56+96.28	147.84
$\nu_1[\{c_4\}][\{u_1\}]$+$\nu_3[\{c_1, c_2, c_3\}][\{u_2\}]$	51.57+93.97	145.54
$\nu_0[\emptyset][\{u_1\}]$+$\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_2\}]$	51.55+96.28	147.83

We gradually combine trips of other users, u_3 and u_4, and update the other combined member columns one by one. For example, in ν_2, cell $\nu_2[\{c_1, c_2\}][\{u_1u_2u_3\}]$ contains the minimum total trip distance of trips T_1, T_2 and T_3, where the trips correspond to users u_1, u_2 and u_3, respectively, and together visit the POI types $\{c_1, c_2\}$. Using ν_0, ν_1 and ν_2 (Tables 5(a-c)), Table 11 shows the candidate combinations of POI types for combined members u_1u_2 and single member u_3 along with the trip distances for computing the value for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2u_3\}]$ in ν_2 (Table 5(c)).

Table 11: Candidate combined combinations with trip distances for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2u_3\}]$

Combined Combinations	Distances	Total
$\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$+$\nu_0[\emptyset][\{u_3\}]$	144.90 + 68.84	213.74
$\nu_1[\{c_1\}][\{u_1u_2\}]$+$\nu_1[\{c_2\}][\{u_3\}]$	144.90 + 68.84	213.74
$\nu_1[\{c_2\}][\{u_1u_2\}]$+$\nu_1[\{c_1\}][\{u_3\}]$	144.88 + 123.61	268.49
$\nu_0[\emptyset][\{u_1u_2\}]$+$\nu_2[\{c_1, c_2\}][\{u_3\}]$	144.88 + 123.61	268.49

Similarly we compute all combined member columns of ν_0 to ν_4. The rightmost cell of the final table ν_m, which is $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1u_2u_3u_4\}]$ in our example scenario, contains the minimum total trip distance of four trips T_1, T_2, T_3 and T_4, where the trips correspond to users u_1, u_2, u_3 and u_4, respectively. These trips together visit all required POI types $\{c_1, c_2, c_3, c_4\}$ and each POI type is visited by a single user. This is actually the minimum total trip distance of the group for the dynamic scheduling based on the retrieved initial POIs: $p_1^1, p_1^2, p_2^1, p_3^1, p_4^1$. The minimum total trip distance 295.53 is stored in cell $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1u_2u_3u_4\}]$.

Note that the rightmost cell of the final table $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1u_2u_3u_4\}]$ contains the minimum total trip distance of the group which is *AggTripDist* that we have mentioned in Section 5.2. To get the values of T_{min_i} and T_{max_i} for each user u_i, we simply take the minimum and maximum values from Table 5(a) and Table 5(e), respectively. T_{min_i} and T_{max_i} values for users $\{u_1, u_2, u_3, u_4\}$ are

$\{51.55, 93.33, 68.84, 81.78\}$ and $\{51.90, 96.28, 123.69, 90.71\}$, respectively. Using these values we refine the search region based on Theorems 1 and 2. For user u_1, based on Theorem 1, the major axis for the elliptic region E_1 is 51.90. On the other hand, based on Theorem 2, the major axis is $295.53 - (93.33 + 68.84 + 81.78) = 51.58$. We take the best bound among them which is 51.58, the second one.

Each cell of ν_0, ν_1, ν_2, ν_3 and ν_4 also stores the set of POIs for which the minimum trip distance is obtained. For the sake of clarity we do not show them in the tables.

