M.Sc. Engg. Thesis

Scheduling Multiple Trips for a Group in Spatial Databases

by

Roksana Jahan

Submitted to

Department of Computer Science and Engineering

in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science and Engineering

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology (BUET)

Dhaka - 1000

January 2017

Dedicated to my loving parents

AUTHOR'S CONTACT

Roksana Jahan
Senior Software Engineer
Email: munia0505064@gmail.com

The thesis titled “Scheduling Multiple Trips for a Group in Spatial Databases”, submitted by Roksana Jahan, Roll No. **0412052076P**, Session April 2012, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on January 24, 2017.

Board of Examiners

1. ______________________________
Dr. Tanzima Hashem — Chairman
Associate Professor — (Supervisor)
Department of CSE, BUET, Dhaka - 1000.

2. ______________________________
Dr. Mohammad Mahfuzul Islam — Member
Head and Professor — (Ex-Officio)
Department of CSE, BUET, Dhaka - 1000.

3. ______________________________
Dr. Md. Abul Kashem Mia — Member
Professor
Department of CSE, BUET, Dhaka - 1000.

4. ______________________________
Dr. Abu Sayed Md. Latiful Hoque — Member
Professor
Department of CSE, BUET, Dhaka - 1000.

5. ______________________________
Dr. Nova Ahmed — Member
Associate Professor — (External)
Department of ECE
North South University, Dhaka - 1229.

Candidate's Declaration

This is hereby declared that the work titled "Scheduling Multiple Trips for a Group in Spatial Databases" is the outcome of research carried out by me under the supervision of Dr. Tanzima Hashem, in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka - 1000. It is also declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

Roksana Jahan
Candidate

Acknowledgment

First and foremost I offer my sincerest gratitude to my supervisor, Dr. Tanzima Hashem, who has supported me throughout my thesis with her patience, motivation, enthusiasm, and immense knowledge. She helped me a lot in every aspect of this work and guided me with proper directions whenever I sought one. I could not have imagined having a better supervisor and mentor for my M.Sc. study and research. Her patient hearing of my ideas, critical analysis of my observations and detecting flaws (and amending thereby) in my thinking and writing have made this thesis a success.

I would also want to thank the members of my thesis committee for their valuable suggestions. I thank Dr. Md. Abul Kashem Mia, Dr. Abu Sayed Md. Latiful Hoque and specially the external member Dr. Nova Ahmed.

My sincere thanks also goes to Sukarna Barua for helping me in my thesis work, specially in implementation. His insightful comments and suggestions regarding performance improvement of my thesis implementation helped me to overcome flaws and to improve my thesis work a lot.

In this regard, I remain ever grateful to my beloved parents, who always exists as sources of inspiration behind every success of mine I have ever made.

Abstract

Planning user trips in an effective and efficient manner has become an important topic in recent years. In this thesis, 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 types of points of interests (POIs) such as restaurants and shopping centers to perform many tasks. A GTS query distributes the tasks among group members in an optimized manner. Given source and destination locations of n group members, a GTS query schedules n individual trips such that each POI type is included in a scheduled trip and the aggregate trip overhead distance for visiting required POI types is minimized. The aggregate trip overhead distance can be either the summation or the maximum of the trip overhead distances of group members. Each trip starts at a member's source location, goes through any number of POI types, and ends at the member's destination location. 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. The trip overhead distance of a group member is measured by deducting the distance between the source and destination locations of a group member from the trip distance. We develop an efficient approach to process GTS queries and variants 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.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 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. Beside families, people are also part of many social, religious, cultural, educational, economical, and professional groups and organizations. They may need to organize different types of group events like annual team outing, annual cultural programs, science fair, and inter-educational institution competitions. Similar to the members of a family, organizers of an event may need to visit different points of interests (POIs) like supermarkets, banks, and restaurants to perform many tasks. For example, to organize an annual cultural program in an educational institution, organizers may need to go to a supermarket for groceries, a catering house to order food, a bank to withdraw or deposit money, and a shopping mall to buy gift items.

In reality, all family or organizing members do not need to visit every POI and they can distribute the tasks among themselves. For example, any member of a family can buy groceries from a supermarket. Furthermore, 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. This scenario motivates 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 aggregate trip overhead distance. 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

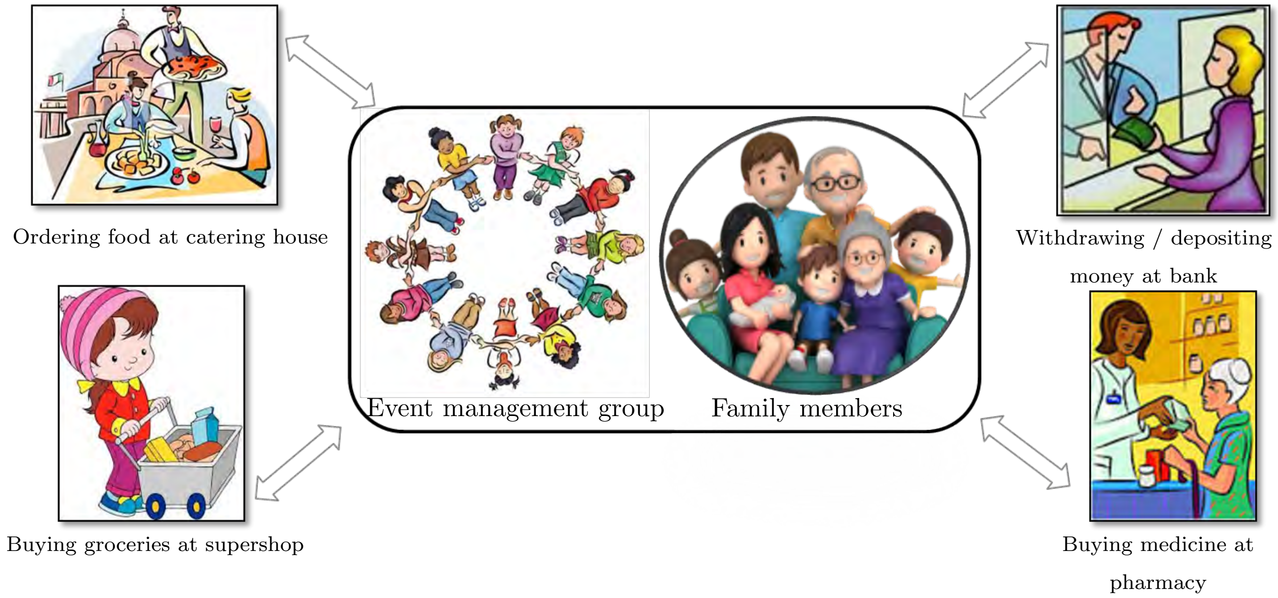


Figure 1.1: Different types of tasks in real life

group, and the aggregate trip overhead distance of n group members is minimized. The aggregate trip overhead distance can be either the summation or the maximum of the trip overhead distances of group members. 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. The trip overhead distance of the group member is the additional distance required to visit any number of POI types. Specifically, the trip overhead distance of a group member is measured by deducting the distance between the source and destination locations of a group member from the trip distance. If the aggregate trip overhead 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.

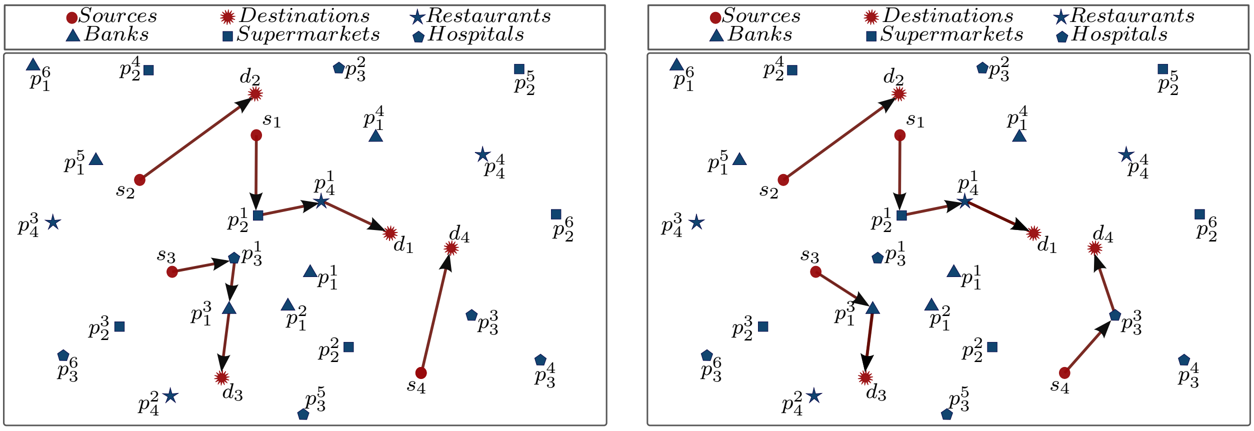
In a GTS query, a group member is flexible to visit any number of POI types, if it minimizes the aggregate trip overhead distance for the group. Sometimes, it may happen that group members want to visit equal number of POI types. To address such scenario, we introduce a new variant of GTS query, *a Uniform GTS (UGTS) query*, where group members visit equal number of POI types. In this thesis, we propose an efficient approach to process GTS and UGTS queries for both Euclidean space and road networks.

1.1 GTS Queries

Formally, in a Group Trip Scheduling (GTS) query, we have a group of n members with specific n source-destination pairs and m required POI types that need to be visited by a member of the group. A GTS query schedules n trips such that

- ✓ Each trip starts and ends at a group member's source and destination locations, respectively.
- ✓ Each of the m POI types is visited by a group member, and each trip visits one or more POI types from m required POI types.
- ✓ The aggregate trip overhead distance is minimized.

The aggregated trip overhead distance can be either the total or the maximum of the trip overhead distances of group members that are measured using aggregate functions SUM and MAX, respectively.



a) *Scheduled trips with the minimum total trip overhead distance of the group*

b) *Scheduled trips with the minimum maximum trip overhead distance of the group*

Figure 1.2: An example GTS query for aggregate function SUM and MAX

In Figures 1.2(a-b), 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

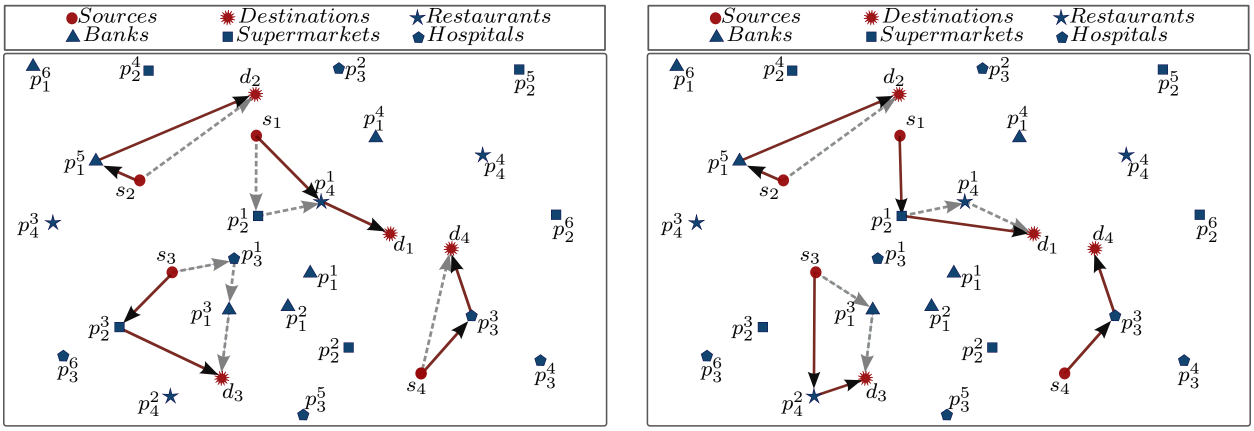
Table 1.1: Scheduled trips for SUM

$s_1 \rightarrow p_2^1 \rightarrow p_4^1 \rightarrow d_1$
$s_2 \rightarrow d_2$
$s_3 \rightarrow p_3^1 \rightarrow p_1^3 \rightarrow d_3$
$s_4 \rightarrow d_4$

Table 1.2: Scheduled trips for MAX

$s_1 \rightarrow p_2^1 \rightarrow p_4^1 \rightarrow d_1$
$s_2 \rightarrow d_2$
$s_3 \rightarrow p_1^3 \rightarrow d_3$
$s_4 \rightarrow p_3^3 \rightarrow d_4$

(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 aggregate trip overhead distance, where each POI type is included in a single trip. Figures 1.2(a) and 1.2(b) show four scheduled trips listed in Tables 1.1 and 1.2 for aggregate functions SUM and MAX, respectively, for a GTS query. Each trip starts and ends at a member's source and destination locations, and four trips together visit all required types of places, i.e., a bank, a supermarket, a restaurant, and a hospital.



a) *Uniform scheduled trips with the minimum total trip overhead distance of the group*

b) *Uniform scheduled trips with the minimum maximum trip overhead distance of the group*

Figure 1.3: An example UGTS query for aggregate function SUM and MAX

A group may impose constraints while scheduling the trips. A trip returned by a GTS query may include any number of POI types ranging from 0 to m, whereas sometimes group members may want the uniform distribution of the tasks among themselves, i.e., they require to visit equal number of POI types. To address such a scenario, we introduce a new variant of a GTS query, a uniform GTS

(UGTS) query, that schedules trips in a uniform manner. Let e be the number of POI types that each group member visits for uniform distributions of POI visits among group members. If m is a multiple of n, then $e = \lfloor \frac{m}{n} \rfloor$. If m is not a multiple of n, then $m \mod n$ number of group members visit $e = \lfloor \frac{m}{n} \rfloor + 1$ number of POI types, and the remaining group members visit $e = \lfloor \frac{m}{n} \rfloor$ number of POI types.

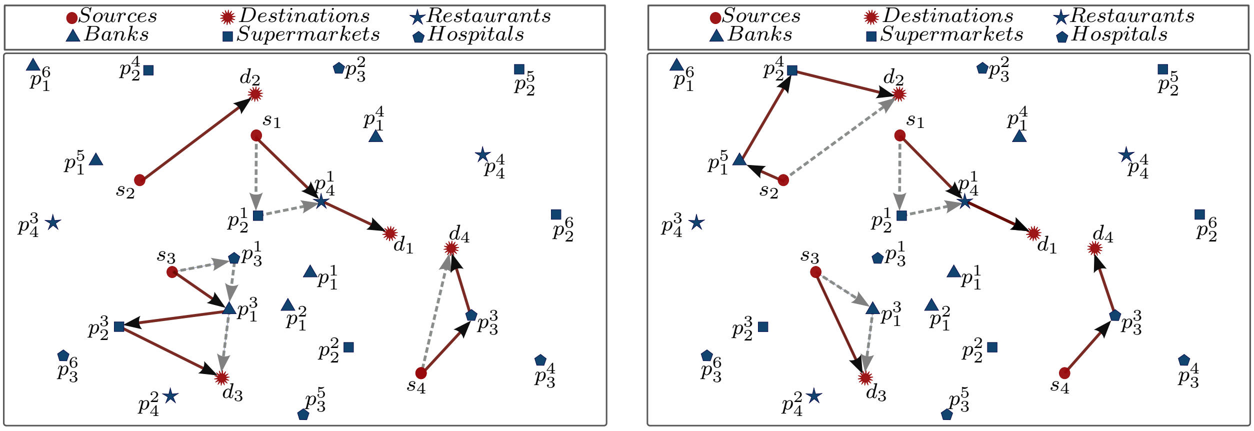
Table 1.3: Uniform scheduled trips for SUM

$s_1 \rightarrow p_4^1 \rightarrow d_1$
$s_2 \rightarrow p_1^5 \rightarrow d_2$
$s_3 \rightarrow p_2^3 \rightarrow d_3$
$s_4 \rightarrow p_3^3 \rightarrow d_4$

Table 1.4: Uniform scheduled trips for MAX

$s_1 \rightarrow p_2^1 \rightarrow d_1$
$s_2 \rightarrow p_1^5 \rightarrow d_2$
$s_3 \rightarrow p_4^2 \rightarrow d_3$
$s_4 \rightarrow p_3^3 \rightarrow d_4$

Considering the same example scenario shown in Figures 1.2(a-b), for a UGTS query, each group member needs to visit one POI type as there are four members in the group and the total number of required POI types to visit is four. Figures 1.3(a) and 1.3(b) show four scheduled trips listed in Tables 1.3 and 1.4 for aggregate functions SUM and MAX, respectively, for a UGTS query.



a) *Scheduled trips with dependency between POIs bank and supermarket for the minimum total trip overhead distance of the group*

b) *Scheduled trips with dependency between POIs bank and supermarket for the minimum maximum trip overhead distance of the group*

Figure 1.4: An example GTS query with dependencies among POIs for aggregate function SUM and MAX

Similar to the equal distribution of tasks, the group members may also need to fix the maximum/minimum/fixed number of tasks that a group member can perform. For such constraints, the answer for GTS and UGTS queries may change. In addition to fixing the number of POI types, group members can also impose constraints considering the dependencies between POIs, and/or dependencies between POIs and group members for both GTS and UGTS queries.

Table 1.5: Scheduled trips with dependency between POIs bank and supermarket for SUM

$s_1 \rightarrow p_4^1 \rightarrow d_1$
$s_2 \rightarrow d_2$
$s_3 \rightarrow p_1^3 \rightarrow p_2^3 \rightarrow d_3$
$s_4 \rightarrow p_3^3 \rightarrow d_4$

Table 1.6: Scheduled trips with dependency between POIs bank and supermarket for MAX

$s_1 \rightarrow p_4^1 \rightarrow d_1$
$s_2 \rightarrow p_1^5 \rightarrow p_2^4 \rightarrow d_2$
$s_3 \rightarrow d_3$
$s_4 \rightarrow p_3^3 \rightarrow d_4$

For example, a group may need to first visit a bank to withdraw money before visiting a supermarket. Thus, the sequence of visiting POI types may be fixed in some cases. Furthermore, the group member who visits the bank needs to visit the supermarket. Considering the same example scenario shown in Figures 1.2(a-b), Figures 1.4(a) and 1.4(b) show four scheduled trips listed in Tables 1.5 and 1.6 for aggregate functions SUM and MAX, respectively, for a GTS query by considering the dependency between the bank and the supermarket.

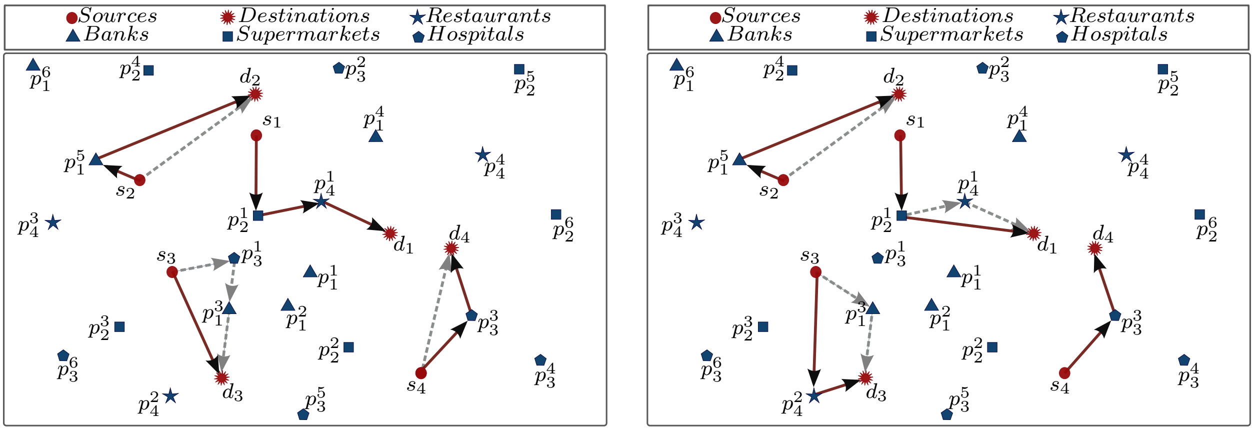
Table 1.7: Scheduled trips with dependency between group member u_2 and POI bank for SUM

$s_1 \rightarrow p_2^1 \rightarrow p_4^1 \rightarrow d_1$
$s_2 \rightarrow p_1^5 \rightarrow d_2$
$s_3 \rightarrow d_3$
$s_4 \rightarrow p_3^3 \rightarrow d_4$

Table 1.8: Scheduled trips with dependency between group member u_2 and POI bank for MAX

$s_1 \rightarrow p_2^1 \rightarrow d_1$
$s_2 \rightarrow p_1^5 \rightarrow d_2$
$s_3 \rightarrow p_4^2 \rightarrow d_3$
$s_4 \rightarrow p_3^3 \rightarrow d_4$

There can be also a dependency between a POI and a group member. For example, in the same example scenario shown in Figures 1.2(a-b), assume that group member u_2 needs to visit the bank. Figures 1.5(a) and 1.5(b) show four scheduled trips listed in Tables 1.7 and 1.8 for aggregate functions



a) *Scheduled trips with dependency between group member u_2 and POI bank for the minimum total trip overhead distance of the group*

b) *Scheduled trips with dependency between group member u_2 and POI bank for the minimum maximum trip overhead distance of the group*

Figure 1.5: An example GTS query with dependencies among members and POIs for aggregate function SUM and MAX

SUM and MAX, respectively, for a GTS query by considering the dependency between the bank and group member u_2.