5.5 Algorithms

Algorithm 1: *GTS_Approach*$(S, D, \mathbb{C})$

```
   input  : $S$, $D$, $\mathbb{C}$
   output: A set of trips, $T$
1  $Initialize()$;
2  $InitDynTables(|S|, |\mathbb{C}|, \mathcal{V})$;
3  $ComputeTable(\nu_0)$;
4  $Enqueue(Q_p, root, MinD(G, root))$;
5  while $Q_p$ is not empty do
6      if $end = 1$ then
7          break;
8      $\{p, d_{min}(p)\} \leftarrow Dequeue(Q_p)$;
9      $r \leftarrow d_{min}(p)$;
10     if $p$ is not a POI then
11         foreach child node $p_c$ of $p$ do
12             $Enqueue(Q_p, p_c, MinD(G, p_c))$;
13     else if $\tau(p) \in \mathbb{C}$ and $p \in \bigcup_{i=1}^{n} E_i$ then
14         $P \leftarrow InsertPOI(p)$;
15         if $init = 0$ and $CheckInclude(P, \mathbb{C})$ then
16             $ComputeTrip(S, D, \mathbb{C}, P, \mathcal{V})$;
17             $init \leftarrow 1$;
18             $isup \leftarrow true$;
19         else if $init = 1$ then
20             $isup \leftarrow UpdateTrip(\tau(p), S, D, \mathbb{C}, p, \mathcal{V})$;
21     if $isup = true$ and $init = 1$ then
22         $\{T, Mx, Mi\} \leftarrow UpDynTables(|S|, |\mathbb{C}|, \mathcal{V})$;
23         $ellipregions \leftarrow UpEllipticRegions(T, Mx, Mi)$;
24     if $IsInCircle(G, r, ellipregions)$ then
25         $end \leftarrow 1$;
26 return $T$
```

Algorithm 1 shows the pseudocode of our approach to evaluate GTS queries for both Euclidean space and road networks. It takes the set of source and destination locations, S and D, respectively for a group of n members and the set of required m POI types $\mathbb{C}$ as input. The output is the set of n scheduled trips $T = \{T_1, T_2, \ldots, T_n\}$, where n trips together visit all POI types in $\mathbb{C}$ and no POI type is visited by more than one trip.

As the first step, using function $Initialize()$, Algorithm 1 initializes G to the geometric centroid of source and destination locations, a priority queue Q_p to $\emptyset$, and other variables as follows: $r = 0$, $end = 0$, $isup = false$, and $init = 0$. The variable r represents the radius of current known region. Flags end and $isup$ indicate whether the terminating condition is true and a user's trip has been updated, respectively. Variable $init$ is used to keep track between compute and update trip operations. $Initialize()$ also declares n elliptic regions for n users as $ellipregions = \{E_1, E_2, \ldots, E_n\}$, where the foci of each ellipse E_i is initialized to the source and destination locations of a user and the length of the major axis is set to ∞.

Function $InitDynTables(|S|, |\mathbb{C}|, \mathcal{V})$ initializes the set of dynamic tables $\mathcal{V} = \{\nu_0, \nu_1, \ldots, \nu_m\}$. After that $ComputeTable(\nu_0)$ computes the values for single member columns and combined member columns of the first dynamic table ν_0. The stored trip distances in ν_0 are Euclidean distances if the GTS is query is processed in the Euclidean space, and they are road network distances, otherwise.

The algorithm starts searching from the *root* of the R^*-tree and inserts the *root* with $MinD(G, root)$ into a priority queue Q_p. Q_p stores its elements in order of their minimum distances from G, $d_{min}(p)$ that are determined by Function $MinD(G, p)$. For both Euclidean space and road networks, $MinD(G, p)$ returns the minimum Euclidean distance between G and p, where p represents a POI or a minimum bounding rectangle of a R^*-tree node. After that the algorithm removes an element p along with $d_{min}(p)$ from Q_p. At this step, the algorithm updates r, the radius of current known region. If p represents a R^*-tree node, then algorithm retrieves its child nodes and enqueues them into Q_p. On the other hand, if p is a POI then it is added to candidate POI set P, if the POI type is specified in $\mathbb{C}$ and falls inside any user's ellipse E_i. The algorithm uses function $\tau(p)$ to determine the POI type of a POI p.