1.2 Research Challenges and Solution Overview

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 [1]. 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 [2–6] 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. A POI outside the search region cannot be a part of the optimal scheduled trips. The smaller the search region, the efficient the technique to evaluate GTS queries in spatial databases. On the other hand, the smaller the number of POI distributions that a technique considers while scheduling trips, the efficient the GTS query processing approach is. 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.

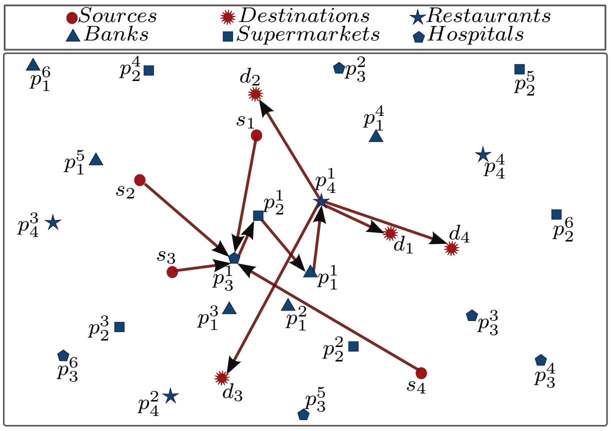


Figure 1.6: An example of a GTP query

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 [4] 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 [7] 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 Figures 1.2(a-b), in Figure 1.6 we show the resultant trips for a GTP query, where the group members visit all required POI types together with minimum total trip distance. A GTS query is also different from traveling salesman problem (TSP) [8] and its variants [9–12]. 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.

Thus there are no existing work to schedule trips for a group of members in spatial databases and we propose the first approach to evaluate GTS query in spatial databases.

1.3 Contributions

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

- We introduce a new type of query, the group trip scheduling (GTS) query in spatial databases.
- 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 propose a variant of GTS queries, a uniform GTS (UGTS) query and provide solution for processing UGTS queries.
- We extend our approach for processing GTS and UGTS queries with constraints like dependencies between POIs and 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.

1.4 Outline

The remaining part of the thesis is organized as follows:

In Chapter 2, we formulate Group Trip Scheduling GTS queries and its variant in spatial databases and give and overview of our system.
In Chapter 3, we outline the research work related to this problem.
In Chapter 4, we propose and explain our GTS query processing approach and extend the proposed approach for the GTS query variant.
In Chapter 5, we present our algorithm to evaluate (GTS) queries and its variant in spatial databases.
In Chapter 6, we discuss a straightforward approach for processing GTS queries and its variant.
In Chapter 7, we show experimental results using both real and synthetic datasets.
In Chapter 8, we conclude the thesis with possible directions for future work.

Chapter 2

Problem Formulation

In this chapter, we first formulate Group Trip Scheduling (GTS) queries and variant, and describe the notions that we use throughout the thesis. Then we give an overview of our system.

2.1 Group Trip Scheduling (GTS) Queries

In a Group Trip Scheduling (GTS) query, a group of members specify their independent source and destination locations and specific POI types that they want to visit. A GTS query schedules trips for every member of the group with the minimum aggregate trip overhead distance, where each trip starts from a member's source location, goes through any number of POI types, and ends at corresponding member's destination location. 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 $D = \{d_1, d_2, \ldots, d_n\}$, a set of m POI types $\mathbb{C} = \{c_1, c_2, \ldots, c_m\}$ and an aggregate function f, a GTS query returns a set of n trips, $T = \{T_1, T_2, \ldots, T_n\}$ that minimizes the aggregate trip overhead distance, $AggTripOvDist$ of group members, where T_i corresponds to a trip of group member u_i, and each 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 $m = |\mathbb{C}| = \sum_{i=1}^{n} |A_i|$. 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 $TripDist_i$.

On the other hand, the trip overhead distance of a group member is the additional distance required to visit any number of POI types. The trip overhead distance of group member u_i is measured by deducting the distance between the source (s_i) and destination (d_i) locations from the trip distance $TripDist_i$. Thus, the trip overhead distance of group member u_i is computed as $(TripDist_i - Dist(s_i, d_i))$.

An aggregate function f could be SUM and MAX. If f represents SUM, the total trip overhead distance of group members is measured as $AggTripOvDist = \sum_{i=1}^{n} (TripDist_i - Dist(s_i, d_i))$. If f represents MAX, the maximum trip overhead distance of group members is measured as $AggTripOvDist = \max_{i=1}^{n} (TripDist_i - Dist(s_i, d_i))$

Group members may have constraints while scheduling the trips for a GTS query. It is a common scenario that group members may want the uniform distribution of POI visits among group members. To address such a scenario, we propose a uniform GTS (UGTS) query. In a UGTS query, we assume that each group member visits an equal number of POI types. If m is a multiple of n, then $e = \lfloor \frac{m}{n} \rfloor$. If m is not a multiple of n, then $m \mod n$ number of group members visit $e = \lfloor \frac{m}{n} \rfloor + 1$ number of POI types, and the remaining group members visit $e = \lfloor \frac{m}{n} \rfloor$ number of POI types. To explain our approach for processing UGTS queries later in this thesis, for the sake of simplicity, we assume that

m is a multiple of n and each group member visits $e = \lfloor \frac{m}{n} \rfloor$ number of POI types.

Sometimes, the group members may also need to fix the maximum/minimum/fixed number of tasks that a group member can perform. Furthermore, there can be dependencies among POIs, and/or dependencies among POIs and group members for both GTS and UGTS queries. For example, it may be required that the bank and supermarket need to be visited by a same member and the bank may need to be visited before the supermarket. The group member, who visits the bank, withdraws money from the bank, and then buy groceries from the supermarket using the money. In some cases, it may also require that a specific member needs to visit the bank to withdraw the money. In this thesis, we develop an approach that can process both GTS and UGTS queries by considering the dependencies between POIs and/or between a POI and a group member.
Table 2.1 summarizes the notations that we use in the rest of the thesis.

Table 2.1: Notations and their meanings

$U = \{u_1, u_2, u_3, \ldots, u_n\}$	A set of n users
$S = \{s_1, s_2, s_3, \ldots, s_n\}$	A set of source locations of n users in the group
$D = \{d_1, d_2, d_3, \ldots, d_n\}$	A set of destination locations of n users in the group
$\mathbb{C} = \{c_1, c_2, c_3, \ldots, c_m\}$	A set of m POI types
$\mathbb{P}$	The set of POIs of different types in a 2-dimensional space
$Dist(x, y)$	The distance between two point locations x and y
T_i	Trip for a user u_i
A_i	A set of POIs visited by T_i
$TripDist_i$	The trip distance of T_i
f	Aggregate function(SUM or MAX)

2.2 System Overview

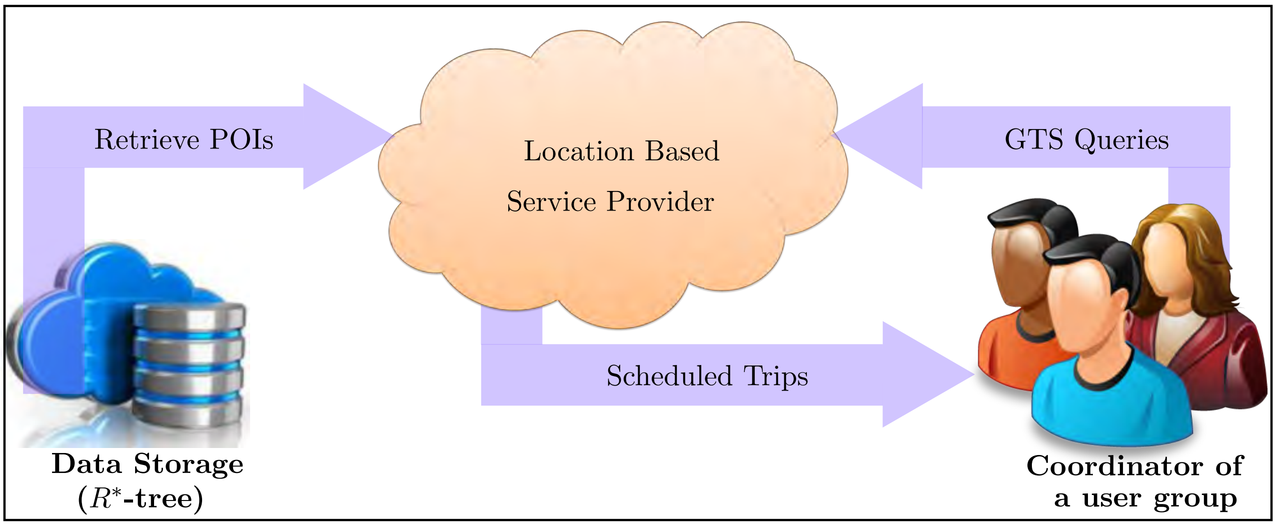


Figure 2.1: System architecture

Figure 2.1 shows an overview of the system architecture. The coordinator of a group sends a GTS or UGTS 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 need to visit. If the group members want to impose any type of constraints, the group coordinator also provides that information to the LSP as input. POI information is indexed using an R^*-tree in the data storage of the LSP. The LSP incrementally retrieves POIs from the database based on the input information, processes GTS or UGTS queries, and returns the scheduled trips to the coordinator of the group that minimizes the aggregate trip overhead distance of the group members.

Chapter 3

Related Work

In this chapter, we discuss the work related to our research problem. We categorize existing related research into two parts: trip and route planning, and traveling salesman problems. Trip planning techniques exist for both single user and group in the literature. In Section 3.1, we discuss existing approaches for planning trip and routes for a single user. In Section 3.2, we discuss existing group trip planning algorithms. In Section 3.3, we discuss the the solutions for the traveling salesman problem and variants. Finally, in Section 3.4, we show how are elliptical search space refinement techniques differ from existing ones in the literature.

3.1 Single User Trip and Route Planning Algorithms

Trip planning (TP) queries have been introduced in [4] 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 [6], 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 [2], 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 [4, 6] of trip planning queries. In [13], 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.

Besides trip planning algorithms, there exist a number of approaches [14–17] for planning routes between the source and destination locations of users. For answering continuous route planning queries over a road network, in [14], the authors have proposed two new classes of approximate techniques: a proximity-based algorithm and K candidate paths algorithms. The proximity-based algorithm recomputes the optimal route when more than some fraction of road delays change within a bounding ellipse, whereas the K candidate-path algorithm computes a set of K possible routes and periodically re-evaluates the best route as the road delays change. In [15], the authors have developed an approach that the shortest path for a group of queries sharing a common travel path. The focus of this approach is to reduce cost for the evaluation of a large number of simultaneous path queries. In [16], the authors have developed algorithms for processing path queries with constraints like finding the shortest path in a road network that avoids toll roads and low overpasses. In [17], the authors have focused on both travel time and energy cost while computing the routes on a scale road network.

3.2 Group Trip Planning Algorithms

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 [7]. In [18, 19], the authors proposed efficient algorithms to process GTP queries for a fixed sequence of visiting POI types. In [3], 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.

Besides GTP queries, group nearest neighbor (GNN) queries have been proposed in the literature, where a group of members visit a POI such that the aggregate distance is minimum. In [20], the authors have proposed efficient algorithms for finding the group nearest neighbors with the minimum total distance in the Euclidean space. In [21], the authors have developed GNN algorithms for minimizing the minimum and the maximum distance in addition to the total distance of group members. In [22], the authors have proposed an approach for processing GNN queries in road networks. In [23], the authors have proposed an efficient bound using vector space property and using that bound they have developed an indexed and a non-index aggregate nearest neighbors (ANN) algorithms.

3.3 Traveling Salesman Problem (TSP) and Variants

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) [10] and multiple traveling salesman problem (MTSP) [9] 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.

In [24], the authors presented a local-global approach for GTSP. In [12], the authors present an improved genetic algorithm to provide an alternative and effective solution to the problem. The initial population was generated by a greedy strategy, and this enabled selected sub-route to be included in the initial population. The authors showed that the convergent speed is increased and at the same time complexity is significantly reduced in their approach.

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). In [25], the authors provided an 3/2- approximation algorithm, which runs in polynomial time when the number of depots is constant.

3.4 Elliptical Search Space Refinement Techniques

Elliptical properties have been used in the literature to refine the search region for queries like group nearest neighbor queries [26], trip planning queries [4], group trip planning queries [3] and privacy preserving trip planning queries [27]. 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 [26], the foci are set at the locations of two group members who are at the at the maximum distance from each other, and the length of the major axis is equal to the smallest aggregate distance computed based on retrieved POIs from the database. Any POI outside the ellipse cannot further minimize the aggregate distance for the group members. In [3], the foci are set at the centroids of source and destination locations of the group members, and the length of the major axis is equal to the smallest average aggregate trip distance computed based on retrieved POIs from the database. Any POI outside the ellipse cannot further minimize the aggregate trip distance for the group members. In [27], the foci of the ellipse are set at the source and destination locations of the user, and the length of the major axis is equal to the smallest trip distance computed based on retrieved POIs from the database. Any POI outside the ellipse cannot further minimize the trip distance for the user. In this thesis, we develop two novel techniques to refine the search region using multiple ellipses for GTS queries.

In this thesis, we present a new type of queries, group trip scheduling (GTS) query for a group in spatial databases and provide the first efficient solution for it. The query returns a set of optimal trips for the group members which ensure that each resultant trip starts from and ends at corresponding member's source and destination point, respectively and jointly all trips visit each specified POI types exactly once with the minimum aggregate trip overhead distance.

Chapter 4

Our Solution

In this chapter, we present our approach to process GTS queries and its variant in the Euclidean space and road networks. In a GTS query and a variant of GTS queries, 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 [28] in the database. The LSP incrementally retrieves POIs from the database until it identifies the trips that minimize the aggregate trip overhead 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.

The chapter is organized as follows. In Section 4.1, we discuss the preliminaries that we use to develop our approach. In Section 4.2, we show an overview of our developed approach for processing GTS queries. Every steps of our proposed approach has been discussed elaborately in Section 4.3.

4.1 Preliminaries

We use the concept of known region and search region [3, 4] for the retrieval of POIs from the database and to keep track of the POI search region, which has been explored and which is required to be explored.

4.1.1 Known Region

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. We incrementally retrieve the nearest neighbors with respect to a query point. In Figure 4.1, suppose in a state of query processing the LSP retrieves the first nearest POIs p_2^1 and the second nearest POI p_1^1 with respect to the geometric centroid G of source and destination locations of a group of three members. Here 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. Initially the known region is empty as no POI has been retrieved from the data storage. As POIs are retrieved by best-first search (BFS) then our known region will gradually expand with respect to G.

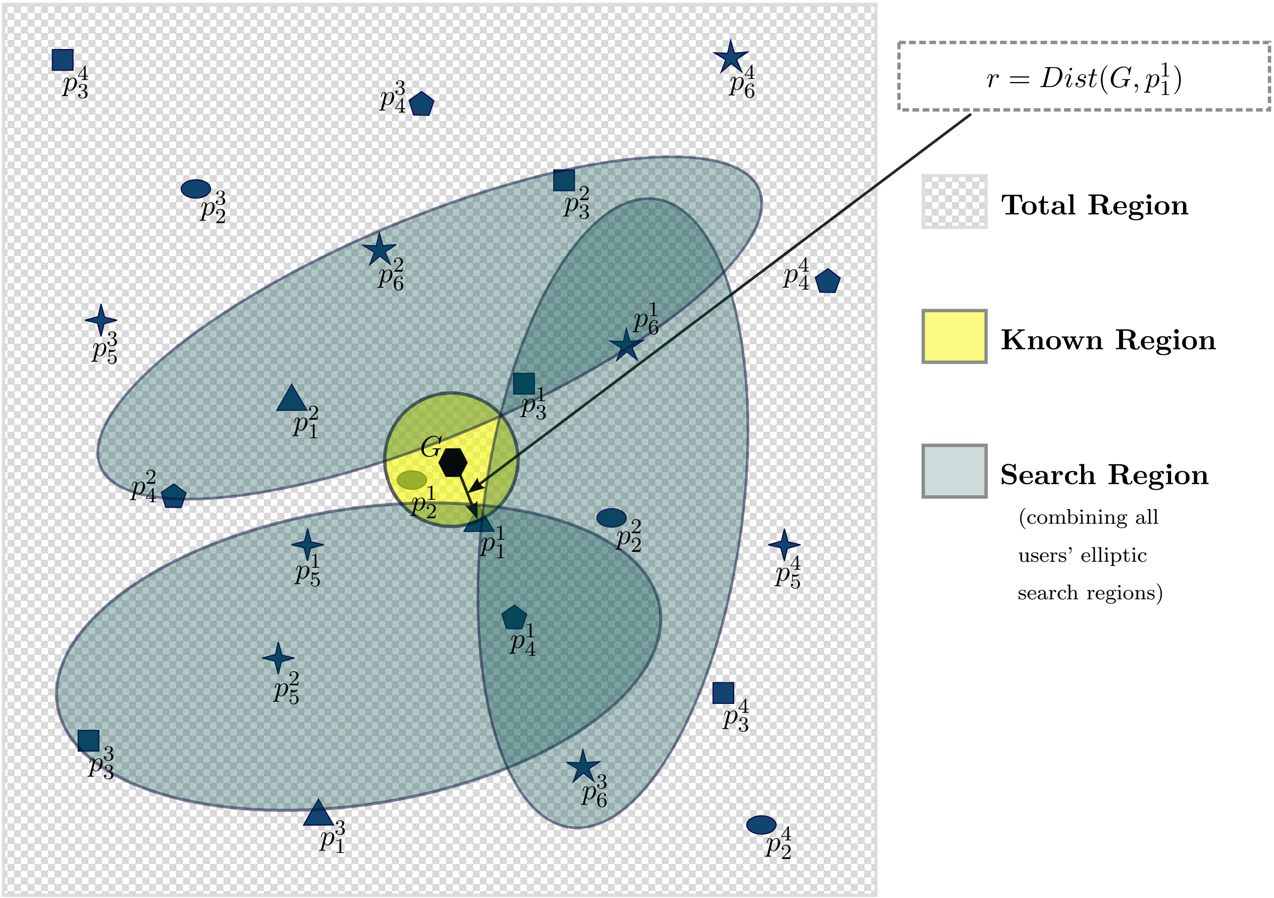


Figure 4.1: Known region and search region

4.1.2 Search Region

The search region represents the refined space that we need to explore for the optimal solution. 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 4.1, based on current retrieved POIs, p_2^1 and p_1^1, the search region is the union of three ellipses.

4.2 Overview of Our Approach

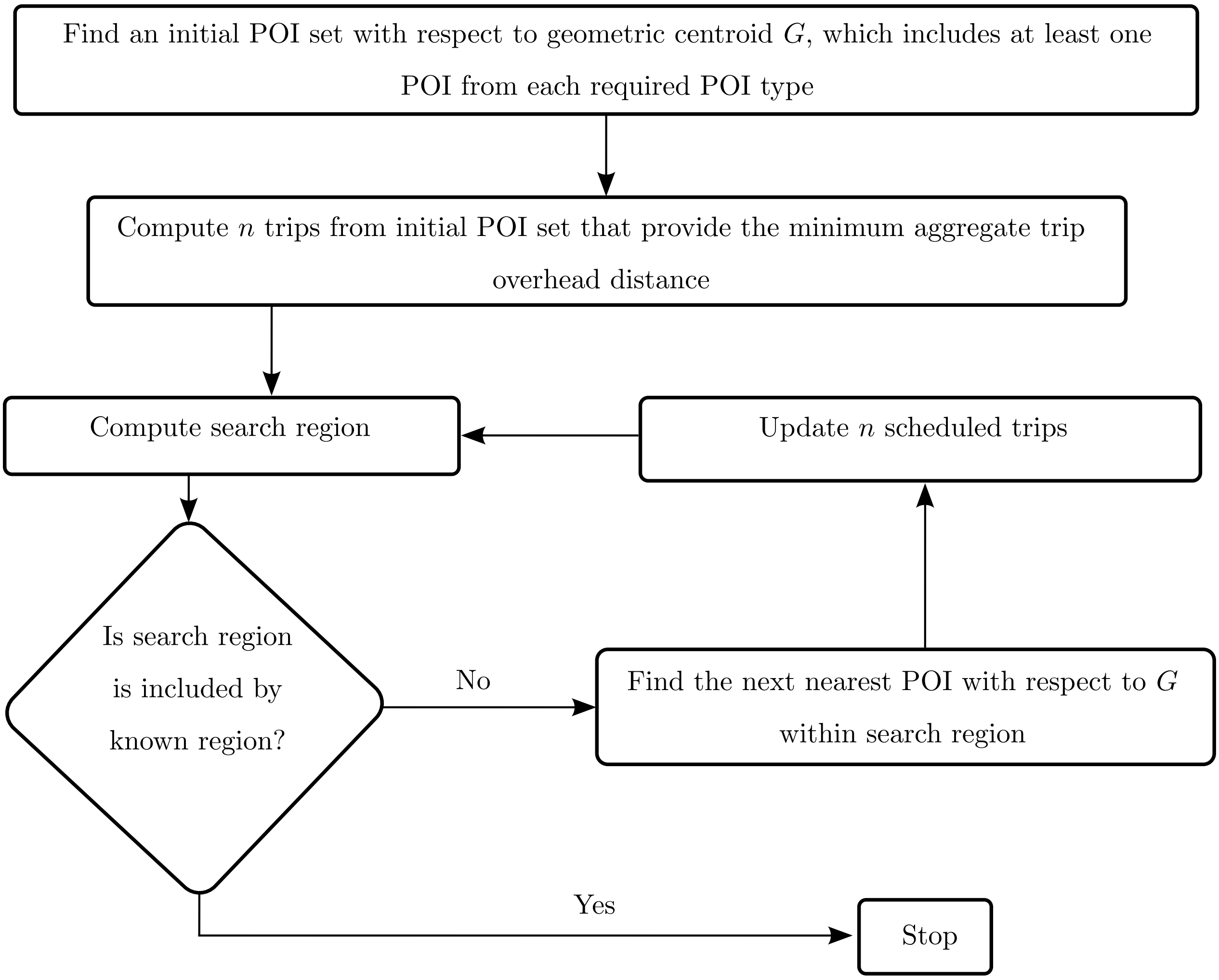


Figure 4.2: Overview of our approach for GTS queries

Figure 4.2 shows an overview of our developed approach for processing GTS queries. In a GTS query, to compute the upper bound of the aggregate trip overhead distance, our approach uses a heuristic to find at least one POI from each required POI types. Our developed approach initially incrementally retrieves the nearest POIs with respect to the geometric centroid of all group members' sources and destination locations, G. The retrieval of POIs continues until it found at least one POI from each required POI type. Using the initial retrieved POI set, our approach schedules n trips that provide the minimum aggregate trip overhead distance for the group members. For scheduling n trips, our approach uses an efficient dynamic programming technique. After that using our developed search region refinement techniques, our approach refines the search region to prune POIs that cannot be the part of the query answer. Then our proposed approach checks whether the known region includes the search region where the known region is computed based on the already retrieved POIs. 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.

4.3 Steps of GTS Query Process

In this section, we will present all the steps of our proposed approach for processing GTS queries. For efficient query processing, our target is to find minimum bound for search region from the whole universal region. We will retrieve all candidate POIs of required POI types incrementally and will approach to find optimal solution for GTS queries. Our proposed GTS query processing approach has following steps:

- Computing the known region
- Refinement of the search region
- Terminating condition for POI retrieval
- Dynamic programming technique for scheduling trips

In the following sections, we elaborate the steps of our approach for processing GTS queries and variant in spatial databases.

4.3.1 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 a group of n members. For the group of members rather than using multiple points (e.g., source points of each members) as query points to retrieve POIs from the database, we are interested of using single query point. It ensures that same POIs will not retrieve through queries accessing same nodes. We use the geometric centroid of the locations of n users' sources and destinations as the single query point to retrieve POIs from the database. 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 [28] in the database. The BFS search also

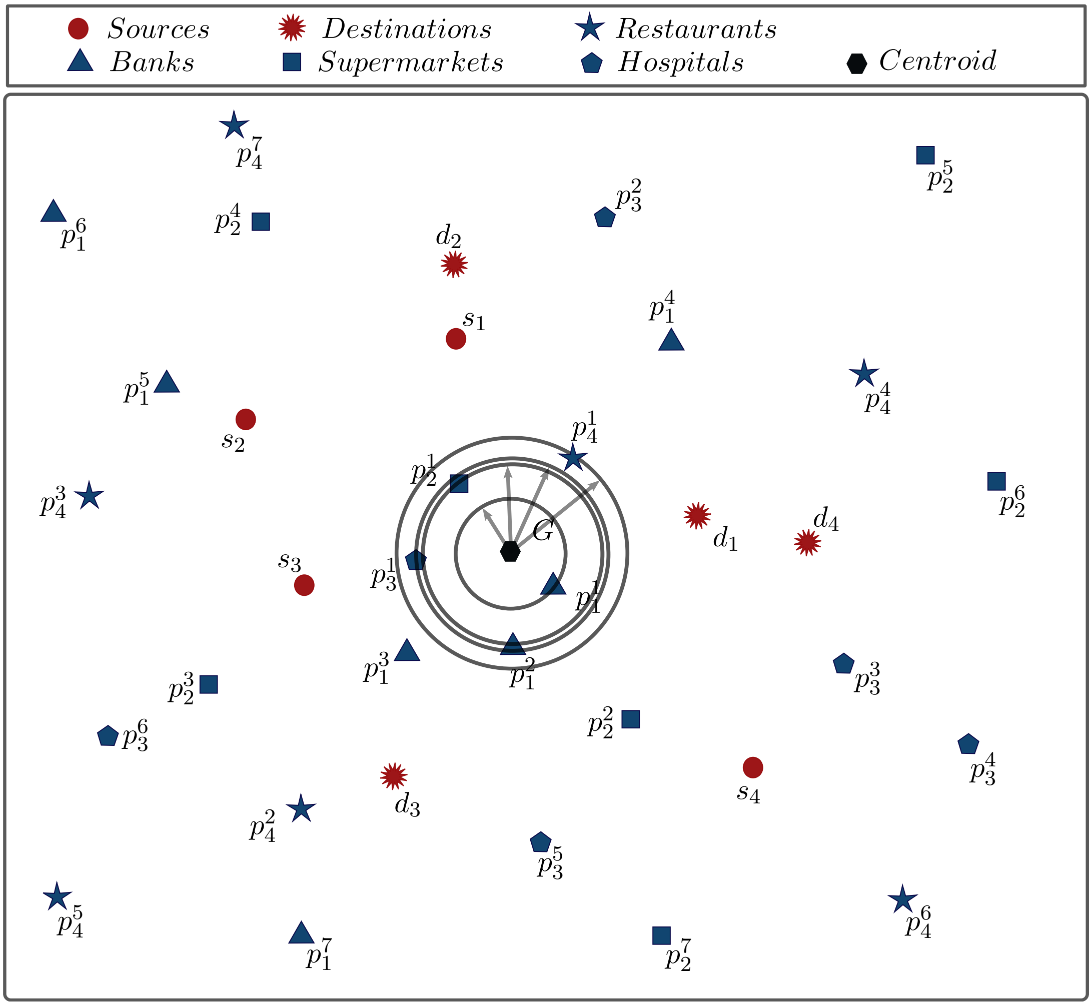


Figure 4.3: Computing the known region (known region expanding with the incremental POI retrieval)

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. The current known region have only one POI of any required POI types retrieved by the BFS search. 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.

In Figure 4.3, G be the centroid of four source-destination pairs, $< s_1, d_1 >$, $< s_2, d_2 >$, $< s_3, d_3 >$, and $< s_4, d_4 >$ of a group of four members for the example scenario that we have described before in Figure 1.2(a). With respect to G, the BFS search retrieves the nearest POI p_1^1 of POI type $c_1(Bank)$. The circular region centered at G with radius, $r =$ Euclidean distance, $Dist(G, p_1^1)$, is the current known region which have only one POI of a required POI type $c_1(Bank)$. The BFS search retrieves the next nearest POI p_2^1 of POI type $c_2(Supermarkets)$ and with the retrieval of this POI, r is updated with the Euclidean distance from G to the farthest POI among already retrieved POIs, p_1^1 and p_2^1. Thus our know region expands incrementally.

4.3.2 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 to refine search region using multiple ellipses for different aggregate functions (SUM and MAX) and for having different user defined constraints. Using Theorem 4.3.1, we present the first search region refinement technique for both aggregate functions SUM and MAX and using Theorems 4.3.2 and 4.3.3, we present the second search region refinement technique for aggregation functions SUM and MAX, respectively. For each individual user's elliptic search region, we choose the one which gives smaller bound between two ellipses computed by two different novel refinement techniques. Finally our refined search region consists of union of the smaller multiple ellipses where each ellipse corresponds to each group member. Based on these refinement techniques, we develop our algorithm to process GTS queries in Chapter 5. Note that existing elliptical property based pruning techniques [2–6] for spatial queries are not directly applicable for GTS queries.

Our proposed two different techniques are :

- First refinement technique:
  - Uses each group member's maximum trip distance (e.g. the trip distance of any trip covering m required POI types).
  - Can be used for both aggregate (SUM/MAX) functions.
- Second refinement technique:
  - Uses each group member's minimum trip distance (e.g. the trip distance of any trip that visits no POI types) and the aggregate trip overhead distance of all members.
  - For aggregate function SUM, it uses the total trip overhead distance of the group where for aggregate function MAX, it uses maximum trip overhead distance of the group.

The notations that we use in our theorems are summarized below:

- 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.
- $AggTripOvDist$: the current minimum aggregate trip overhead distance of the group, for aggregate function SUM, it will be $\sum_{i=1}^{n}(TripDist_i - T_{min_i})$ and for aggregate function MAX, it will be $\max_{i=1}^{n}(TripDist_i - T_{min_i})$.

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 4.3.1, 4.3.2 and 4.3.1 shows some ways to refine the search region for a GTS query in the Euclidean space and road networks.

4.3.2.1 First Refinement Technique for Aggregate Functions(sum and max)

Theorem 4.3.1 *The search region can be refined as* $E_1 \cup E_2 \cup \ldots \cup E_n$*, 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}*.*

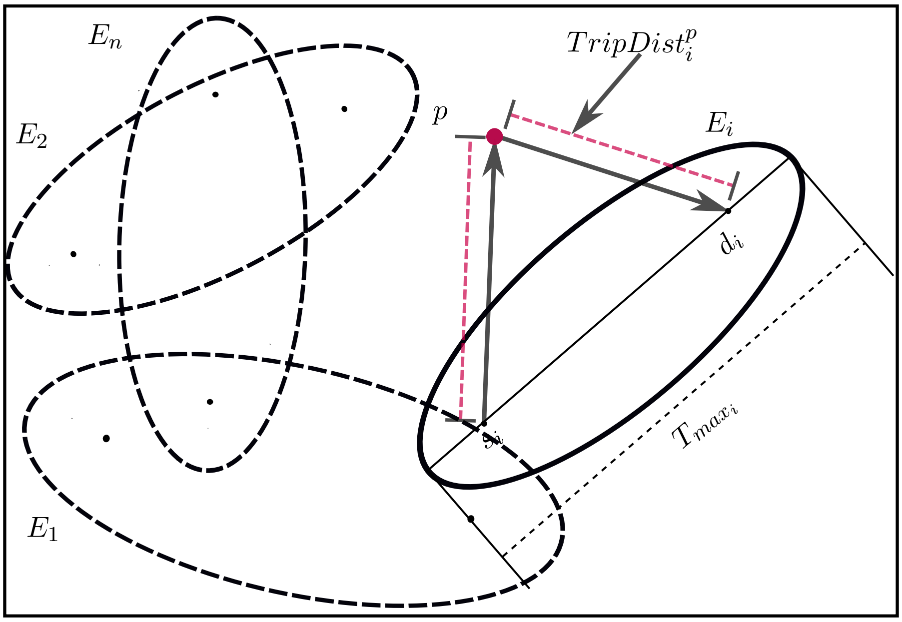


Figure 4.4: Proof of Theorem 4.3.1

Proof

Let a POI p lie outside the search region, $E_1 \cup E_2 \cup \ldots \cup E_n$, and $AggTripOvDist^p$ be the aggregate trip overhead distance of the group, where a group member u_i's trip includes POI p as shown in Figure 4.4. We have to prove that POI p can not be a part of the optimal solution, i.e., $AggTripOvDist^p > AggTripOvDist$.

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,

$$TripDist_i^p > T_{max_i} \tag{4.1}$$

which follows that,

$$(TripDist_i^p - T_{min_i}) > (T_{max_i} - T_{min_i}) \tag{4.2}$$

$(T_{max_i} - T_{min_i})$ represents the trip overhead 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 overhead distance for user u_i. Thus, any POI outside the union of ellipses $E_1, E_2, \ldots, E_n$ can not improve the aggregate trip overhead distance *AggTripOvDist* for the group and can not be a part of an optimally scheduled group of trips. Thus, $AggTripOvDist^p > AggTripOvDist$.

□

4.3.2.2 Second Refinement Technique for Aggregate Function sum

Theorem 4.3.2 *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 $AggTripOvDist + T_{min_i}$.*

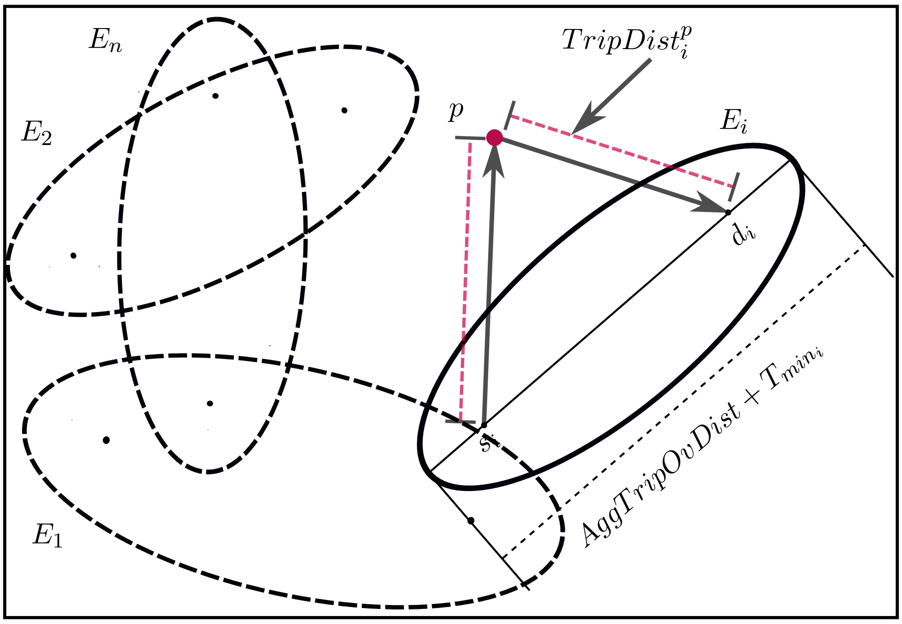


Figure 4.5: Proof of Theorem 4.3.2

Proof

Let a POI p lie outside the search region, $E_1 \cup E_2 \cup \ldots \cup E_n$, and $AggTripOvDist^p$ be the total trip overhead distance of the group, where a group member u_i's trip includes POI p as shown in Figure 4.5. We have to prove that POI p can not be a part of the optimal solution, i.e., $AggTripOvDist^p > AggTripOvDist$.

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,

$$TripDist_i^p > AggTripOvDist + T_{min_i}$$

Rearranging the equation we get,

$$TripDist_i^p - T_{min_i} > AggTripOvDist \tag{4.3}$$

For aggregate function SUM, by definition we know,

$$AggTripOvDist^p = (TripDist_i^p - T_{min_i}) + \sum_{l=1, l\neq i}^{n} (TripDist_l^p - T_{min_l}) \tag{4.4}$$

and

$$\sum_{l=1, l\neq i}^{n} TripDist_l^p \geq \sum_{l=1, l\neq i}^{n} T_{min_l} \tag{4.5}$$

From Equations 4.4 and 4.5, we get,

$$AggTripOvDist^p \geq (TripDist_i^p - T_{min_i}) \tag{4.6}$$

Combining inequalities of 4.3 and 4.6,

$$AggTripOvDist^p > AggTripOvDist$$

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.

□

For aggregate function SUM, our approach refines the ellipses of every group member independently using both bounds proposed in Theorems 4.3.1 and 4.3.2, 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 4.3.1 provides the minimum length of the major axis and for another member's ellipse, Theorem 4.3.2 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.

4.3.2.3 Second Refinement Technique for Aggregate Function max

Theorem 4.3.3 *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* $AggTripOvDist + T_{min_i}$.

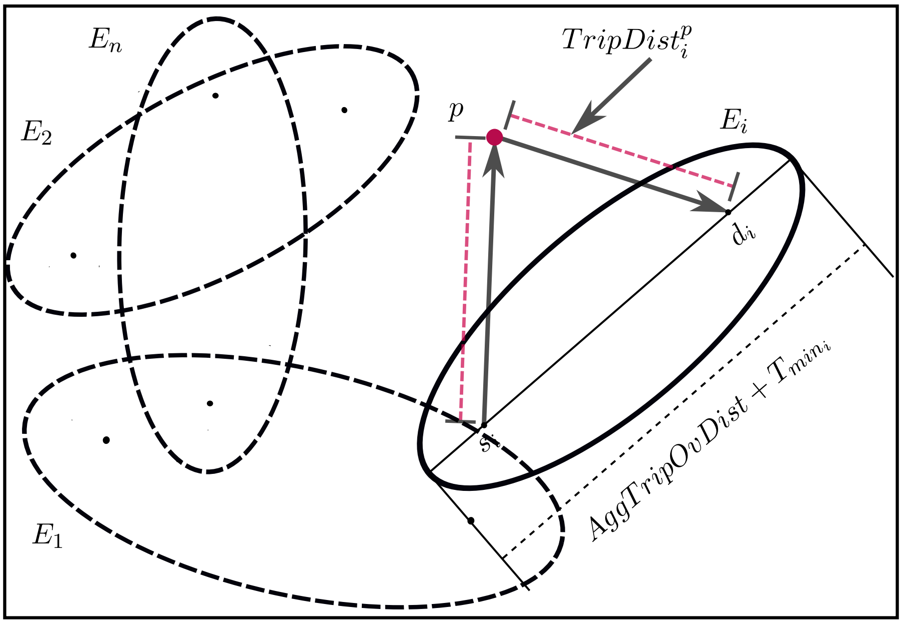


Figure 4.6: Proof of Theorem 4.3.3

Proof

Let a POI p lie outside the search region, $E_1 \cup E_2 \cup \ldots \cup E_n$, and $AggTripOvDist^p$ be the maximum trip overhead distance of the group, where a group member u_i's trip includes POI p as shown in Figure 4.6. We have to prove that POI p can not be a part of the optimal solution, i.e., $AggTripOvDist^p > AggTripOvDist$.

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,

$$TripDist_i^p > AggTripOvDist + T_{min_i}$$

Rearranging the equation we get,

$$TripDist_i^p - T_{min_i} > AggTripOvDist \tag{4.7}$$

For aggregate function MAX, by definition we know,

$$AggTripOvDist^p = \max(\max_{l=1, l\neq i}^{n}(TripDist_l^p - T_{min_l}), (TripDist_i^p - T_{min_i})) \tag{4.8}$$

and

$$AggTripOvDist^p \geq (TripDist_i^p - T_{min_i}) \tag{4.9}$$

Combining inequalities of 4.7 and 4.9, we get,

$$AggTripOvDist^p > AggTripOvDist \tag{4.10}$$

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.

$\square$

Similar to aggregate function SUM, for aggregate function MAX, our approach also uses Theorem 4.3.1 to refine search region. Using both bounds proposed in Theorems 4.3.1 and 4.3.3, our approach refines the ellipses of every group member independently 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 4.3.1 provides the minimum length of the major axis and for another member's ellipse, Theorem 4.3.3 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.

4.3.2.4 Extensions for Uniform GTS (UGTS) Queries

Refinement techniques that has been described in Section 4.3.2.1, 4.3.2.2 and 4.3.2.3 for aggregate functions SUM and MAX applicable for uniform GTS queries where each group member visits equal number of POI types for both aggregate functions SUM and MAX with slight modifications. It is possible to refine the search region more optimally for having the uniform POI type constraint. For achieving that, we update the definition of T_{min_i} and T_{max_i} according to the constraints for the Theorem 4.3.1, Theorem 4.3.2 and Theorem 4.3.3 for UGTS queries.

In a UGTS query, it should not happen that a group member visits no POI types or visits all required POI types. Here every group member visits uniform or equal number of POI types and each required POI type is included in a single trip. So we consider a subset of uniform number of POI types instead of considering all required POI types for T_{max_i} and no POI type for T_{min_i}. It is possible that we may find better bound for each user's elliptic search region which will help us to refine search region more efficiently.

Suppose in a UGTS query, a group of n members want to visit m required POI types combinedly where each group member should visit equal or fixed ($e = m/n$) number of POI types. For having the constraint, T_{min_i} represents the minimum trip distance of any trip covering any subset of e POI types from all required m POI types for a group member u_i instead of the distance between s_i and d_i without visiting any POI type. Similarly, T_{max_i} represents the maximum trip distance of any trip covering any subset of e POI types from all required m POI types for a group member u_i instead of the trip distance from s_i to d_i via required m POI types.

4.3.2.5 Extensions for GTS and UGTS Queries with Constraints

In a GTS or a UGTS query with "Dependencies among POIs" constraint, a group may need to visit a subset POI types in a user defined fixed order. Because of having defined fixed POI type order, some combinations or subset of POI types should be invalid for all group members. For computing the values of T_{min_i} and T_{max_i} for a group member u_i, we should discard those invalid combinations as well.

Similarly, for having constraint "Dependencies among users and POIs" in a GTS or UGTS query, a group member may need to visit any fixed POI type among the required POI types that the group should visit combinedly. Some combinations or subset of POI types and group members should be invalid because of having dependencies among users and POIs. For computing the values of T_{min_i} and T_{max_i} for a group member u_i, we must discard those invalid user and POI types combinations.

4.3.2.6 Example Scenario of the Search Region Refinement

In 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 aggregate trip overhead distance for the group using the dynamic programming technique shown in Section 4.3.4, and refines the search region using Theorems 4.3.1, 4.3.2 and 4.3.3 for aggregate functions SUM and MAX. 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 aggregate trip overhead distance of the group $AggTripOvDist$, which can further refine the search region.