Function $CheckInclude(P, \mathbb{C})$ checks whether the POI set P contains at least one POI from each POI type in $\mathbb{C}$. When the initial POI set has been found, Function $ComputeTrip(S, D, \mathbb{C}, P, \mathcal{V})$ computes possible trips for all users and populates the single member columns of ν_1 to ν_m using our dynamic programming technique. The algorithm sets *init* to 1 and *isup* to *true*. As mentioned before, the stored trip distances in the dynamic tables are Euclidean distances if the GTS is query is processed in the Euclidean space, and they are road network distances, otherwise.

After computing the trips from the initial POI set, if the algorithm retrieves any new POI p, it uses Function $UpdateTrip(\tau(p), S, D, \mathbb{C}, p, \mathcal{V})$ to compute new trips using p and update the single member columns of ν_1 to ν_m, if new trips can improve the stored trip distances in the tables. The function also updates *isup* accordingly.

If *isup* is true and the initial set is already found (i.e., $init = 1$), Function $UpDynTables(|S|, |\mathbb{C}|, \mathcal{V})$ updates combined member columns of tables from ν_1 to ν_m based on the logic described in Section 5.4. The function takes n, m and the set of all dynamic tables $\mathcal{V}$ as input, updates the combined member columns of the dynamic tables and returns T, Mx and Mi, where T represents the scheduled trips, Mx and Mi represent the sets $\{T_{max_1}, \ldots, T_{max_n}\}$ and $\{T_{min_1}, \ldots, T_{min_n}\}$, respectively. T_{max_i} and T_{min_i} for $1 \leq i \leq n$ are defined in Section 5.2.

Then using $UpEllipticRegions(T, Mx, Mi)$, the algorithm updates the elliptic bound for all n users, where *ellipregions* represents the elliptic search regions of the users. The bounds for the elliptic search regions are determined using both Theorem 1 and 2. The algorithm checks the terminating condition of our GTS queries using Function $IsInCircle(G, r, ellipregions)$. This function checks whether all n elliptic search regions is included by the current circular known region or not. If the terminating condition is true, the algorithm updates the terminating flag *end* to 1. At the end of the algorithm, it returns scheduled trips T for n users that provide the minimum total distance.

6 A Straightforward Approach

To the best of our knowledge, we introduce GTS queries in spatial databases and thus, there exists no approach to process GTS queries in the literature. To validate the efficiency of our proposed approach in experiments, we develop a straightforward approach for processing GTS queries, S-GTS, using existing trip planning algorithms.

A straightforward way to process a GTS query would be independently evaluating optimal trips for every group member and for all possible combinations of POI types, and then selecting n trips that together satisfies the conditions of GTS queries and provides the minimum total trip distance for the group. This approach requires multiple independent searches into the database and accesses same POIs multiple times.

```
Algorithm 2: S-GTS_Approach($S, D, \mathbb{C}$)
   input  : $S$, $D$, $\mathbb{C}$
   output: A set of trips, $T$
1  $m \leftarrow |\mathbb{C}|$;
2  $n \leftarrow |S|$;
3  $InitDynTables(|S|, |\mathbb{C}|, \mathcal{V})$;
4  $ComputeTable(\nu_0)$;
5  for group member $u_i$ do
6      for $g \leftarrow 1$ to $m$ do
7          foreach member $t_c$ of ${}^{\mathbb{C}}C_g$ do
8              $\nu_g[t_c][\{u_i\}] \leftarrow GTP(s_i, d_i, t_c)$;
9  $\{T, Mx, Mi\} \leftarrow UpDynTables(n, m, \mathcal{V})$;
10 return $T$
```

Algorithm 2 shows the pseudocode of the S-GTS approach to evaluate GTS queries in the Euclidean and road network spaces. It takes the following parameters as input: the set of source and destination locations, S and D, respectively, for a group of n members and the set of required m POI types $\mathbb{C}$. The output is the set of n scheduled trips $T = \{T_1, T_2, \ldots, T_n\}$, where n trips together visit all POI types in $\mathbb{C}$ and no POI type is visited by more than one trip.

In the first step, Algorithm 2 initializes the dynamic tables ν_0 to ν_m using the function $InitDynTables(|S|, |\mathbb{C}|, \mathcal{V})$, which we mentioned in Section 5.4. After that $ComputeTable(\nu_0)$ computes single member columns and combined member columns of the first dynamic table ν_0. After updating table ν_0, for each member u_i of the group and for each dynamic table ν_g, the algorithm calculates trips for ${}^{m}C_g$ possible sets of POI types using function $GTP(s_i, d_i, t_c)$, and populates the dynamic tables ν_1 to ν_m. The function takes the source and destination locations of u_i, and a set of POI types t_c from $\mathbb{C}$ as input and returns the optimal trip with the trip distance in the Euclidean space or road networks, where the trip starts from s_i, passes through POI types in t_c and ends at d_i. The $GTP(s_i, d_i, t_c)$ function considers all possible orders of POI types in t_c while computing trip distances and returns the minimum one. For the function $GTP(s_i, d_i, t_c)$, any existing trip planning algorithm or group trip planning algorithm (by assuming one group member) can be used. In our experiment, we use the most recent and efficient group trip planning algorithm [7] for this purpose. However, in the S-GTS approach, the function $GTP(s_i, d_i, t_c)$ is called multiple times, and a same POI may be accessed in the database more than once. On the other hand, our GTS approach requires a single traversal on the database and ensures that a single POI is accessed once in the database.

Finally, the algorithm uses the same function $UpDynTables(n, m, \mathcal{V})$ as Algorithm 1 to select the final n scheduled trips for the group. The function updates the combined member columns of the dynamic tables from ν_1 to ν_m, and returns T, and Mx and Mi, where T represents the scheduled trips, Mx and Mi are not used for the S-GTS approach.

Although for the S-GTS approach, we apply the similar dynamic programming that we use for our GTS approach in Section 5, two approaches are different. In the S-GTS approach, we use the dynamic programming technique *once* to find the final scheduled n trips from the already calculated optimal trips of users. On the other hand, the GTS approach incrementally retrieves POIs from the database, calculates the trips of users based on the retrieved POIs, and applies the dynamic programming technique *every time* with the retrieval of a new POI to check whether the new POI can improve the scheduled trips.

7 Experiments

In this section, we evaluate the performance of our approach for processing GTS queries through extensive experiments. Since there is no existing work for GTS queries in the literature, we compare our proposed GTS approach with the straightforward approach (S-GTS) discussed in Section 6 by varying a wide range of parameters.

We evaluate our approach in both Euclidean and road network dataspaces using synthetic and real world datasets. For the real dataset, we used California [2] dataset that contains 87635 POIs of 63 different types. The road network of California has 21048 nodes and 21693 edges. We generated the synthetic datasets of POIs of different types using the uniform random distribution. The whole data space is normalized to 1000x1000 sq. units for both real and synthetic datasets. An R^*-tree is used to store all the POIs of a dataset and a in-memory graph data structure is used to store the road network.

We performed several set of experiments by varying the following parameters: (i) the group size n, (ii) the number of specified POI types m in a GTS query, (iii) the query area A, i.e., the minimum bounding rectangle covering the source and destination locations, and (iv) the dataset size d_s (only in the Euclidean space).

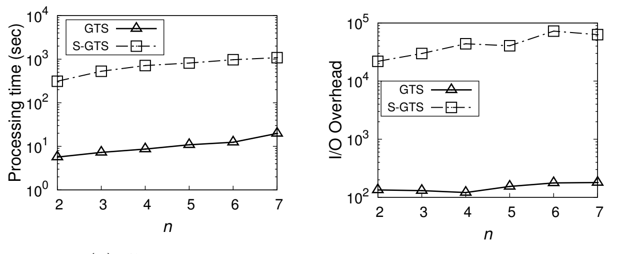
Table 12: Parameter settings

Parameter	**Values**	**Default**
Group size(n)	2, 3, 4, 5, 6, 7	3
Number of POI types (m)	2, 3, 4 , 5 , 6	4
Query area(A) (in sq. units)	50x50, 100x100, 150x150, 200x200, 250x250, 300x300	100x100
Dataset size(d_s) (number of POIs in thousands)	5, 10, 20, 40, 80, 160	-
Dataset distribution	Uniform	-