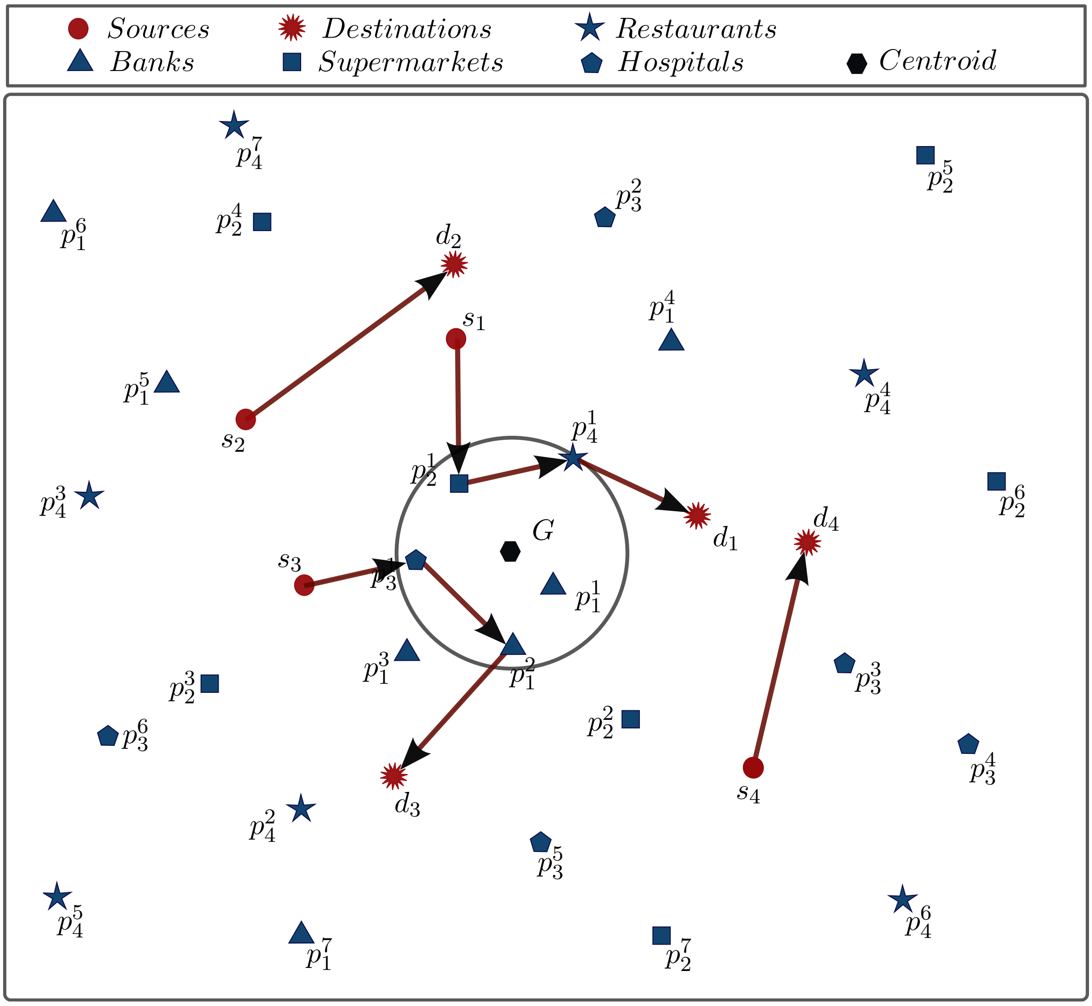


Figure 4.7: Initial known region (the circle with center G) and scheduled trips calculated using initial POIs

For our current example scenario that we have already described in Figure 1.2(a), Figure 4.7 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 with minimum total trip overhead distance of the group. Each scheduled trip starts from and ends at corresponding user's source and destination location, respectively, and each required POI type is included in a single trip. Note that the initial set may include more than one POI 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.

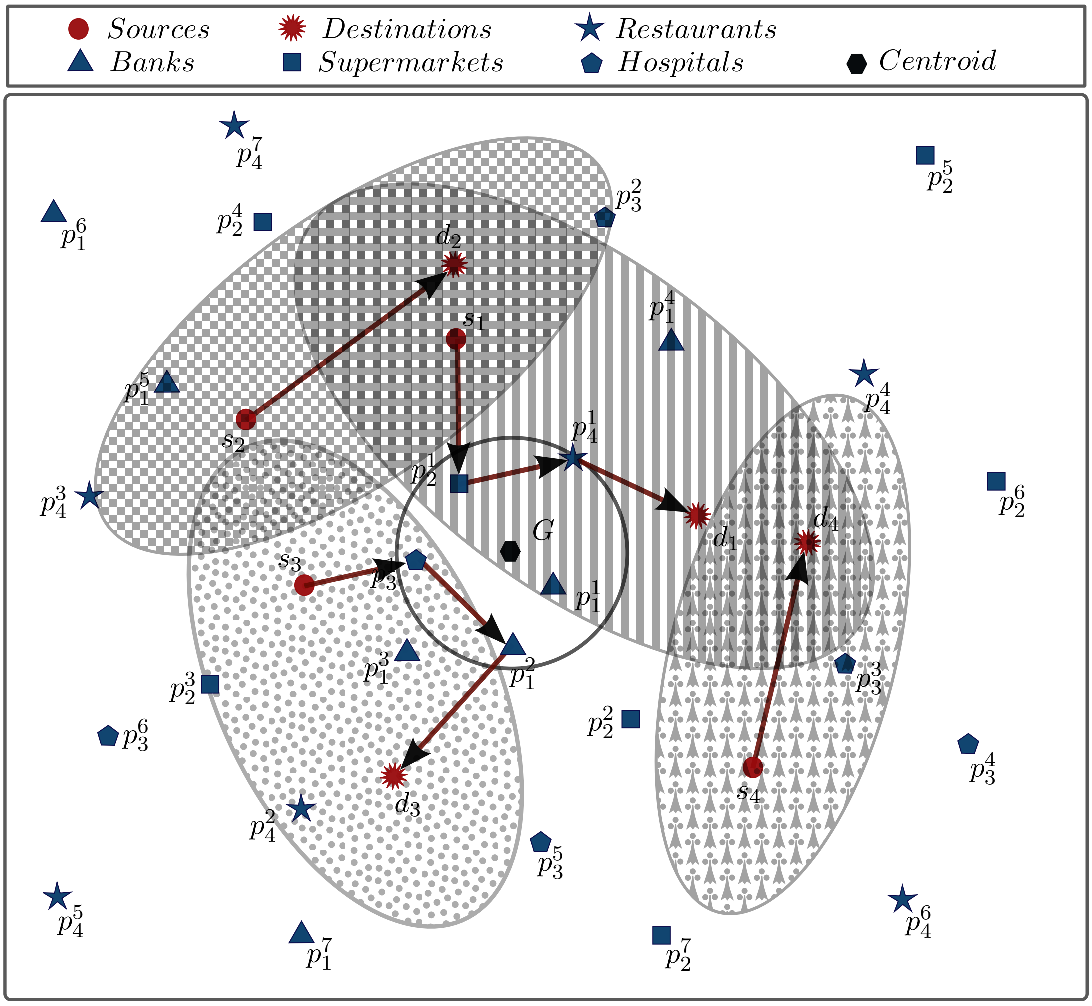


Figure 4.8: Refined search region

Using bounds from Theorem 4.3.1 and 4.3.2 for aggregate function SUM, we compute and refine the search region. Figure 4.8 shows the refined search region as the union of four ellipses.

For aggregate function MAX, our approach refines the search region using Theorems 4.3.1 and 4.3.3. For uniform GTS (UGTS) queries and GTS or UGTS queries having different types of constraints, the search region will use the refinement techniques that has been described in Section 4.3.2.4 and in Section 4.3.2.5, respectively.

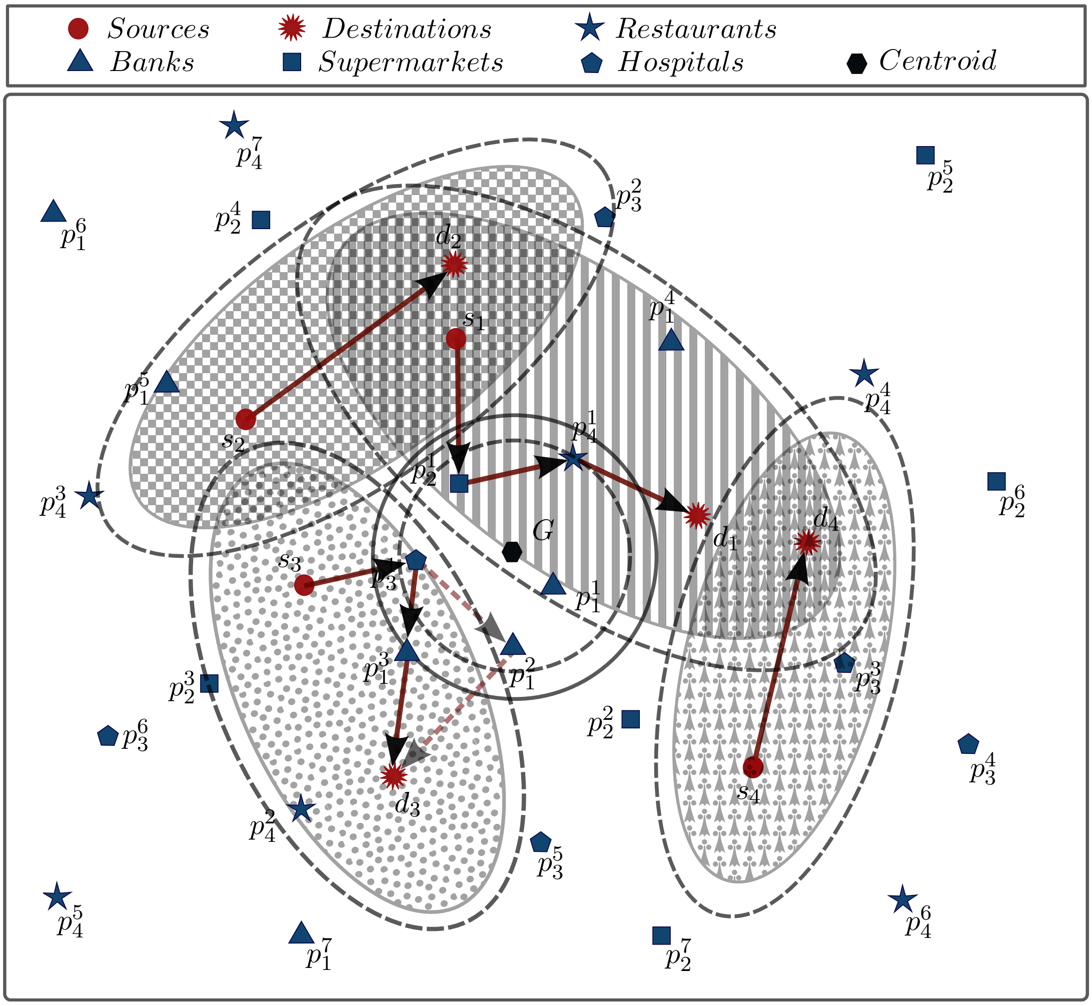


Figure 4.9: Known region expands (outer circle) and search region shrinks (inner ellipses)

In our example scenario, in Figure 4.9, after retrieving the next nearest POI p_1^3 within search region, 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 $s_3 \rightarrow p_3^1 \rightarrow p_1^3 \rightarrow d_3$. It also improves the aggregate trip overhead distance and shrinks the search region for all group members. In Figure 4.9, 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.

4.3.3 Terminating Condition for POI Retrieval

When the known region covers the search region, no more minimization in the aggregate trip overhead distance is further possible. At this point, we can terminate traversing R^*-tree and retrieving POIs. Figure 4.10 shows that the known region covers the search region which is the union of all users' elliptic search regions. This is the termination condition of our algorithm and our algorithm terminates here.

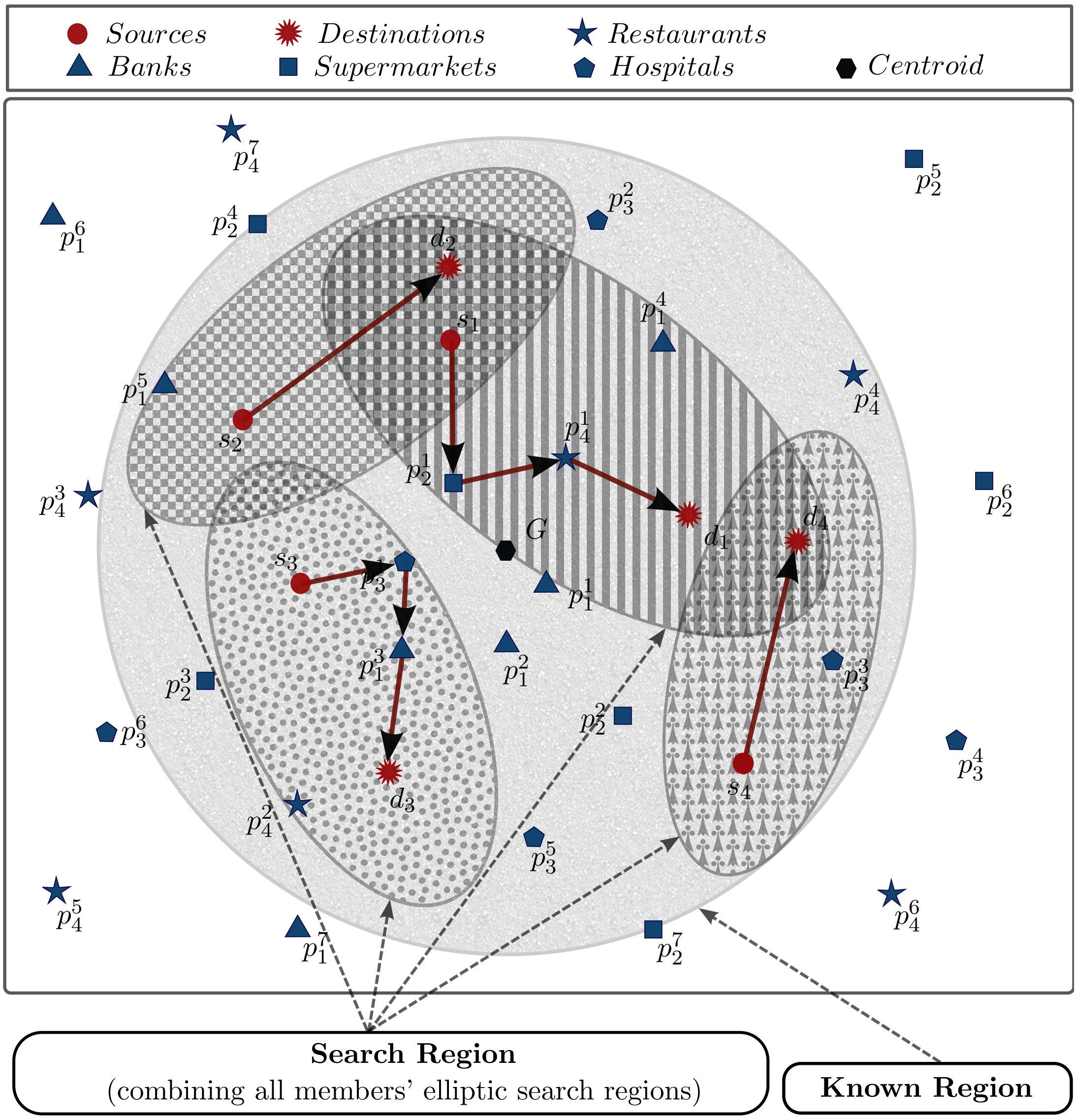


Figure 4.10: Terminating condition: the known region includes the search region

4.3.4 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 aggregate trip overhead distance of the group is minimized. Each time our approach retrieves new POIs, it again schedules the trips using new POIs, if the new trips improve the aggregate trip overhead 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 aggregate trip overhead 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.

In Sections 4.3.4.1 and 4.3.4.2, we elaborately discuss our proposed dynamic programming approach for GTS and UGTS queries, respectively, for both aggregate functions SUM and MAX. We extend our dynamic programming approach to schedule trips for GTS or UGTS queries with "dependencies among POIs" and "dependencies among users and POIs" constraints in Sections 4.3.4.3 and 4.3.4.4, respectively.

4.3.4.1 Trip Scheduling for GTS Queries

For aggregate function SUM, our dynamic programming technique minimizes the following objective function:

$$\sum_{i=1}^{n}(TripDist_i - T_{min_i})$$

On other hand, for aggregate function MAX, the objective function that our dynamic programming technique minimizes is as follows:

$$\max_{i=1}^{n}(TripDist_i - T_{min_i})$$

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,

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

Constraints $\sum_{i=1}^{n} |\mathbb{C}_{T_i}| = m$ and $\bigcup_{i=1}^{n} \mathbb{C}_{T_i} = \mathbb{C}$ ensure that each required POI type should be visited by any group member. Another constraint $\forall_{i,j}(\mathbb{C}_{T_i} \cap \mathbb{C}_{T_j}) = \emptyset$ ensures each POI type should be included in a single trip exactly once.

For the GTS query, we have a set of m POI types $\mathbb{C}$=$\{c_1, c_2, \ldots, c_m\}$, where a group member visits any number of POI types from 0 to m. Thus, there are $\sum_{y=0}^{m}({}^{m}C_y)$ 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 ${}^{|\mathbb{C}|}C_y = {}^{4}C_2 = 6$ 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 $({}^{\mathbb{C}}C_y)^1 = \{c_1, c_2\}$, $({}^{\mathbb{C}}C_y)^2 = \{c_1, c_3\}, \ldots, ({}^{\mathbb{C}}C_y)^6 = \{c_3, c_4\}$.

For each member of the set ${}^{\mathbb{C}}C_y$, we calculate optimal trips for each group member in $U = \{u_1, u_2, u_3, \ldots, u_n\}$ and store trip overhead distances for future computations. This is the initial step for our dynamic programming technique. We define $m+1$ dynamic tables, $\nu_0, \nu_1, \nu_2, \ldots \nu_m$ to store the trip overhead distances of every single member of the group and the aggregate trip overhead distances of the combined group members. Table ν_y has ${}^{m}C_y$ 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 $U = \{u_1, u_2, u_3, \ldots, u_n\}$. Except the dynamic table ν_0, the cells of these columns of all other dynamic tables $\nu_1, \nu_2, \ldots \nu_m$ store the minimum trip overhead distances for the corresponding column's member to visit the POI types of the corresponding rows of that table. The single member columns of the dynamic table ν_0 store the trip distance which is actually the distance between s_i and d_i via no POI types, instead of storing the trip overhead distances of the group members. The motivation of this exceptional case is, whenever we need trip distance (e.g., to compute T_{max_i} value which represents the maximum trip distance of a group member u_i) instead of trip overhead distance of any trip, we can easily use the stored value of table ν_0 and compute the actual trip distance from the stored trip overhead distance.

Each dynamic table except ν_m has $(n-2)$ combined member columns $u_1u_2, u_1u_2u_3, \ldots, u_1u_2\ldots u_{n-1}$, where the cells of the corresponding columns store the minimum aggregate trip overhead distances of the corresponding column's multiple members passing through the POI types of the corresponding rows of that table. For example, each cell of the column u_1u_2 stores the minimum aggregate trip overhead distance of users 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 4.1 shows the structure of ν_y where $0 \leq y \leq (m-1)$. Table 4.2 shows the structure of ν_m that has an extra column $u_1u_2\ldots u_n$ to store the minimum aggregate trip overhead 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.

Table 4.1: Structure of dynamic table ν_y, where $0 \leq y \leq (m-1)$

	$\{u_1\}$	$\{u_2\}$	$\ldots$	$\{u_n\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$	$\ldots$	$\{u_1u_2\ldots u_{n-1}\}$
$\{c_1, c_2, \ldots, c_y\}$								
$\{c_1, c_3, \ldots, c_y\}$								
$\vdots$								

Table 4.2: Structure of dynamic table ν_m

	$\{u_1\}$	$\{u_2\}$	$\ldots$	$\{u_n\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$	$\ldots$	$\{u_1u_2\ldots u_n\}$
$\{c_1, c_2, \ldots, c_m\}$								

In addition to storing the minimum trip overhead distances, each cell of the dynamic tables $\nu_1, \nu_2, \ldots \nu_m$ stores the set of POIs for which the minimum trip overhead distance for single member columns or the minimum aggregate trip overhead distance for combined member columns is obtained. For example, cell $\nu_3[\{c_1, c_3, c_4\}][\{u_1\}]$ stores the minimum trip overhead distance and the POI set

$< p_3, p_1, p_4 >$, for which u_1 obtains the minimum trip overhead distance to visit POI types $\{c_1, c_3, c_4\}$.

The size of a dynamic table ν_y is : ${}^mC_y \times (n + (n-2))$, where $0 \le y \le (m-1)$, 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 only the single member columns (e.g., to compute combined member column u_1u_2) or both single and combined member columns (e.g., to compute combined member columns $u_1u_2u_3$ to $u_1u_2 \ldots u_n$) of dynamic tables $\nu_0, \nu_1, \ldots, \nu_y$. For example, for computing each cell of combined member column u_1u_2 of table ν_3, we use the already calculated single member columns of dynamic tables 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 4.3.