Table 12 shows the range and default values used for each parameter. To observe the effect of a parameter in an experiment, the value of the parameter is varied within its range, and other parameters are set to their default values. We use an Intel Core i5 machine with 2.30 GHz CPU and 4GB RAM to run the experiments. For each set of experiments, we measure two performance metrics: the average processing time and average I/O overhead (I/O access in R^*-tree). The metrics are measured by running 100 independent GTS queries having random source and destination locations, and then taking the average of processing time and I/O access. Since both GTS and S-GTS approaches require the same amount of storage for storing dynamic tables, we do not show them in our experiments.

7.1 Euclidean Space

Effect of group size (n): Figures 9(a) and 9(b) show the processing time and I/O access, respectively, for our GTS and S-GTS approaches. We observe that both processing time and I/O access slightly increase with the increase of the group size. Our GTS approach requires significantly less processing time and I/O access than the S-GTS approach, which is expected. The S-GTS approach computes the optimal trips for each group member and for every possible combination of POI types independently, and thus, accesses the same POIs multiple times in the database. On the other hand, our GTS approach accesses a POI in the database only once and gradually refines the search regions based on the scheduled trips using the dynamic programming technique.

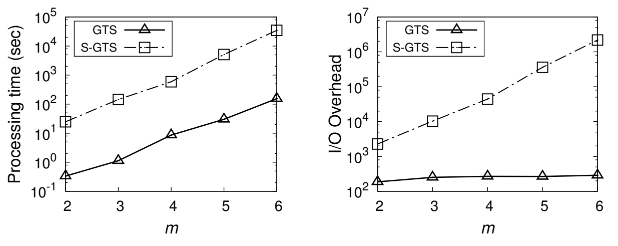


(a) CPU time (b) I/O access

Figure 9: Effect of group size (n) (California dataset)

Effect of m: Figures 10(a) and 10(b) show that the processing time and I/O access, respectively, increase with the increase of m. The results show that our GTS approach outperforms the S-GTS approach by a large margin in terms of both I/O access and processing time. Specifically, the improvement for the I/O access is more pronounced for the larger values of m. We observe in Figure 10(b) that the I/Os required by the GTS approach remains almost constant, and the number of I/O access for the S-GTS approach sharply increases with the increase of m. The reason is as follows. For the change of m to $m+1$, the number of independent trip computations in the S-GTS approach for each group member increases by $\sum_{y=0}^{m+1}({}^{m+1}C_y) - \sum_{y=0}^{m}({}^{m}C_y)$, whereas the I/O access of the GTS approach depends on the size of its search region. For an additional POI type, the search region only slightly increases since the $AggTripDist$ and T_{max_i} for any user u_i increase by only a small amount.

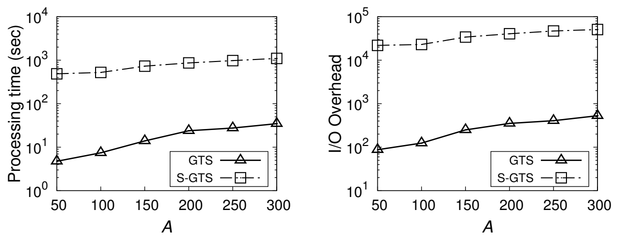
Effect of A: Figures 11(a) and 11(b) show experimental results for different values of the query area A. We see that for both approaches, the processing time and I/O access increase with the increase of A. This is because the POI search region becomes large if the source and destination



(a) CPU time (b) I/O access

Figure 10: Effect of number of types (m) (California dataset)

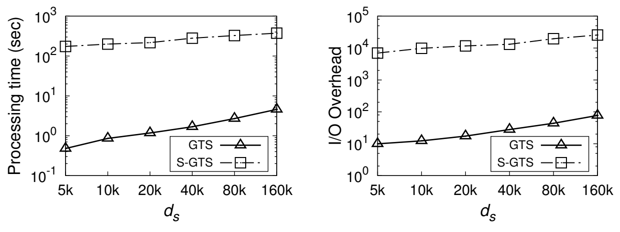
locations are distributed in a large area of the total space. For both metrics, our GTS approach outperforms the S-GTS approach, which is for the similar reasons mentioned for the experiments of varying n.