Table 4.3: Possible number of POI type distributions between u_1 and u_2

u_1	u_2
3	0
2	1
1	2
0	3

Formally, for aggregate function SUM, the minimum total trip overhead 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 $({}^{\mathbb{C}}C_g)^j \cap ({}^{\mathbb{C}}C_{(y-g)})^k = \emptyset$.

For aggregate function MAX, the minimum value of maximum trip overhead 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
$\min_{g=0}^{y}\{\min_{j=1}^{{}^{m}C_g}\{\min_{k=1}^{{}^{m}C_{y-g}}\{\max(\nu_g[({}^{\mathbb{C}}C_g)^j][\{u_1\}], \nu_{(y-g)}[({}^{\mathbb{C}}C_{(y-g)})^k][\{u_2\}])\}\}\}$,
where $({}^{\mathbb{C}}C_g)^j \cap ({}^{\mathbb{C}}C_{(y-g)})^k = \emptyset$.
The constraints guarantee that no POI type is considered twice while computing the minimum aggregate trip overhead 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 distributions listed in Table 4.3 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 GTS query result.

***We elaborate our dynamic programming technique with an example for aggregate function* sum.** In our current example scenario, 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 overhead 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 4.7 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 (single member columns of dynamic table ν_0) and trip overhead distances (combined member columns of dynamic table ν_0 and both single and combined member columns of dynamic tables ν_1, ν_2, ν_3 and ν_4) 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 overhead 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 4.4 (a-e) show ν_0, ν_1, ν_2, ν_3 and ν_4 for the considered example.

Table 4.4: Dynamic tables for an example scenario for aggregate function SUM

(a) Dynamic table ν_0

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\emptyset$	28.75	25.00	47.55	77.48	0.0	0.0

(b) Dynamic table ν_1

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1\}$	105.36	76.55	57.61	7.32	76.55	57.61
$\{c_2\}$	49.05	23.99	7.03	4.72	23.99	7.03
$\{c_3\}$	105.32	67.80	41.85	5.42	67.80	41.85
$\{c_4\}$	102.51	70.29	42.02	5.65	70.29	42.02

(c) Dynamic table ν_2

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1, c_2\}$	105.37	76.58	57.61	7.36	76.58	57.61
$\{c_1, c_3\}$	105.41	78.54	57.98	7.32	78.54	57.98
$\{c_1, c_4\}$	106.98	81.84	58.61	7.34	81.84	58.61
$\{c_2, c_3\}$	105.34	69.92	41.86	5.46	69.92	41.86
$\{c_2, c_4\}$	106.19	72.64	43.19	5.67	72.64	43.19
$\{c_3, c_4\}$	107.01	73.28	43.62	5.83	73.28	43.62

(d) Dynamic table ν_3

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1, c_2, c_3\}$	105.41	78.54	57.98	7.36	78.54	57.98
$\{c_1, c_2, c_4\}$	107.06	81.84	58.62	7.36	81.84	58.62
$\{c_1, c_3, c_4\}$	107.03	81.84	58.61	7.34	81.84	58.61
$\{c_2, c_3, c_4\}$	107.14	73.52	43.93	5.86	73.52	43.93

(e) Dynamic table ν_4

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$	$\{u_1u_2u_3u_4\}$
$\{c_1, c_2, c_3, c_4\}$	107.15	81.84	58.62	7.36	81.84	58.62	7.36

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 overhead distance for the corresponding column's user passing through POI types of the corresponding row of that table. For example, in Table 4.4(c), cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$ contains the minimum trip overhead distance for user u_1 passing through POI types c_1 and c_2. For computing this trip overhead 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 overhead distances are listed in Table 4.5.

Table 4.5: Candidate trips with trip overhead distances for cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$

Candidate trips	**Trip overhead distances**
$s_1 \rightarrow p_2^1 \rightarrow p_1^1 \rightarrow d_1$	105.37
$s_1 \rightarrow p_2^1 \rightarrow p_1^2 \rightarrow d_1$	109.89
$s_1 \rightarrow p_1^1 \rightarrow p_2^1 \rightarrow d_1$	106.62
$s_1 \rightarrow p_1^2 \rightarrow p_2^1 \rightarrow d_1$	126.58

Among the candidate trips listed in this table, the minimum trip overhead distance 105.37 for trip $s_1 \rightarrow p_2^1 \rightarrow p_1^1 \rightarrow d_1$ 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 instead of trip overhead distance for particular user's trip from her source to destination location and trip overhead distances for the combined members.

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 $\nu_0[\emptyset][\{u_1 u_2\}]$ contains the minimum total trip overhead distance of trips T_1 and T_2, where the trips correspond to users u_1 and u_2, respectively, and visit no POI types. Table 4.6 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 (single member columns) and trip overhead distances (combined member columns).

Table 4.6: Candidate combined combinations with trip overhead distances for cell $\nu_0[\emptyset][\{u_1u_2\}]$.

Combined combinations	**Distances**	**Trip overhead**
$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}])$ + $(\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$	$(28.75 - 28.75) + (25.00 - 25.00)$	0.00

Table 4.7: Candidate combined combinations with trip overhead distances for cell $\nu_1[\{c_1\}][\{u_1u_2\}]$.

Combined combinations	**Distances**	**Trip overhead**
$\nu_1[\{c_1\}][\{u_1\}] + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$	$105.36 + (25.00 - 25.00)$	105.36
$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + \nu_1[\{c_1\}][\{u_2\}]$	$(28.75 - 28.75) + 76.55$	76.55

To compute the values for the cells of the combined member columns for any 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 overhead 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 overhead distances and multiple member trip overhead distances in ν_0, ν_1 and ν_2. Using ν_0, ν_1 and ν_2 (Tables 4.4(a-c)), Table 4.8 shows the candidate combinations of POI types for u_1 and u_2 along with the trip overhead distances for computing the value for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$ in ν_2 (Table 4.4(c)). Among candidate combinations listed in Table 4.8, the minimum total trip overhead distance 76.58 is stored in cell $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$.

Table 4.8: Candidate combined combinations with trip overhead distances for cell $\nu_2[\{c_1, c_2\}][\{u_1 u_2\}]$.

Combined Combinations	**Distances**	**Trip overhead**
$\nu_2[\{c_1, c_2\}][\{u_1\}] + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$	$105.37 + (25.00 - 25.00)$	105.37
$\nu_1[\{c_1\}][\{u_1\}] + \nu_1[\{c_2\}][\{u_2\}]$	$105.36 + 23.99$	129.35
$\nu_1[\{c_2\}][\{u_1\}] + \nu_1[\{c_1\}][\{u_2\}]$	$49.05 + 76.55$	125.60
$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + \nu_2[\{c_1, c_2\}][\{u_2\}]$	$(28.75 - 28.75) + 76.58$	76.58

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 overhead distances for cell $\nu_1[\{c_1\}][\{u_1 u_2\}]$, $\nu_3[\{c_1, c_2, c_3\}][\{u_1 u_2\}]$ and $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1 u_2\}]$ are listed in Table 4.7, Table 4.9 and Table 4.10, respectively.

Table 4.9: Candidate combined combinations with trip overhead distances for cell $\nu_3[\{c_1, c_2, c_3\}][\{u_1 u_2\}]$

Combined combinations	**Distances**	**Trip overhead**
$\nu_3[\{c_1, c_2, c_3\}][\{u_1\}] + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$	$105.41 + (25.00 - 25.00)$	105.41
$\nu_2[\{c_1, c_2\}][\{u_1\}] + \nu_1[\{c_3\}][\{u_2\}]$	$105.37 + 67.80$	173.17
$\nu_2[\{c_1, c_3\}][\{u_1\}] + \nu_1[\{c_2\}][\{u_2\}]$	$105.41 + 23.99$	129.40
$\nu_2[\{c_2, c_3\}][\{u_1\}] + \nu_1[\{c_1\}][\{u_2\}]$	$105.34 + 76.55$	181.89
$\nu_1[\{c_1\}][\{u_1\}] + \nu_2[\{c_2, c_3\}][\{u_2\}]$	$105.36 + 69.92$	175.28
$\nu_1[\{c_2\}][\{u_1\}] + \nu_2[\{c_1, c_3\}][\{u_2\}]$	$49.05 + 78.54$	127.59
$\nu_1[\{c_3\}][\{u_1\}] + \nu_2[\{c_1, c_2\}][\{u_2\}]$	$105.32 + 76.58$	181.90
$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + \nu_3[\{c_1, c_2, c_3\}][\{u_2\}]$	$(28.75 - 28.75) + 78.54$	78.54

Table 4.10: Candidate combined combinations with trip overhead distances for cell $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1 u_2\}]$

Combined Combinations	**Distances**	**Trip overhead**
$\nu_4[\{c_1,c_2,c_3,c_4\}][\{u_1\}] + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$	$107.15 + (25.00 - 25.00)$	107.15
$\nu_3[\{c_1,c_2,c_3\}][\{u_1\}] + \nu_1[\{c_4\}][\{u_2\}]$	$105.41 + 70.29$	175.70
$\nu_3[\{c_1,c_2,c_4\}][\{u_1\}] + \nu_1[\{c_3\}][\{u_2\}]$	$107.06 + 67.80$	174.86
$\nu_3[\{c_1,c_3,c_4\}][\{u_1\}] + \nu_1[\{c_2\}][\{u_2\}]$	$107.03 + 23.99$	131.02
$\nu_3[\{c_2,c_3,c_4\}][\{u_1\}] + \nu_1[\{c_1\}][\{u_2\}]$	$107.14 + 76.55$	183.69
$\nu_2[\{c_1,c_2\}][\{u_1\}] + \nu_2[\{c_3,c_4\}][\{u_2\}]$	$105.37 + 73.28$	178.65
$\nu_2[\{c_1,c_3\}][\{u_1\}] + \nu_2[\{c_2,c_4\}][\{u_2\}]$	$105.41 + 72.64$	178.05
$\nu_2[\{c_1,c_4\}][\{u_1\}] + \nu_2[\{c_2,c_3\}][\{u_2\}]$	$106.98 + 69.92$	176.90
$\nu_2[\{c_2,c_3\}][\{u_1\}] + \nu_2[\{c_1,c_3\}][\{u_2\}]$	$105.34 + 81.84$	187.18
$\nu_2[\{c_2,c_4\}][\{u_1\}] + \nu_2[\{c_1,c_3\}][\{u_2\}]$	$106.19 + 78.54$	184.73
$\nu_2[\{c_3,c_4\}][\{u_1\}] + \nu_2[\{c_1,c_2\}][\{u_2\}]$	$107.01 + 76.58$	183.59
$\nu_1[\{c_1\}][\{u_1\}] + \nu_3[\{c_2,c_3,c_4\}][\{u_2\}]$	$105.36 + 73.52$	178.88
$\nu_1[\{c_2\}][\{u_1\}] + \nu_3[\{c_1,c_3,c_4\}][\{u_2\}]$	$49.05 + 81.84$	130.89
$\nu_1[\{c_3\}][\{u_1\}] + \nu_3[\{c_1,c_2,c_4\}][\{u_2\}]$	$105.32 + 81.84$	187.16
$\nu_1[\{c_4\}][\{u_1\}] + \nu_3[\{c_1,c_2,c_3\}][\{u_2\}]$	$102.51 + 78.54$	181.05
$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + \nu_4[\{c_1,c_2,c_3,c_4\}][\{u_2\}]$	$(28.75 - 28.75) + 81.84$	81.84

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 $\nu_2[\{c_1, c_2\}][\{u_1 u_2 u_3\}]$ contains the minimum total trip overhead 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 4.4(a-c)), Table 4.11 shows the candidate combinations of POI types for combined members $u_1 u_2$ and single member u_3 along with the trip overhead distances for computing the value for cell $\nu_2[\{c_1, c_2\}][\{u_1 u_2 u_3\}]$ in ν_2 (Table 4.4(c)).

Table 4.11: Candidate combined combinations with trip overhead distances for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2u_3\}]$.

Combined Combinations	**Distances**	**Trip overhead**
$\nu_2[\{c_1, c_2\}][\{u_1u_2\}] + (\nu_0[\emptyset][\{u_3\}] - \nu_0[\emptyset][\{u_3\}])$	$76.58 + (47.55 - 47.55)$	76.58
$\nu_1[\{c_1\}][\{u_1u_2\}] + \nu_1[\{c_2\}][\{u_3\}]$	$76.55 + 7.03$	83.58
$\nu_1[\{c_2\}][\{u_1u_2\}] + \nu_1[\{c_1\}][\{u_3\}]$	$23.99 + 57.61$	81.60
$\nu_0[\emptyset][\{u_1u_2\}] + \nu_2[\{c_1, c_2\}][\{u_3\}]$	$0.0 + 57.61$	57.61

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 overhead 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 overhead 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 overhead distance is 7.36 and 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 *AggTripOvDist* that we have mentioned in Section 4.3.2. To get the values of T_{min_i} for each user u_i, we simply take the minimum values from Table 4.4(a). On the other hand, to get the values of T_{max_i} which is the maximum trip distance for each user u_i for visiting all required POI types, we take the maximum values from Table 4.4(e) and then add the distance from s_i to d_i without visiting any POI types. T_{min_i} and T_{max_i} values for users $\{u_1, u_2, u_3, u_4\}$ are $\{28.75, 25.00, 47.55, 77.48\}$ and $\{(107.15 + 28.75), (81.84 + 25.00), (58.62 + 47.55), (7.36 + 77.48)\} \equiv \{135.90, 106.84, 106.17, 84.84\}$, respectively. Using these values we refine the search region based on Theorems 4.3.1 and 4.3.2. For user u_1, based on Theorem 4.3.1, the major axis for the elliptic region E_1 is 135.90. On the other hand, based on Theorem 4.3.2, the major axis is $7.36 + 28.75 = 36.11$. We take the best bound among them which is 36.11, 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 overhead distance is obtained. For the sake of clarity we do not show them in the tables.

***Now we elaborate our dynamic programming technique for aggregate function* max.** To elaborate the dynamic programming technique for aggregate function MAX, we consider similar example scenario that we have used for aggregate function SUM. For aggregate function MAX, 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 value of maximum trip overhead of the group members 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.

After initiating the GTS query for aggregate function MAX by the coordinator of the group of four members, the LSP retrieves initial set of POIs which includes at least a POI from every required POI type. Using the initial set of POIs, we first compute all possible trips for the group members and then schedule the trips using our proposed dynamic programming technique for aggregate function MAX.

Tables 4.12(a-e) show $(m + 1)$, i.e., 5 tables, ν_0, ν_1, ν_2, ν_3 and ν_4 for the considered example. The tables store the computed minimum trip overhead distances and combined minimum value of maximum trip overhead 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 value of maximum trip overhead of the 4 scheduled trips for 4 users that together visit 4 POI types, where each POI type is visited by a single user.

Computing single member columns: In the dynamic tables, columns u_1, u_2, u_3 and u_4 are the single member columns. Similar to GTS queries for aggregate function SUM, except table ν_0, each cell of these columns of a table from ν_1 to ν_4, stores the minimum trip overhead distance for the corresponding column's user passing through POI types of the corresponding row of that table. For example, in Table 4.12(c), cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$ contains the minimum trip overhead distance for user u_1 passing through POI types c_1 and c_2. For computing this trip overhead distance, we consider user u_1's source (s_1) and destination (d_1) locations along with all candidate POIs of POI types c_1 and c_2 in the initial set of POIs that has been retrieved by the LSP. Among the candidate trips including all

Table 4.12: Dynamic tables for an example scenario for aggregate function MAX

(a) Dynamic table ν_0

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\emptyset$	28.75	25.00	47.55	77.48	0.0	0.0

(b) Dynamic table ν_1

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1\}$	105.36	76.55	57.61	7.32	76.55	57.61
$\{c_2\}$	49.05	23.99	7.03	4.72	23.99	7.03
$\{c_3\}$	105.32	67.80	41.85	5.42	67.80	41.85
$\{c_4\}$	102.51	70.29	42.02	5.65	70.29	42.02

(c) Dynamic table ν_2

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1, c_2\}$	105.37	76.58	57.61	7.36	76.55	57.61
$\{c_1, c_3\}$	105.41	78.54	57.98	7.32	78.54	57.98
$\{c_1, c_4\}$	106.98	81.84	58.61	7.34	81.84	58.61
$\{c_2, c_3\}$	105.34	69.92	41.86	5.46	67.80	41.85
$\{c_2, c_4\}$	106.19	72.64	43.19	5.67	70.29	42.02
$\{c_3, c_4\}$	107.01	73.28	43.62	5.83	73.28	43.62

(d) Dynamic table ν_3

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1, c_2, c_3\}$	105.41	78.54	57.98	7.36	78.54	57.98
$\{c_1, c_2, c_4\}$	107.06	81.84	58.62	7.36	81.84	58.61
$\{c_1, c_3, c_4\}$	107.03	81.84	58.61	7.34	81.84	58.61
$\{c_2, c_3, c_4\}$	107.14	73.52	43.93	5.86	73.28	43.62

(e) Dynamic table ν_4

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$	$\{u_1u_2u_3u_4\}$
$\{c_1, c_2, c_3, c_4\}$	107.15	81.84	58.62	7.36	81.84	58.61	7.34

combinations of POIs of both POI types with any POI order, the minimum trip overhead distance 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 (single member columns) and trip overhead distances (combined member columns).

Computing combined member columns: For aggregate function MAX, the combined member columns store the minimum value of maximum trip overhead of the corresponding column's group members instead of storing the minimum total trip distance of the group members. This is the main difference between the dynamic programming approach for aggregate function SUM and MAX. Using the single member columns and already calculated combined member columns, we dynamically calculate the combined member columns of tables ν_0, ν_1, ν_2, ν_3 and ν_4 one by one. Now we elaborately explain the way to compute the cell values of the dynamic tables.

In ν_0, cell $\nu_0[\emptyset][\{u_1u_2\}]$ contains the minimum value of maximum trip overhead of trips T_1 and T_2, where the trips correspond to users u_1 and u_2, respectively, and visit no POI types. Table 4.13 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. As the trips visit no POI types, the minimum value of maximum trip overhead is zero here.

Table 4.13: Candidate combined combinations with trip overhead distances for cell $\nu_0[\emptyset][\{u_1u_2\}]$.

Combined Combinations	**Distances**	**Trip overhead**
$\max((\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]),$ $(\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}]))$	$\max((28.75 - 28.75), (25.00 - 25.00))$	0.0

Table 4.14: Candidate combined combinations with trip overhead distances for cell $\nu_1[\{c_1\}][\{u_1u_2\}]$.

Combined Combinations	**Distances**	**Trip overhead**
$\max(\nu_1[\{c_1\}][\{u_1\}], (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}]))$	$\max(105.36, (25.00 - 25.00))$	105.36
$\max((\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]), \nu_1[\{c_1\}][\{u_2\}])$	$\max((28.75 - 28.75), 76.55)$	76.55

To compute the values for the cells of the combined member columns for any table ν_y, we need to consider all dynamic tables from ν_0 to ν_y. For example, in ν_2, cell $\nu_2[\{c_1, c_2\}][\{u_1 u_2\}]$ stores the minimum value of maximum trip overhead 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 already computed and stored single member trip overhead distances and combined member trip overhead distances in ν_0, ν_1 and ν_2. Using ν_0, ν_1 and ν_2 (Tables 4.12(a-c)), Table 4.15 shows the candidate combinations of POI types for u_1 and u_2 along with the trip overhead distances for computing the value for cell $\nu_2[\{c_1, c_2\}][\{u_1 u_2\}]$ in ν_2 (Table 4.12(c)). Among candidate combinations listed in Table 4.15, the minimum value of maximum trip distance 76.55 is stored in cell $\nu_2[\{c_1, c_2\}][\{u_1 u_2\}]$.

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 overhead distances for cell $\nu_1[\{c_1\}][\{u_1 u_2\}]$, $\nu_3[\{c_1, c_2, c_3\}][\{u_1 u_2\}]$ and $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1 u_2\}]$ are listed in Table 4.14, Table 4.16 and Table 4.17, respectively.

Table 4.15: Candidate combined combinations with trip overhead distances for cell $\nu_2[\{c_1, c_2\}][\{u_1 u_2\}]$.

Combined Combinations	**Distances**	**Trip overhead**
$\max(\nu_2[\{c_1, c_2\}][\{u_1\}], (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}]))$	$\max(105.37, (25.00 - 25.00))$	105.37
$\max(\nu_1[\{c_1\}][\{u_1\}], \nu_1[\{c_2\}][\{u_2\}])$	$\max(105.36, 23.99)$	105.36
$\max(\nu_1[\{c_2\}][\{u_1\}], \nu_1[\{c_1\}][\{u_2\}])$	$\max(49.05, 76.55)$	76.55
$\max((\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]), \nu_2[\{c_1, c_2\}][\{u_2\}])$	$\max((28.75 - 28.75), 76.58)$	76.58

Table 4.16: Candidate combined combinations with trip overhead distances for cell $\nu_3[\{c_1, c_2, c_3\}][\{u_1 u_2\}]$

Combined Combinations	**Distances**	**Trip overhead**
$\max(\nu_3[\{c_1, c_2, c_3\}][\{u_1\}], (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}]))$	$\max(105.41, (25.00 - 25.00))$	105.41
$\max(\nu_2[\{c_1, c_2\}][\{u_1\}], \nu_1[\{c_3\}][\{u_2\}])$	$\max(105.37, 67.80)$	105.37
$\max(\nu_2[\{c_1, c_3\}][\{u_1\}], \nu_1[\{c_2\}][\{u_2\}])$	$\max(105.41, 23.99)$	105.41
$\max(\nu_2[\{c_2, c_3\}][\{u_1\}], \nu_1[\{c_1\}][\{u_2\}])$	$\max(105.34, 76.55)$	105.34
$\max(\nu_1[\{c_1\}][\{u_1\}], \nu_2[\{c_2, c_3\}][\{u_2\}])$	$\max(105.36, 69.92)$	105.36
$\max(\nu_1[\{c_2\}][\{u_1\}], \nu_2[\{c_1, c_3\}][\{u_2\}])$	$\max(49.05, 78.54)$	78.54
$\max(\nu_1[\{c_3\}][\{u_1\}], \nu_2[\{c_1, c_2\}][\{u_2\}])$	$\max(105.32, 76.58)$	105.32
$\max((\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]), \nu_3[\{c_1, c_2, c_3\}][\{u_2\}])$	$\max((28.75 - 28.75), 78.54)$	78.54

Table 4.17: Candidate combined combinations with trip overhead distances for cell $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1 u_2\}]$

Combined Combinations	**Distances**	**Trip overhead**
$\max(\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1\}], (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}]))$	$\max(107.15, (25.00 - 25.00))$	107.15
$\max(\nu_3[\{c_1, c_2, c_3\}][\{u_1\}], \nu_1[\{c_4\}][\{u_2\}])$	$\max(105.41, 70.29)$	105.41
$\max(\nu_3[\{c_1, c_2, c_4\}][\{u_1\}], \nu_1[\{c_3\}][\{u_2\}])$	$\max(107.06, 67.80)$	107.06
$\max(\nu_3[\{c_1, c_3, c_4\}][\{u_1\}], \nu_1[\{c_2\}][\{u_2\}])$	$\max(107.03, 23.99)$	107.03
$\max(\nu_3[\{c_2, c_3, c_4\}][\{u_1\}], \nu_1[\{c_1\}][\{u_2\}])$	$\max(107.14, 76.55)$	107.14
$\max(\nu_2[\{c_1, c_2\}][\{u_1\}], \nu_2[\{c_3, c_4\}][\{u_2\}])$	$\max(105.37, 73.28)$	105.37
$\max(\nu_2[\{c_1, c_3\}][\{u_1\}], \nu_2[\{c_2, c_4\}][\{u_2\}])$	$\max(105.41, 72.64)$	105.41
$\max(\nu_2[\{c_1, c_4\}][\{u_1\}], \nu_2[\{c_2, c_3\}][\{u_2\}])$	$\max(106.98, 69.92)$	106.98
$\max(\nu_2[\{c_2, c_3\}][\{u_1\}], \nu_2[\{c_1, c_3\}][\{u_2\}])$	$\max(105.34, 81.84)$	105.34
$\max(\nu_2[\{c_2, c_4\}][\{u_1\}], \nu_2[\{c_1, c_3\}][\{u_2\}])$	$\max(106.19, 78.54)$	106.19
$\max(\nu_2[\{c_3, c_4\}][\{u_1\}], \nu_2[\{c_1, c_2\}][\{u_2\}])$	$\max(107.01, 76.58)$	107.01
$\max(\nu_1[\{c_1\}][\{u_1\}], \nu_3[\{c_2, c_3, c_4\}][\{u_2\}])$	$\max(105.36, 73.52)$	105.36
$\max(\nu_1[\{c_2\}][\{u_1\}], \nu_3[\{c_1, c_3, c_4\}][\{u_2\}])$	$\max(49.05, 81.84)$	81.84
$\max(\nu_1[\{c_3\}][\{u_1\}], \nu_3[\{c_1, c_2, c_4\}][\{u_2\}])$	$\max(105.32, 81.84)$	105.32
$\max(\nu_1[\{c_4\}][\{u_1\}], \nu_3[\{c_1, c_2, c_3\}][\{u_2\}])$	$\max(102.51, 78.54)$	102.51
$\max((\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]), \nu_4[\{c_1, c_2, c_3, c_4\}][\{u_2\}])$	$\max((28.75 - 28.75), 81.84)$	81.84

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 value of maximum trip overhead 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 4.12(a-c)), Table 4.18 shows the candidate combinations of POI types for combined members u_1u_2 and single member u_3 along with the trip overhead distances for computing the value for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2u_3\}]$ in ν_2 (Table 4.12(c)).

Table 4.18: Candidate combined combinations with trip overhead distances for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2u_3\}]$.

Combined Combinations	**Distances**	**Trip overhead**
$\max(\nu_2[\{c_1, c_2\}][\{u_1u_2\}], (\nu_0[\emptyset][\{u_3\}] - \nu_0[\emptyset][\{u_3\}]))$	$\max(76.55, (47.55 - 47.55))$	76.55
$\max(\nu_1[\{c_1\}][\{u_1u_2\}], \nu_1[\{c_2\}][\{u_3\}])$	$\max(76.55, 7.03)$	76.55
$\max(\nu_1[\{c_2\}][\{u_1u_2\}], \nu_1[\{c_1\}][\{u_3\}])$	$\max(23.99, 57.61)$	57.61
$\max(\nu_0[\emptyset][\{u_1u_2\}], \nu_2[\{c_1, c_2\}][\{u_3\}])$	$\max(0.0, 57.61)$	57.61

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 value of maximum trip overhead 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 value of maximum trip overhead distance of the group of four members in our example scenario. The minimum value of maximum trip overhead distance 7.34 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 value of maximum trip overhead distance of the group which is *AggTripOvDist* that we have mentioned in Section 4.3.2. To get the values of T_{min_i} for each user u_i, we simply take the minimum values from Table 4.4(a). On the other hand, to get the values of T_{max_i} which is the maximum trip distance for each user u_i for visiting all required POI types, we take the maximum trip overhead distance values from Table 4.4(e) and then add the distance from s_i to d_i without visiting any POI types to get the actual maximum trip distance. T_{min_i} and T_{max_i} values for users $\{u_1, u_2, u_3, u_4\}$ are $\{28.75, 25.00, 47.55, 77.48\}$ and $\{(107.15 + 28.75), (81.84 + 25.00), (58.62 + 47.55), (7.36 + 77.48)\} \equiv \{135.90, 106.84, 106.17, 84.84\}$, respectively. Using these values we refine the search region based on Theorems 4.3.1 and 4.3.3. For user u_1, based on Theorem 4.3.1, the major axis for the elliptic region E_1 is 135.90. On the other hand, based on Theorem 4.3.3, the major axis is $7.34 + 28.75 = 36.09$. We take the best bound among them which is 36.09, the second one.

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

4.3.4.2 Trip Scheduling for UGTS Queries

In a Uniform GTS (UGTS) query where group members visit uniform number of POI types is an variation of our proposed GTS queries in spatial databases. In UGTS queries, a group of n members $\{u_1, u_2, \ldots, u_n\}$ with independent source and destination pairs $\{(s_1, d_1), (s_2, d_2), \ldots, (s_n, d_n)\}$ want to visit a set of specific m POI types $C = \{c_1, c_2, c_3, \ldots, c_m\}$ where each group member visits equal e number of POI types. For uniform distributions of POI types among group members, we assume that each group member visits an equal number of POI types. If m is a multiple of n, then $e = \lfloor \frac{m}{n} \rfloor$. If m is not a multiple of n, then $m \mod n$ number of group members visit $e = \lfloor \frac{m}{n} \rfloor + 1$ number of POI types, and the remaining group members visit $e = \lfloor \frac{m}{n} \rfloor$ number of POI types. For simplicity, we assume that, m is exact multiples of n and each user visits e number of POI types, where $m = n \times e$. We have to find multiple n trips for each user so that each user visits exact e number of POI types, each POI type is visited by exactly one user with minimum aggregate trip overhead distance of the group.

In a UGTS query, the definition of T_{min_i} and T_{max_i} slightly changes because of having the equal number of POI types visiting constraints. In this query any member should not visit no POI type or all required POI types. Thus, in a UGTS query, T_{min_i} and T_{max_i} represents the minimum and maximum trip distance of any trip covering any subset of e POI types from all required m POI types for a group member u_i, respectively. For computing the trip overhead distance of any trip, in a UGTS query, $(TripDist_i - T_{min_i})$ still represents the trip overhead distance of any group member u_i, where T_{min_i} represents the minimum value of trip distance for group member u_i.

For the UGTS queries, our dynamic programming approach that schedules trips, minimizes the following objective function:
$\sum_{i=1}^{n}(TripDist_i - T_{min_i})$, for aggregation function SUM,
$\max_{i=1}^{n}(TripDist_i - T_{min_i})$, for aggregation function MAX.

satisfying constrains that a group of n members need to visit m different POI types, where each group member visits e number of 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. Formal representation of the constraints are as follows. The dynamic programming approach has to satisfy,

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

For the UGTS queries, we have a set of m POI types $\mathbb{C}$=$\{c_1, c_2, \ldots, c_m\}$ where each member should visit e POI types. So there are ${}^{m}C_e$ or ${}^{|\mathbb{C}|}C_e$ different ways to choose e POI types from $m(= |\mathbb{C}|)$ different POI types. ${}^{\mathbb{C}}C_e$ denotes the set of all possible e choices from the set of POI types $\mathbb{C}$. Here $({}^{\mathbb{C}}C_e)^i$ represents the ith member of the set ${}^{\mathbb{C}}C_e$. For example, suppose we have a set of 6 POI types, $\mathbb{C} = \{c_1, c_2, c_3, c_4, c_5, c_6\}$, where each group member visits 2 POI types. Here, $m = |\mathbb{C}| = 6$ and $e = 2$. So, the number of different ways to choose e POI types from $m(= |\mathbb{C}|)$ different POI types is ${}^{|\mathbb{C}|}C_e = {}^{6}C_2 = 15$ and the set all possible e choices from the set $\mathbb{C}$ is ${}^{\mathbb{C}}C_e = \{\{c_1, c_2\}, \{c_1, c_3\}, \{c_1, c_4\}, \ldots, \{c_5, c_6\}\}$. Also we can say that, $({}^{\mathbb{C}}C_e)^1 = \{c_1, c_2\}$, $({}^{\mathbb{C}}C_e)^2 = \{c_1, c_3\}, \ldots, ({}^{\mathbb{C}}C_e)^{15} = \{c_5, c_6\}$.

As each group member visits e number of POI types, any two group members should visit any $2 \times e = 2e$ number of POI types among m required POI types, any three group members visit any $3 \times e = 3e$ number of POI types among m required POI types and so on. Thus, we define n dynamic tables, $\nu_e, \nu_{2e}, \ldots, \nu_{(n-1)e}, \nu_{(ne=m)}$ to store the trip distances of each single group member and the aggregate trip overhead distances of the combined group members. A dynamic table ν_y where $0 \leq y \leq m$, has ${}^{m}C_y$ rows, where jth row corresponds to jth member of the set ${}^{\mathbb{C}}C_y$, i.e., $({}^{\mathbb{C}}C_y)^j$.

For each member of the set ${}^{\mathbb{C}}C_e$, we calculate optimal trips for each group member in $U = \{u_1, u_2, u_3, \ldots, u_n\}$ and the resultant values are stored in dynamic table ν_e for future calculations. This is the initial step for our dynamic programming approach. Unlike to GTS queries, in UGTS queries, we store the trip distances instead of storing trip overhead distances for each group member to visit any subset of e POI types from m required POI types in table ν_e. In the UGTS queries, to compute trip overhead distance of any trip, we need to reduce T_{min_i} from the trip distance where the value of T_{min_i} can be computed after computing all possible trip distances that visit any e number of POI types. Because of having this type of circular dependency, we prefer to store the actual trip distance instead of storing the trip overhead distance in dynamic table ν_e for the UGTS queries.

Unlike to GTS queries, in a UGTS query, where each group member visits uniform number of POI types, each dynamic table has only types of column, either ***single member columns*** or ***combined member columns***. For having the uniform POI types visiting constraint, the group members should visit e number of POI types instead of visiting any number of POI types from 0 to m. So it is not necessary to compute all possible minimum trips or trip overhead distances that visit any number of POI types for all group members. We only need to compute minimum trips for visiting e number of POI types for every member of the group. Thus, table ν_e has n single member columns, where each column corresponds to a member of the group $U = \{u_1, u_2, u_3, \ldots, u_n\}$. 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. The table does not have any combined member columns as well, because unlike to GTS queries, for UGTS queries, it will not happen that any number of group members together visit any subset of e number of POI types from m required POI types.

On the other hand, for similar reason, other dynamic tables $\nu_{2e}, \ldots, \nu_{(n-1)e}, \nu_{ne=m}$ do not need to have single member columns. They also don't need to have all possible combined member columns $u_1u_2, \ldots, u_1u_2..u_{n-1}$. Instead of having all combined member columns, each of them has only one combined member column where the columns are $u_1u_2, \ldots, u_1u_2..u_{n-1}, u_1u_2..u_n$, for the dynamic tables $\nu_{2e}, \ldots, \nu_{(n-1)e}, \nu_{ne=m}$, respectively. The cells of the corresponding columns of each table store the aggregate trip overhead distances of the corresponding column's multiple members passing through the set of POI types of corresponding rows. For example, each cell of the column u_1u_2 stores the minimum total trip overhead distance or the minimum value of maximum trip overhead 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 4.19: Structure of dynamic table ν_e

	$\{u_1\}$	$\{u_2\}$	$\ldots$	$\{u_{(n-1)}\}$	$\{u_n\}$
$\{c_1, c_2, \ldots, c_e\}$					
$\{c_1, c_3, \ldots, c_e\}$					
$\vdots$					

Table 4.20: Structure of dynamic table ν_y, where $y \in \{2 \times e, \ldots, (n-1) \times e, n \times e\}$

	$\{u_1 u_2 \ldots u_{y/e}\}$
$\{c_1, c_2, \ldots, c_y\}$	
$\{c_1, c_3, \ldots, c_y\}$	
$\vdots$	

Table 4.19 shows the structure of ν_e and Table 4.20 shows the structure of other dynamic tables, $\nu_{2\times e}, \ldots, \nu_{(n-1)\times e}, \nu_{n\times e=m}$, that has only one combined member column. $\nu_{n\times e=m}$ that has only one column $u_1 u_2 \ldots u_n$ which stores the minimum total trip overhead distance or the minimum value of maximum trip overhead 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 final table ν_{ne} or ν_m, has only one row which contains all m POI types.

In addition to storing the minimum trip distances (single member columns of table ν_e) and the minimum aggregate trip overhead distances (combined member columns), each cell of the dynamic tables stores the set of POIs for which the minimum trip distance or minimum aggregate trip overhead distance is obtained. For example, cell $\nu_2[\{c_1, c_3\}][\{u_1\}]$ stores the minimum trip distance and the POI set $< p_3, p_1 >$, for which u_1 obtains the minimum trip or trip overhead distance.

Contents of the cells of the single member columns of dynamic table ν_e are computed using already retrieved POIs from the database. To compute the contents of the cells of the combined member columns of a dynamic table ν_ye, we use the single member columns of the table ν_e, and combined member columns of table $\nu_(y-1)e$.

For aggregate function SUM, any cell (e.g., $\nu_{ye}[^{\mathbb{C}}C_{ye}][\{u_1u_2\}]$) of this table is calculated using the equation : $\nu_{2e}[^{\mathbb{C}}C_{2e}][\{u_1u_2\}] = \min_{i,j=1}^{^mC_e}\{(\nu_e[(^{\mathbb{C}}C_e)^i][\{u_1\}] - T_{min_1}) + (\nu_e[(^{\mathbb{C}}C_e)^j][\{u_2\}] - T_{min_2})\}$, where $(^{\mathbb{C}}C_e)^i \cap (^{\mathbb{C}}C_e)^j = \emptyset$.

For aggregate function MAX, the equation is : $\nu_{2e}[^{\mathbb{C}}C_{2e}][\{u_1u_2\}] = \min_{i,j=1}^{^mC_e}\{\max((\nu_e[(^{\mathbb{C}}C_e)^i][\{u_1\}] - T_{min_1}), (\nu_e[(^{\mathbb{C}}C_e)^j][\{u_2\}] - T_{min_2}))\}$, where $(^{\mathbb{C}}C_e)^i \cap (^{\mathbb{C}}C_e)^j = \emptyset$.

The size of table ν_e is $^mC_e \times n$ and the size of a dynamic table ν_y is : $^mC_y \times 1$, where $y \in \{2 \times e, \ldots, (n-1) \times e, n \times e\}$. Thus, the total space required for dynamic tables is $^mC_e \times n + {^mC_{2\times e}} + \ldots + {^mC_{(n-2)\times e}} + {^mC_{n\times e=m}}$ 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.

Now we will give an elaborate example for the dynamic programming approach of UGTS queries. To explain the dynamic programming approach and to understand the intermediate steps of the trip scheduling for the UGTS queries where each group member visits uniform number of POI types, we consider an example scenario where have a group of 3 members, $\{u_1, u_2, u_3\}$ with source-destination pairs $< s_1, d_1 >$, $< s_2, d_3 >$ and s_3, d_3, respectively. The group members need to visit 6 different POI types $\{c_1, c_2, c_3, c_4, c_5, c_6\}$ with minimum total trip overhead distance. Here we have In this scenario, each group member visits 2 POI types. We have, $n = 3$, $m = 6$ and $e = 2$.

After finding at least one POI from every required POI types, our approach computes all possible sub trips for the group members and then we compute the scheduled trips using our proposed dynamic programming approach. Now we will simulate the approach for our example scenario.

For our example scenario, Tables 4.21(a-c) represents the complete structure of the dynamic tables ν_2, ν_4 and ν_6 to store the computed trip distances of the single members and combined trip overhead distances of the multiple group members.

Computing single member columns: In the dynamic table ν_2, columns u_1, u_2 and u_3 are the single member columns. Each cell of these columns of table ν_e 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 4.21(a), 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 that has been retrieved from the database with POI types c_1 and c_2, respectively.

Table 4.21: Dynamic tables for UGTS queries with aggregate function SUM

(a) Dynamic table ν_2

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$
$\{c_1, c_2\}$	50.46	30.75	66.36
$\{c_1, c_3\}$	50.05	24.16	66.35
$\{c_1, c_4\}$	45.95	24.74	66.34
$\{c_1, c_5\}$	50.09	23.90	66.38
$\{c_1, c_6\}$	60.64	32.61	66.34
$\{c_2, c_3\}$	50.55	29.14	66.34
$\{c_2, c_4\}$	50.46	29.31	66.34
$\{c_2, c_5\}$	50.53	29.10	66.35
$\{c_2, c_6\}$	60.64	32.62	66.35
$\{c_3, c_4\}$	50.19	24.34	66.34
$\{c_3, c_5\}$	50.27	21.55	66.37
$\{c_3, c_6\}$	60.63	32.61	66.34
$\{c_4, c_5\}$	50.56	24.25	66.34
$\{c_4, c_6\}$	60.71	32.61	64.34
$\{c_5, c_6\}$	60.63	32.61	66.38

(b) Dynamic table ν_4

	$\{u_1 u_2\}$
$\{c_2, c_3, c_4, c_5\}$	4.51
$\{c_2, c_3, c_4, c_6\}$	15.31
$\{c_2, c_3, c_5, c_6\}$	14.69
$\{c_2, c_4, c_5, c_6\}$	15.57
$\{c_3, c_4, c_5, c_6\}$	14.76
$\{c_1, c_3, c_4, c_5\}$	0.00
$\{c_1, c_3, c_4, c_6\}$	11.06
$\{c_1, c_3, c_5, c_6\}$	14.69
$\{c_1, c_4, c_5, c_6\}$	11.06
$\{c_1, c_2, c_4, c_5\}$	6.86
$\{c_1, c_2, c_4, c_6\}$	11.07
$\{c_1, c_2, c_5, c_6\}$	15.21
$\{c_1, c_2, c_3, c_5\}$	4.51
$\{c_1, c_2, c_3, c_6\}$	15.17
$\{c_1, c_2, c_3, c_4\}$	7.12

(c) Dynamic table ν_6

	$\{u_1 u_2 u_3\}$
$\{c_1, c_2, c_3, c_4, c_5, c_6\}$	2.01

Using the computed values in Table 4.21(a), we compute the T_{min_i} value for group member u_i by taking the minimum value of the cells of single member column u_i of the dynamic table ν_e. Table 4.22 shows the T_{min_i} values of the group member u_i.

Table 4.22: T_{min_i} values for three group members of the example scenario

	Distances
T_{min_1}	45.95
T_{min_2}	21.55
T_{min_3}	64.34

Computing combined member columns: Using the single member columns of dynamic table ν_2, we dynamically calculate the combined member column of table ν_4. Gradually using the single member columns of dynamic table ν_2 and already computed combined user column of dynamic table ν_4, we dynamically calculate the combined member column of dynamic table ν_6.

In Table 4.21(b), cell $\nu_4[\{c_2, c_3, c_4, c_5\}][\{u_1 u_2\}]$ contains the minimum total trip overhead distance of trips T_1 and T_2 where the trips corresponds to user u_1 and u_2, respectively and together the trips visit the POI types $\{c_2, c_3, c_4, c_5\}$ where each POI type is visited by either user u_1 or user u_2. To compute the cell value, we use precomputed trip distances which have been stored in Table 4.21(a). All candidate combined combinations for both user along with trip overhead distances using Table 4.21(a) are listed in Table 4.23, to compute value of cell $\nu_4[\{c_2, c_3, c_4, c_5\}][\{u_1 u_2\}]$.