(a) CPU time (b) I/O access

Figure 11: Effect of query area (A) (California dataset)

Effect of dataset size (d_s): In this experiment, we examine the performance difference of the two approaches with respect to data set size (d_s). Figures 12(a) and 12(b) show that as the size increases, processing time and I/O access increase for both approaches, which is expected. Like other experiments, the GTS approach takes much less processing time (approx. 192 times) and I/O access (approx. 570 times) than the S-GTS approach for any dataset size.



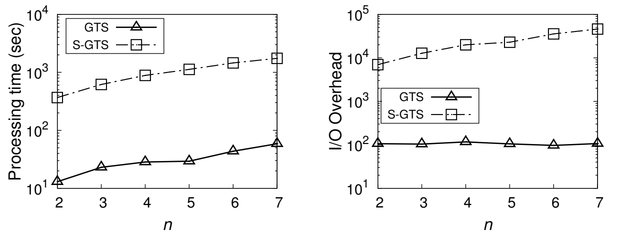
(a) CPU time (b) I/O access

Figure 12: Effect of dataset size (d_s) (Synthetic dataset)

7.2 Road Networks

Experimental results for processing GTS queries in road networks using our proposed approach, GTS, show similar performance and trends like the Euclidean space except that the GTS approach requires on average 6.6 times more query processing time compared to the required processing time in the Euclidean space.

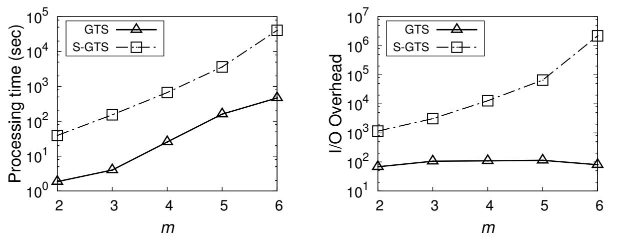
Effect of group size (n): Figures 13(a) and 13(b) show that the query processing time and I/O access increase with the increase of group size n for both approaches, GTS and S-GTS. This is because the number of road network distance computations increase with the increase of n. On the other hand, with the increase of group size n, for our GTS approach, the number of I/O access slightly changes, whereas for the S-GTS approach, the I/O access increases significantly due to the access of same POIs multiple times. For both metrics, the GTS approach outperforms the S-GTS approach.



(a) CPU time (b) I/O access

Figure 13: Effect of group size (n) (California dataset)

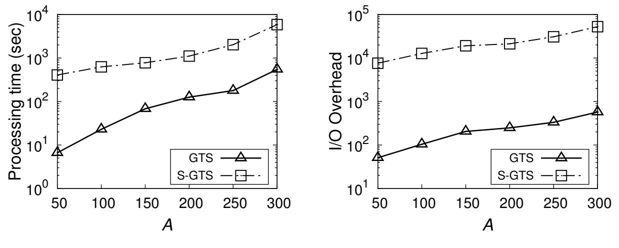
Effect of $\boldsymbol{m}$***:*** Figures 14(a) and 14(b) show the performance of the GTS approach and the S-GTS approach for varying the total number of POI types m. We observe that the performance trends are similar to those for the Euclidean space. For any number of types, the GTS approach outperforms the S-GTS approach in terms of both I/O access and processing time.



(a) CPU time (b) I/O access

Figure 14: Effect of number of types (m) (California dataset)

Effect of $\boldsymbol{A}$***:*** Figures 15(a) and 15(b) show that both query processing time and I/O access increase with the increase of A for both approaches, and the GTS approach performs significantly better than the S-GTS approach for both metrics.