Among all candidate combined combinations listed in Table 4.23, the best combined trip overhead distance is stored in cell $\nu_4[\{c_2, c_3, c_4, c_5\}][\{u_1 u_2\}]$ which is 4.51. Similarly, our dynamic programming approach populates all cells of Table 4.21(b).

Using precomputed Tables 4.21(a) and 4.21(b), the dynamic programming approach computes the next table which is Table 4.21(c). The table has only one cell which is $\nu_6[\{c_1, c_2, c_3, c_4, c_5, c_6\}][\{u_1 u_2 u_3\}]$ that contains the minimum total trip overhead 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 each required POI types $\{c_1, c_2, c_3, c_4, c_5, c_6\}$ is included in a single trip. This is actually our minimum total trip overhead distance of the group

Table 4.23: Candidate combined combinations with trip overhead distances for cell $\nu_4[\{c_2, c_3, c_4, c_5\}][\{u_1 u_2\}]$.

Combined Combinations	**Distances**	**Trip over-head**
$(\nu_2[\{c_2, c_3\}][\{u_1\}] - T_{min_1}) + (\nu_2[\{c_4, c_5\}][\{u_2\}] - T_{min_2})$	$(50.55 - 45.95) + (24.25 - 21.55)$	7.30
$(\nu_2[\{c_2, c_4\}][\{u_1\}] - T_{min_1}) + (\nu_2[\{c_3, c_5\}][\{u_2\}] - T_{min_2})$	$(50.46 - 45.95) + (21.55 - 21.55)$	4.51
$(\nu_2[\{c_2, c_5\}][\{u_1\}] - T_{min_1}) + (\nu_2[\{c_3, c_4\}][\{u_2\}] - T_{min_2})$	$(50.53 - 45.95) + (24.34 - 21.55)$	7.37
$(\nu_2[\{c_3, c_4\}][\{u_1\}] - T_{min_1}) + (\nu_2[\{c_2, c_5\}][\{u_2\}] - T_{min_2})$	$(50.19 - 45.95) + (29.10 - 21.55)$	11.79
$(\nu_2[\{c_3, c_5\}][\{u_1\}] - T_{min_1}) + (\nu_2[\{c_2, c_4\}][\{u_2\}] - T_{min_2})$	$(50.27 - 45.95) + (29.31 - 21.55)$	12.08
$(\nu_2[\{c_4, c_5\}][\{u_1\}] - T_{min_1}) + (\nu_2[\{c_2, c_3\}][\{u_2\}] - T_{min_2})$	$(50.56 - 45.95) + (29.14 - 21.55)$	12.20

for the dynamic trip scheduling. For computing the cell value, we use precomputed values which have been stored in Tables 4.21(a) and 4.21(b). To compute the cell value, all candidate combined combinations along with trip overhead distances are listed in Table 4.24.

Among all candidate combinations listed in Table 4.24, the best combined trip overhead distance is stored in cell $\nu_6[\{c_1, c_2, c_3, c_4, c_5, c_6\}][\{u_1 u_2 u_3\}]$ which is 2.01.

Note that the only cell of Table 4.21(c) contains the minimum total trip overhead distance of the group which is *AggTripOvDist* that we have mentioned in Section 4.3.2. To get the values of T_{min_i} and T_{max_i} for each user u_i, we simply take the minimum and maximum value, respectively, from Table 4.21(a) for all the rows of respective user's column. The T_{min_i} and T_{max_i} values for users $\{u_1, u_2, u_3\}$ are $\{45.95, 21.55, 64.34\}$ and $\{60.71, 32.62, 66.38\}$, respectively. Using these values we refined search region based on Theorems 4.3.1 and 4.3.2. For user u_1, based on Theorem 4.3.1, the major axis for the elliptic region E_1 is 60.71. On the other hand, based on Theorem 4.3.2, the major axis is $2.01+45.95 = 47.96$. We take the best bound among them which is 47.96, the second one.

Table 4.24: Candidate combined combinations with trip overhead distances for cell $\nu_6[\{c_1, c_2, c_3, c_4, c_5, c_6\}][\{u_1u_2u_3\}]$

Combined Combinations	**Distances**	**Trip overhead**
$\nu_4[\{c_2, c_3, c_4, c_5\}][\{u_1u_2\}] + (\nu_2[\{c_1, c_6\}][\{u_3\}] - T_{min_3})$	$4.51 + (66.34 - 64.34)$	6.51
$\nu_4[\{c_2, c_3, c_4, c_6\}][\{u_1u_2\}] + (\nu_2[\{c_1, c_5\}][\{u_3\}] - T_{min_3})$	$15.31 + (66.38 - 64.34)$	17.35
$\nu_4[\{c_2, c_3, c_5, c_6\}][\{u_1u_2\}] + (\nu_2[\{c_1, c_4\}][\{u_3\}] - T_{min_3})$	$14.69 + (66.34 - 64.34)$	16.69
$\nu_4[\{c_2, c_4, c_5, c_6\}][\{u_1u_2\}] + (\nu_2[\{c_1, c_3\}][\{u_3\}] - T_{min_3})$	$15.57 + (66.35 - 64.34)$	17.58
$\nu_4[\{c_3, c_4, c_5, c_6\}][\{u_1u_2\}] + (\nu_2[\{c_1, c_2\}][\{u_3\}] - T_{min_3})$	$14.76 + (66.36 - 64.34)$	16.78
$\nu_4[\{c_1, c_3, c_4, c_5\}][\{u_1u_2\}] + (\nu_2[\{c_2, c_6\}][\{u_3\}] - T_{min_3})$	$0.00 + (66.35 - 64.34)$	2.01
$\nu_4[\{c_1, c_3, c_4, c_6\}][\{u_1u_2\}] + (\nu_2[\{c_2, c_5\}][\{u_3\}] - T_{min_3})$	$11.06 + (66.35 - 64.34)$	13.07
$\nu_4[\{c_1, c_3, c_5, c_6\}][\{u_1u_2\}] + (\nu_2[\{c_2, c_4\}][\{u_3\}] - T_{min_3})$	$14.69 + (66.34 - 64.34)$	16.69
$\nu_4[\{c_1, c_4, c_5, c_6\}][\{u_1u_2\}] + (\nu_2[\{c_2, c_3\}][\{u_3\}] - T_{min_3})$	$11.06 + (66.34 - 64.34)$	13.06
$\nu_4[\{c_1, c_2, c_4, c_5\}][\{u_1u_2\}] + (\nu_2[\{c_3, c_6\}][\{u_3\}] - T_{min_3})$	$6.86 + (66.34 - 64.34)$	8.86
$\nu_4[\{c_1, c_2, c_4, c_6\}][\{u_1u_2\}] + (\nu_2[\{c_3, c_5\}][\{u_3\}] - T_{min_3})$	$11.07 + (66.37 - 64.34)$	13.10
$\nu_4[\{c_1, c_2, c_5, c_6\}][\{u_1u_2\}] + (\nu_2[\{c_3, c_4\}][\{u_3\}] - T_{min_3})$	$15.21 + (66.34 - 64.34)$	17.21
$\nu_4[\{c_1, c_2, c_3, c_5\}][\{u_1u_2\}] + (\nu_2[\{c_4, c_6\}][\{u_3\}] - T_{min_3})$	$4.51 + (64.34 - 64.34)$	4.51
$\nu_4[\{c_1, c_2, c_3, c_6\}][\{u_1u_2\}] + (\nu_2[\{c_4, c_5\}][\{u_3\}] - T_{min_3})$	$15.17 + (66.34 - 64.34)$	17.17
$\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1u_2\}] + (\nu_2[\{c_5, c_6\}][\{u_3\}] - T_{min_3})$	$7.12 + (66.38 - 64.34)$	9.16

For aggregation function MAX, the dynamic programming approach will be similar that we have already described for aggregate function SUM. Instead of taking the summation of the trip overhead distances of different combinations, we have to take the maximum values of them. Thus, we skipped to give elaborate example for the UGTS queries with aggregate function MAX.

4.3.4.3 Extensions of Trip Scheduling for Dependencies Among POIs

For processing the GTS and the UGTS query with dependencies among POIs constraint, we have to satisfy user provided POI dependencies along with satisfying other constraints for the GTS and the UGTS query as well. For this variation of the GTS or the UGTS query, the dynamic programming approach for trip scheduling is almost similar with the dynamic programming approach that has been described in Section 4.3.4.1 for the GTS query and in Section 4.3.4.2 for the UGTS query without having user defined constraints.

For having dependencies among POIs, some combinations of POI type will become invalid which we should not consider while scheduling trips using our proposed dynamic programming approach for each member in the group. For example, suppose, a group of n members $\{u_1, u_2, \ldots, u_n\}$ need to visit m POI types $\{c_1, c_2, \ldots, c_m\}$ with minimum aggregate trip overhead distance where each POI type is visited exactly once by any group member. The group impose a constraint that, any member of the group need to visit POI type c_1 first and then POI type c_2. The constraint follows that, POI types c_1 and c_2 should be visited one by one by any member of the group and who will visit these POI types should visit POI type c_1 before visiting POI type c_2. The dependency among POI types c_1 and c_2 also assures that POI types c_1 and c_2 should not be visited by any group members separately. So some POI type combinations and also some user and POI type combinations become invalid for the variation of the GTS or the UGTS query.

***Now we will give an elaborate example for dynamic programming approach with this variation of GTS queries "dependencies among POIs" for aggregate function* sum.** To explain the dynamic programming approach elaborately, we use the example scenario that we have used in Section 4.3.4.1 for the aggregate function SUM. The purpose of using similar scenario is to easily find out the changes for having the constraint dependencies among POIs. In this example scenario, 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 overhead distance, and each POI type is visited by a single member. The group impose a constraint that, any member of the group need to visit POI type c_1 first and then POI type c_2. Here, $n = 4$, $m = 4$, and a group member can visit any number of POI types between 0 to m.

Following the similar process described in Section 4.3.4.1, we define $(m + 1)$, i.e., 5 tables, ν_0, ν_1, ν_2, ν_3 and ν_4 to store the computed trip overhead distances and combined trip overhead distances of the group members. As the user group have imposed some constraints among POIs, all combinations of dynamic tables will not valid. Some combinations will become invalid for this variation of GTS queries. The constraint of visiting POI type c_1 first and then POI type c_2 follows that, both POI types c_1 and c_2 should be visited one by one by any member of the group and who will visit these POI types should visit POI type c_1 before visiting POI type c_2. Tables 4.25(a-e) show ν_0, ν_1, ν_2,

Table 4.25: Dynamic tables for GTS queries with dependency between POI types c_1 and c_2 for aggregate function SUM

(a) Dynamic table ν_0

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\emptyset$	28.75	25.00	47.55	77.48	0.0	0.0

(b) Dynamic table ν_1

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
~~$\{c_1\}$~~	~~105.36~~	~~76.55~~	~~57.61~~	~~7.32~~	~~76.55~~	~~57.61~~
~~$\{c_2\}$~~	~~49.05~~	~~23.99~~	~~7.03~~	~~4.72~~	~~23.99~~	~~7.03~~
$\{c_3\}$	105.32	67.80	41.85	5.42	67.80	41.85
$\{c_4\}$	102.51	70.29	42.02	5.65	70.29	42.02

(c) Dynamic table ν_2

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1, c_2\}$	106.62	76.58	57.61	7.36	76.58	57.61
~~$\{c_1, c_3\}$~~	~~105.41~~	~~78.54~~	~~57.98~~	~~7.32~~	~~78.54~~	~~57.98~~
~~$\{c_1, c_4\}$~~	~~106.98~~	~~81.84~~	~~58.61~~	~~7.34~~	~~81.84~~	~~58.61~~
~~$\{c_2, c_3\}$~~	~~105.34~~	~~69.92~~	~~41.86~~	~~5.46~~	~~69.92~~	~~41.86~~
~~$\{c_2, c_4\}$~~	~~106.19~~	~~72.64~~	~~43.19~~	~~5.67~~	~~72.64~~	~~43.19~~
$\{c_3, c_4\}$	107.01	73.28	43.62	5.83	73.28	43.62

(d) Dynamic table ν_3

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1, c_2, c_3\}$	105.41	78.54	57.98	7.36	78.54	57.98
$\{c_1, c_2, c_4\}$	107.06	81.84	58.62	7.36	81.84	58.62
~~$\{c_1, c_3, c_4\}$~~	~~107.03~~	~~81.84~~	~~58.61~~	~~7.34~~	~~81.84~~	~~58.61~~
~~$\{c_2, c_3, c_4\}$~~	~~107.14~~	~~73.52~~	~~43.93~~	~~5.86~~	~~73.52~~	~~43.93~~

(e) Dynamic table ν_4

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$	$\{u_1u_2u_3u_4\}$
$\{c_1, c_2, c_3, c_4\}$	107.15	81.84	58.62	7.36	81.84	58.62	7.36

ν_3 and ν_4 for the considered example. For having constraint, the cells of invalid combinations are crossed out in these dynamic tables.

Computing single member columns: In the dynamic tables, each cell of the single member columns u_1, u_2, u_3 and u_4 of a table stores the minimum trip overhead distance for the corresponding column's user passing through POI types of the corresponding row of that table. While calculating the trip overhead distances, we also consider the imposed constraint by the group members too. For example, in Table 4.25(c), cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$ contains the minimum trip overhead distance for user u_1 passing through POI types c_1 and c_2 with having POI type order, $c_1 \rightarrow c_2$. 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 overhead distances are listed in Table 4.26.

Table 4.26: Candidate trips with trip overhead distances for cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$

Candidate trips	**Trip distances**
~~$s_1 \rightarrow p_2^1 \rightarrow p_1^1 \rightarrow d_1$~~	~~105.37~~
~~$s_1 \rightarrow p_2^1 \rightarrow p_1^2 \rightarrow d_1$~~	~~109.89~~
$s_1 \rightarrow p_1^1 \rightarrow p_2^1 \rightarrow d_1$	106.62
$s_1 \rightarrow p_1^2 \rightarrow p_2^1 \rightarrow d_1$	126.58

Among the candidate trips listed in this table, the minimum trip overhead distance is 105.37 for trip $s_1 \rightarrow p_2^1 \rightarrow p_1^1 \rightarrow d_1$, but the POI type order is $c_2 \rightarrow c_1$ doesn't match with the given constraint which is $c_1 \rightarrow c_2$. Thus, we choose the trip overhead distance 106.62 for trip $s_1 \rightarrow p_1^1 \rightarrow p_2^1 \rightarrow d_1$ which satisfies the constraint and the value 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 (single member columns) and trip overhead distance (combined member columns).

Computing combined member columns: Using the valid single member columns and already calculated valid 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. Candidate combinations with trip overhead distances for cell $\nu_0[\emptyset][\{u_1u_2\}]$, $\nu_1[\{c_1\}][\{u_1u_2\}]$, $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$, $\nu_3[\{c_1, c_2, c_3\}][\{u_1u_2\}]$ and $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1u_2\}]$ are listed in Table 4.27, Table 4.28, Table 4.29, Table 4.30 and Table 4.31, respectively. We have used the similar example that we have used in Section 4.3.4.1 and have crossed out the invalid combinations for each table so that we can understand the main differences for having the constraint. Note that for having constraint of fixed POI type sequence, the cell $\nu_1[\{c_1\}][\{u_1u_2\}]$ in table ν_1 become invalid and we do not need to calculate this cell anymore. Thus all the combinations of Table 4.28 has been crossed out because all those combinations are invalid.

Table 4.27: Candidate combined combinations with trip overhead distances for cell $\nu_0[\emptyset][\{u_1u_2\}]$.

Combined combinations	**Distances**	**Trip overhead**
$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$	$(28.75 - 28.75) + (25.00 - 25.00)$	0.00

Table 4.28: Candidate combined combinations with trip overhead distances for cell $\nu_1[\{c_1\}][\{u_1u_2\}]$.

Combined combinations	**Distances**	**Trip overhead**
~~$\nu_1[\{c_1\}][\{u_1\}] + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$~~	~~$105.36 + (25.00 - 25.00)$~~	~~105.36~~
~~$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + \nu_1[\{c_1\}][\{u_2\}]$~~	~~$(28.75 - 28.75) + 76.55$~~	~~76.55~~

Table 4.29: Candidate combined combinations with trip overhead distances for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$.

Combined Combinations	**Distances**	**Trip overhead**
$\nu_2[\{c_1, c_2\}][\{u_1\}] + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$	$106.62 + (25.00 - 25.00)$	106.62
~~$\nu_1[\{c_1\}][\{u_1\}] + \nu_1[\{c_2\}][\{u_2\}]$~~	~~$105.36 + 23.99$~~	~~129.35~~
~~$\nu_1[\{c_2\}][\{u_1\}] + \nu_1[\{c_1\}][\{u_2\}]$~~	~~$49.05 + 76.55$~~	~~125.60~~
$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + \nu_2[\{c_1, c_2\}][\{u_2\}]$	$(28.75 - 28.75) + 76.58$	76.58

Note that 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 overhead 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 also satisfies user provided depen-

Table 4.30: Candidate combined combinations with trip overhead distances for cell $\nu_3[\{c_1, c_2, c_3\}][\{u_1 u_2\}]$

Combined combinations	**Distances**	**Trip overhead**
$\nu_3[\{c_1, c_2, c_3\}][\{u_1\}] + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$	$105.41 + (25.00 - 25.00)$	105.41
$\nu_2[\{c_1, c_2\}][\{u_1\}] + \nu_1[\{c_3\}][\{u_2\}]$	$106.62 + 67.80$	174.42
~~$\nu_2[\{c_1, c_3\}][\{u_1\}] + \nu_1[\{c_2\}][\{u_2\}]$~~	~~$105.41 + 23.99$~~	~~129.40~~
~~$\nu_2[\{c_2, c_3\}][\{u_1\}] + \nu_1[\{c_1\}][\{u_2\}]$~~	~~$105.34 + 76.55$~~	~~181.89~~
~~$\nu_1[\{c_1\}][\{u_1\}] + \nu_2[\{c_2, c_3\}][\{u_2\}]$~~	~~$105.36 + 69.92$~~	~~175.28~~
~~$\nu_1[\{c_2\}][\{u_1\}] + \nu_2[\{c_1, c_3\}][\{u_2\}]$~~	~~$49.05 + 78.54$~~	~~127.59~~
$\nu_1[\{c_3\}][\{u_1\}] + \nu_2[\{c_1, c_2\}][\{u_2\}]$	$105.32 + 76.58$	181.90
$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + \nu_3[\{c_1, c_2, c_3\}][\{u_2\}]$	$(28.75 - 28.75) + 78.54$	78.54

Table 4.31: Candidate combined combinations with trip overhead distances for cell $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1 u_2\}]$

Combined Combinations	**Distances**	**Trip overhead**
$\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1\}] + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$	$107.15 + (25.00 - 25.00)$	107.15
$\nu_3[\{c_1, c_2, c_3\}][\{u_1\}] + \nu_1[\{c_4\}][\{u_2\}]$	$105.41 + 70.29$	175.70
$\nu_3[\{c_1, c_2, c_4\}][\{u_1\}] + \nu_1[\{c_3\}][\{u_2\}]$	$107.06 + 67.80$	174.86
~~$\nu_3[\{c_1, c_3, c_4\}][\{u_1\}] + \nu_1[\{c_2\}][\{u_2\}]$~~	~~$107.03 + 23.99$~~	~~131.02~~
~~$\nu_3[\{c_2, c_3, c_4\}][\{u_1\}] + \nu_1[\{c_1\}][\{u_2\}]$~~	~~$107.14 + 76.55$~~	~~183.69~~
$\nu_2[\{c_1, c_2\}][\{u_1\}] + \nu_2[\{c_3, c_4\}][\{u_2\}]$	$106.62 + 73.28$	179.90
~~$\nu_2[\{c_1, c_3\}][\{u_1\}] + \nu_2[\{c_2, c_4\}][\{u_2\}]$~~	~~$105.41 + 72.64$~~	~~178.05~~
~~$\nu_2[\{c_1, c_4\}][\{u_1\}] + \nu_2[\{c_2, c_3\}][\{u_2\}]$~~	~~$106.98 + 69.92$~~	~~176.90~~
~~$\nu_2[\{c_2, c_3\}][\{u_1\}] + \nu_2[\{c_1, c_3\}][\{u_2\}]$~~	~~$105.34 + 81.84$~~	~~187.18~~
~~$\nu_2[\{c_2, c_4\}][\{u_1\}] + \nu_2[\{c_1, c_3\}][\{u_2\}]$~~	~~$106.19 + 78.54$~~	~~184.73~~
$\nu_2[\{c_3, c_4\}][\{u_1\}] + \nu_2[\{c_1, c_2\}][\{u_2\}]$	$107.01 + 76.58$	183.59
~~$\nu_1[\{c_1\}][\{u_1\}] + \nu_3[\{c_2, c_3, c_4\}][\{u_2\}]$~~	~~$105.36 + 73.52$~~	~~178.88~~
~~$\nu_1[\{c_2\}][\{u_1\}] + \nu_3[\{c_1, c_3, c_4\}][\{u_2\}]$~~	~~$49.05 + 81.84$~~	~~130.89~~
$\nu_1[\{c_3\}][\{u_1\}] + \nu_3[\{c_1, c_2, c_4\}][\{u_2\}]$	$105.32 + 81.84$	187.16
$\nu_1[\{c_4\}][\{u_1\}] + \nu_3[\{c_1, c_2, c_3\}][\{u_2\}]$	$102.51 + 78.54$	181.05
$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + \nu_4[\{c_1, c_2, c_3, c_4\}][\{u_2\}]$	$(28.75 - 28.75) + 81.84$	81.84

dencies among POI types c_1 and c_2. The cell contains the minimum total trip overhead distance of the group which is $AggTripOvDist$ that we have mentioned in Section 4.3.2. To get the values of T_{min_i} and T_{max_i} for each user u_i, we simply take the minimum trip distance (from table ν_0) and the minimum trip overhead distance (from table ν_m) from Table 4.32(a) and Table 4.32(e), respectively and add the distance from source (s_i) to destination (d_i) with the minimum trip overhead distance to get the actual trip distance for user u_i. T_{min_i} and T_{max_i} values for users $\{u_1, u_2, u_3, u_4\}$ are $\{28.75, 25.00, 47.55, 77.48\}$ and $\{(107.15+28.75), (81.84+25.00), (58.62+47.55), (7.36+77.48)\} \equiv \{135.90, 106.84, 106.17, 84.84\}$, respectively. Using these values we refine the search region based on Theorems 4.3.1 and 4.3.2. For user u_1, based on Theorem 4.3.1, the major axis for the elliptic region E_1 is 135.90. On the other hand, based on Theorem 4.3.2, the major axis is $7.36 + 28.75 = 36.11$. We take the best bound among them which is 36.11, 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 overhead distance is obtained with satisfying the constraint. For the sake of clarity we do not show them in the tables.