(a) CPU time (b) I/O access

Figure 15: Effect of query area (A) (California dataset)

8 Conclusion

In this paper, we have introduced a new type of query, a group trip scheduling (GTS) query in spatial databases that enables a group of users to schedule multiple trips among themselves with the minimum total trip distance of the group members. We propose the first solution to evaluate GTS queries in both Euclidean space and road networks. The refinement technique of the POI search space and the dynamic approach to schedule trips among group members are the key ideas behind the efficiency of our approach. Experiments show that our approach is on average 107 times faster and requires on average 635 times less I/Os than the straightforward approach for the Euclidean space. For road networks, we observed that our approach requires on average 30 times less processing time and 1768 times less I/O access than the straightforward approach. In the future, we aim to protect location privacy [9, 10] of users for GTS queries.

Acknowledgments

This research is supported by Bangladesh University of Engineering and Technology (BUET).

9 References

[1] http://rtreeportal.org/.

[2] California road network data. https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm.

[3] E. Ahmadi and M. A. Nascimento. A mixed breadth-depth first search strategy for sequenced group trip planning queries. In *MDM*, pages 24–33, 2015.

[4] T. Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209 – 219, 2006.

[5] H. Chen, W. Ku, M. Sun, and R. Zimmermann. The multi-rule partial sequenced route query. In *SIGSPATIAL*, page 10, 2008.

[6] G. Gutin and D. Karapetyan. A memetic algorithm for the generalized traveling salesman problem. *Natural Computing*, 9(1):47–60, 2010.

[7] T. Hashem, S. Barua, M. E. Ali, L. Kulik, and E. Tanin. Efficient computation of trips with friends and families. In *CIKM*, pages 931–940, 2015.

[8] T. Hashem, T. Hashem, M. E. Ali, and L. Kulik. Group trip planning queries in spatial databases. In *SSTD*, pages 259–276, 2013.

[9] T. Hashem and L. Kulik. Safeguarding location privacy in wireless ad-hoc networks. In *Ubicomp*, pages 372–390, 2007.

[10] T. Hashem, L. Kulik, and R. Zhang. Privacy preserving group nearest neighbor queries. In *EDBT*, pages 489–500, 2010.

[11] G. Laporte. A concise guide to the traveling salesman problem. *JORS*, 61(1):35–40, 2010.

[12] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S. Teng. On trip planning queries in spatial databases. In *SSTD*, pages 273–290, 2005.

[13] H. Li, H. Lu, B. Huang, and Z. Huang. Two ellipse-based pruning methods for group nearest neighbor queries. In *GIS*, pages 192–199. ACM, 2005.

[14] H. Li, H. Lu, B. Huang, and Z. Huang. Two ellipse-based pruning methods for group nearest neighbor queries. In *International Workshop on GIS*, pages 192–199, 2005.

[15] J. Li, Q. Sun, M. Zhou, and X. Dai. A new multiple traveling salesman problem and its genetic algorithm-based solution. In *SMC*, pages 627–632, 2013.

[16] Y. Ohsawa, H. Htoo, N. Sonehara, and M. Sakauchi. Sequenced route query in road network distance based on incremental euclidean restriction. In *DEXA*, pages 484–491, 2012.

[17] S. Samrose, T. Hashem, S. Barua, M. E. Ali, M. H. Uddin, and M. I. Mahmud. Efficient computation of group optimal sequenced routes in road networks. In *MDM*, pages 122–127, 2015.

[18] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *VLDB J.*, 17(4):765–787, 2008.

[19] S. C. Soma, T. Hashem, M. A. Cheema, and S. Samrose. Trip planning queries with location privacy in spatial databases. *World Wide Web*, 2016.

[20] W. Zhou and Y. Li. An improved genetic algorithm for multiple traveling salesman problem. In *Informatics in Control, Automation and Robotics (CAR)*, volume 1, pages 493–495, 2010.