4.3.4.4 Extensions of Trip Scheduling for Dependencies Among Users and POIs

In a GTS or a UGTS query with the constraint of dependencies among users and POIs, we have to satisfy user provided POI dependencies with users along with satisfying all other constraints for the GTS and the UGTS query that we have to satisfy without having the user provided constraints as well. The dynamic programming approach to schedule trip among the group members for this variation of the GTS or the UGTS query is almost similar with the dynamic programming approach that has been described in Section 4.3.4.1 for the GTS query and in Section 4.3.4.2 for the UGTS query.

Some combinations of POI types and users or group members will become invalid for having user defined dependencies among users and POIs which we should not consider while scheduling trips using our proposed dynamic programming approach for every members of the group. For example, suppose, a group of n members $\{u_1, u_2, \ldots, u_n\}$ need to visit m POI types $\{c_1, c_2, \ldots, c_m\}$ with minimum aggregate trip overhead distance where each POI type is visited exactly once by any group member. The group impose a constraint that group member u_1 should visit POI type c_1. This constraint follows that, POI type c_1 should not be visited by other members of the group. So the combinations of POI type c_1 and group members except member u_1 will be invalid and we have to ignore these combinations from our computation while scheduling trips for the group members.

Now we will give an elaborate example for dynamic programming approach with the variation of GTS queries "dependencies among users and POIs" for aggregate function sum. To explain our proposed dynamic programming technique elaborately, we use the example scenario that we have used to explain the dynamic programming technique for GTS queries in Section 4.3.4.1 for aggregate function SUM. This will help us to find out the changes that we have to make for scheduling trips with the constraint more specifically. In this example scenario, 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 overhead distance, and each POI type is visited by a single member. The group impose a constraint that, group member u_1 need to visit POI type c_1. Here, $n = 4$, $m = 4$, and a group member can visit any number of POI types between 0 to m.

For trip scheduling, we follow the similar steps that has been described in Section 4.3.4.1 for GTS queries without any type of user imposed constraints. We define $(m + 1)$, i.e., 5 tables, ν_0, ν_1, ν_2, ν_3 and ν_4 to store the computed trip overhead distances and combined trip overhead distances of the group members. The single member columns of dynamic table ν_0 stores the distance from source to destination via no POI instead of storing the overhead distance for the corresponding columns's group member. As the user group have impose some constraints among users and POIs, all combinations among users and POI types of dynamic tables will not valid. Tables 4.32(a-e) show ν_0, ν_1, ν_2, ν_3 and ν_4 for the considered example. For having constraint, the invalid cells are crossed out in the dynamic tables.

Computing single member columns: In the dynamic tables, each cell of the single member columns u_1, u_2, u_3 and u_4 of a table stores the minimum trip overhead distance for the corresponding column's user passing through the POI types of the corresponding row of that table. For example, in Table 4.32(c), cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$ contains the minimum trip overhead distance for user u_1 passing through POI types c_1 and c_2. For computing this trip overhead distance, we consider user, u_1's source

Table 4.32: Dynamic tables for GTS queries with dependency between user u_1 and POI type c_1 for aggregate function SUM

(a) Dynamic table ν_0

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\emptyset$	~~28.75~~	25.00	47.55	77.48	~~0.0~~	~~0.0~~

(b) Dynamic table ν_1

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1\}$	105.36	~~76.55~~	~~57.61~~	~~7.32~~	105.36	105.36
$\{c_2\}$	~~49.05~~	23.99	7.03	4.72	~~23.99~~	~~7.03~~
$\{c_3\}$	~~105.32~~	67.80	41.85	5.42	~~67.80~~	~~41.85~~
$\{c_4\}$	~~102.51~~	70.29	42.02	5.65	~~70.29~~	~~42.02~~

(c) Dynamic table ν_2

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1, c_2\}$	105.37	~~76.58~~	~~57.61~~	~~7.36~~	105.37	105.37
$\{c_1, c_3\}$	105.41	~~78.54~~	~~57.98~~	~~7.32~~	105.41	105.41
$\{c_1, c_4\}$	106.98	~~81.84~~	~~58.61~~	~~7.34~~	106.98	106.98
$\{c_2, c_3\}$	~~105.34~~	69.92	41.86	5.46	~~69.92~~	~~41.86~~
$\{c_2, c_4\}$	~~106.19~~	72.64	43.19	5.67	~~72.64~~	~~43.19~~
$\{c_3, c_4\}$	~~107.01~~	73.28	43.62	5.83	~~73.28~~	~~43.62~~

(d) Dynamic table ν_3

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1, c_2, c_3\}$	105.41	~~78.54~~	~~57.98~~	~~7.36~~	105.41	105.41
$\{c_1, c_2, c_4\}$	107.06	~~81.84~~	~~58.62~~	~~7.36~~	107.06	107.06
$\{c_1, c_3, c_4\}$	107.03	~~81.84~~	~~58.61~~	~~7.34~~	107.03	107.03
$\{c_2, c_3, c_4\}$	~~107.14~~	73.52	43.93	5.86	~~73.52~~	~~43.93~~

(e) Dynamic table ν_4

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$	$\{u_1u_2u_3u_4\}$
$\{c_1, c_2, c_3, c_4\}$	107.15	~~81.84~~	~~58.62~~	~~7.36~~	107.15	107.15	107.15

(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 overhead distances are listed in Table 4.33. Among the candidate trips listed in this table, the minimum trip overhead distance is 105.37 for the trip $s_1 \to p_2^1 \to p_1^1 \to d_1$ and this value is stored in the cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$. Similarly, our dynamic programming technique populates all the cells of the single member columns of dynamic tables ν_1, ν_2, ν_3 and ν_4. As we have already mentioned that, table ν_0 is a trivial one that stores trip distances for particular user's trip from her source to destination location only (single member columns) and trip overhead distances (combined member columns).

Table 4.33: Candidate trips with trip overhead distances for cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$

Candidate trips	**Trip distances**
$s_1 \to p_2^1 \to p_1^1 \to d_1$	105.37
$s_1 \to p_2^1 \to p_1^2 \to d_1$	109.89
$s_1 \to p_1^1 \to p_2^1 \to d_1$	106.62
$s_1 \to p_1^2 \to p_2^1 \to d_1$	126.58

Computing combined member columns: Using the valid single member columns and already calculated valid 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. Candidate combinations with trip overhead distances for cell $\nu_0[\emptyset][\{u_1u_2\}]$, $\nu_1[\{c_1\}][\{u_1u_2\}]$, $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$, $\nu_3[\{c_1, c_2, c_3\}][\{u_1u_2\}]$ and $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1u_2\}]$ are listed in Table 4.34, Table 4.35, Table 4.36, Table 4.37 and Table 4.38, respectively. We have used the similar example that we have used in Section 4.3.4.1 and have crossed out the invalid combinations for each table so that we can understand the main differences for having the constraint. Note that for having user and POI type dependency, the cell $\nu_0[\emptyset][\{u_1u_2\}]$ in table ν_0 become invalid because group member u_1 have to visit POI at least one POI type which is POI type c_1. It is not possible that both group member u_1 and u_2 will combinedly visit no POI types. Thus we do not need to calculate this cell anymore. That's why all the combination of Table 4.34 has been crossed out because those combinations are invalid as well.

As we have already mentioned that, to compute the actual trip distance for any trip from it's trip overhead distances, we use the trip distances that are stored in table ν_0 which are actually the distance between source s_i and destination d_i via no POIs. As the cell for u_1 has been crossed out in table ν_0, we have to calculate the distance between s_1 and d_1 while needed.

Table 4.34: Candidate combined combinations with trip overhead distances for cell $\nu_0[\emptyset][\{u_1u_2\}]$.

Combined combinations	Distances	Trip overhead
~~$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$~~	~~$(28.75 - 28.75) + (25.00 - 25.00)$~~	~~0.00~~

Table 4.35: Candidate combined combinations with trip overhead distances for cell $\nu_1[\{c_1\}][\{u_1u_2\}]$.

Combined combinations	Distances	Trip overhead
$\nu_1[\{c_1\}][\{u_1\}] + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$	$105.36 + (25.00 - 25.00)$	105.36
~~$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + \nu_1[\{c_1\}][\{u_2\}]$~~	~~$(28.75 - 28.75) + 76.55$~~	~~76.55~~

Table 4.36: Candidate combined combinations with trip overhead distances for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$.

Combined Combinations	Distances	Trip overhead
$\nu_2[\{c_1, c_2\}][\{u_1\}] + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$	$105.37 + (25.00 - 25.00)$	105.37
$\nu_1[\{c_1\}][\{u_1\}] + \nu_1[\{c_2\}][\{u_2\}]$	$105.36 + 23.99$	129.35
~~$\nu_1[\{c_2\}][\{u_1\}] + \nu_1[\{c_1\}][\{u_2\}]$~~	~~$49.05 + 76.55$~~	~~125.60~~
~~$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + \nu_2[\{c_1, c_2\}][\{u_2\}]$~~	~~$(28.75 - 28.75) + 76.58$~~	~~76.58~~

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 overhead 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. Along with satisfying all constraints of GTS queries, these trips also satisfies user provided dependencies among user u_1 and POI type c_1. The cell contains the minimum total trip overhead distance 107.15 of the group which is *AggTripOvDist* that we have mentioned in Section 4.3.2. To get the values of T_{min_i} and T_{max_i} for each user u_i, for GTS queries, we simply take the minimum trip distance (from table ν_0) and trip overhead values (from table ν_m) from Table 4.32(a) and Table 4.32(e), respectively and add the distance from source s_i to destination d_i with the trip overhead distances to get the actual trip distance for user u_i. Note that,

Table 4.37: Candidate combined combinations with trip overhead distances for cell $\nu_3[\{c_1, c_2, c_3\}][\{u_1 u_2\}]$

Combined combinations	**Distances**	**Trip overhead**
$\nu_3[\{c_1, c_2, c_3\}][\{u_1\}] + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$	$105.41 + (25.00 - 25.00)$	105.41
$\nu_2[\{c_1, c_2\}][\{u_1\}] + \nu_1[\{c_3\}][\{u_2\}]$	$105.37 + 67.80$	173.17
$\nu_2[\{c_1, c_3\}][\{u_1\}] + \nu_1[\{c_2\}][\{u_2\}]$	$105.41 + 23.99$	129.40
~~$\nu_2[\{c_2, c_3\}][\{u_1\}] + \nu_1[\{c_1\}][\{u_2\}]$~~	~~$105.34 + 76.55$~~	~~181.89~~
$\nu_1[\{c_1\}][\{u_1\}] + \nu_2[\{c_2, c_3\}][\{u_2\}]$	$105.36 + 69.92$	175.28
~~$\nu_1[\{c_2\}][\{u_1\}] + \nu_2[\{c_1, c_3\}][\{u_2\}]$~~	~~$49.05 + 78.54$~~	~~127.59~~
~~$\nu_1[\{c_3\}][\{u_1\}] + \nu_2[\{c_1, c_2\}][\{u_2\}]$~~	~~$105.32 + 76.58$~~	~~181.90~~
~~$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + \nu_3[\{c_1, c_2, c_3\}][\{u_2\}]$~~	~~$(28.75 - 28.75) + 78.54$~~	~~78.54~~

Table 4.38: Candidate combined combinations with trip overhead distances for cell $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1 u_2\}]$

Combined Combinations	**Distances**	**Trip overhead**
$\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1\}] + (\nu_0[\emptyset][\{u_2\}] - \nu_0[\emptyset][\{u_2\}])$	$107.15 + (25.00 - 25.00)$	107.15
$\nu_3[\{c_1, c_2, c_3\}][\{u_1\}] + \nu_1[\{c_4\}][\{u_2\}]$	$105.41 + 70.29$	175.70
$\nu_3[\{c_1, c_2, c_4\}][\{u_1\}] + \nu_1[\{c_3\}][\{u_2\}]$	$107.06 + 67.80$	174.86
$\nu_3[\{c_1, c_3, c_4\}][\{u_1\}] + \nu_1[\{c_2\}][\{u_2\}]$	$107.03 + 23.99$	131.02
~~$\nu_3[\{c_2, c_3, c_4\}][\{u_1\}] + \nu_1[\{c_1\}][\{u_2\}]$~~	~~$107.14 + 76.55$~~	~~183.69~~
$\nu_2[\{c_1, c_2\}][\{u_1\}] + \nu_2[\{c_3, c_4\}][\{u_2\}]$	$105.37 + 73.28$	178.65
$\nu_2[\{c_1, c_3\}][\{u_1\}] + \nu_2[\{c_2, c_4\}][\{u_2\}]$	$105.41 + 72.64$	178.05
$\nu_2[\{c_1, c_4\}][\{u_1\}] + \nu_2[\{c_2, c_3\}][\{u_2\}]$	$106.98 + 69.92$	176.90
~~$\nu_2[\{c_2, c_3\}][\{u_1\}] + \nu_2[\{c_1, c_3\}][\{u_2\}]$~~	~~$105.34 + 81.84)$~~	~~187.18~~
~~$\nu_2[\{c_2, c_4\}][\{u_1\}] + \nu_2[\{c_1, c_3\}][\{u_2\}]$~~	~~$106.19 + 78.54$~~	~~184.73~~
~~$\nu_2[\{c_3, c_4\}][\{u_1\}] + \nu_2[\{c_1, c_2\}][\{u_2\}]$~~	~~$107.01 + 76.58$~~	~~183.59~~
$\nu_1[\{c_1\}][\{u_1\}] + \nu_3[\{c_2, c_3, c_4\}][\{u_2\}]$	$105.36 + 73.52$	178.88
~~$\nu_1[\{c_2\}][\{u_1\}] + \nu_3[\{c_1, c_3, c_4\}][\{u_2\}]$~~	~~$49.05 + 81.84$~~	~~130.89~~
~~$\nu_1[\{c_3\}][\{u_1\}] + \nu_3[\{c_1, c_2, c_4\}][\{u_2\}]$~~	~~$105.32 + 81.84$~~	~~187.16~~
~~$\nu_1[\{c_4\}][\{u_1\}] + \nu_3[\{c_1, c_2, c_3\}][\{u_2\}]$~~	~~$102.51 + 78.54$~~	~~181.05~~
~~$(\nu_0[\emptyset][\{u_1\}] - \nu_0[\emptyset][\{u_1\}]) + \nu_4[\{c_1, c_2, c_3, c_4\}][\{u_2\}]$~~	~~$(28.75 - 28.75) + 81.84$~~	~~81.84~~

for having constraint, user u_1 have to visit at least POI type c_1, cell of table Table 4.32(b) contains the T_{min_i} value for user u_1. Similarly, it is not valid that other users u_2, u_3 and u_4 should visit all required POI types including c_1. So for other users instead of user u_1, we can take the maximum overhead value of the valid combinations from Table 4.32(d) and add the distance from source to destination location for corresponding user for T_{max_i} values. Thus, the T_{min_i} and T_{max_i} values for users $\{u_1, u_2, u_3, u_4\}$ are $\{(105.36 + 28.75), 25.00, 47.55, 77.48\} \equiv \{134.11, 25.00, 47.55, 77.48\}$ and $\{(107.15 + 28.75), (73.52 + 25.00), (43.93 + 47.55), (5.86 + 77.48)\} \equiv \{135.90, 98.52, 91.48, 83.34\}$, respectively. Using these values we refine the search region based on Theorems 4.3.1 and 4.3.2. For user u_1, based on Theorem 4.3.1, the major axis for the elliptic region E_1 is 135.90. On the other hand, based on Theorem 4.3.2, the major axis is $107.15 + 134.11 = 241.26$. We take the best bound among them which is 135.90, the first one.

Similar to all other variations of GTS queries, 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 overhead distance is obtained with satisfying the constraint. For the sake of clarity we do not show them in the tables.

Chapter 5

Algorithms

In this chapter we present algorithms for GTS and UGTS queries based on our solution described at Chapter 4 and discuss, how we can extend the algorithms for GTS and UGTS queries with different types of constraints.

The organization of this chapter is as follows. We present and elaborate the algorithms for GTS and UGTS queries in Sections 5.1 and 5.2, respectively. In Section 5.3, we discuss ways to extend our proposed algorithms for processing GTS and UGTS queries for having different types of constraints (e.g. dependencies among POIs, dependencies among a user and POIs).

5.1 GTS Approach

The key idea of our algorithm is to incrementally retrieve nearest POIs with respect to the geometric centroid G of all users' source and destination locations. Our algorithm uses best first search (BFS) to incrementally retrieve POIs from the data storage. We assume that, POIs are indexed using an R^*-tree in the database. Our algorithm retrieves POIs until they minimize the aggregate trip overhead distance from the user's source to destination via the required POI types.

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

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

```
input  : $S, D, \mathbb{C}, f$
output: A set of trips, $T$
$Initialize()$;
$InitDynTables(|S|, |\mathbb{C}|, \mathcal{V})$;
$ComputeTable(\nu_0, f)$;
$Enqueue(Q_p, root, MinD(G, root))$;
while $Q_p$ is not empty do
    if $end = 1$ then
        break;
    $\{p, d_{min}(p)\} \leftarrow Dequeue(Q_p)$;
    $r \leftarrow d_{min}(p)$;
    if $p$ is not a POI then
        foreach child node $p_c$ of $p$ do
            $Enqueue(Q_p, p_c, MinD(G, p_c))$;
    else if $\tau(p) \in \mathbb{C}$ and $p \in \bigcup_{i=1}^{n} E_i$ then
        $P \leftarrow InsertPOI(p)$;
        if $init = 0$ and $CheckInclude(P, \mathbb{C})$ then
            $ComputeTrip(S, D, \mathbb{C}, P, \mathcal{V})$;
            $init \leftarrow 1$;
            $isup \leftarrow true$;
        else if $init = 1$ then
            $isup \leftarrow UpdateTrip(\tau(p), S, D, \mathbb{C}, p, \mathcal{V})$;
    if $isup = true$ and $init = 1$ then
        $\{T, Mx, Mi\} \leftarrow UpDynTables(|S|, \mathbb{C}, \mathcal{V}, f)$;
        $ellipregions \leftarrow UpEllipticRegions(T, Mx, Mi, f)$;
    if $IsInCircle(G, r, ellipregions)$ then
        $end \leftarrow 1$;
return $T$
```

for a group of n members and the set of required m POI types $\mathbb{C}$ and aggregation function f which may be either SUM or MAX 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 $ComputeTable(\nu_0, f)$ computes the values for single member columns and combined member columns of the first dynamic table ν_0. The calculation of combined member columns differs based on the aggregate function f. If $f =$ SUM, the combined member columns store the total value of the corresponding column's multiple users' trip overhead distances. Otherwise, if $f =$ MAX, the combined member columns store the minimum value of maximum trip overhead distances of the corresponding column's multiple users' trips. The stored trip distances and trip overhead distances in ν_0 are Euclidean distances if the GTS 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 tables ν_1 to ν_m with trip overhead distances that our dynamic programming technique uses. The algorithm sets *init* to 1 and *isup* to *true*. As mentioned before, the stored trip distances or overhead 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 overhead 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 $UpDynTables(|S|, \mathbb{C}, \mathcal{V}, f)$ updates combined member columns of tables from ν_1 to ν_m based on the logic described in Section 4.3.4. The function takes n, m, the set of all dynamic tables $\mathcal{V}$ and the aggregate function f 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 4.3.2 for both aggregate function SUM and MAX. As we already mentioned, based on the aggregate function f, the calculation of combined member columns of the dynamic table differs. If $f =$ SUM, the combined member columns of the dynamic tables store the total value of the corresponding column's multiple users' trip overhead distances. Otherwise the combined member columns stores the minimum value of maximum trip overhead distances of the corresponding column's multiple users' trips, if $f =$ MAX. Algorithm 2 shows the pseudocode of the function $UpDynTables(|S|, \mathbb{C}, \mathcal{V}, f)$ which we will explain shortly.

Then using function *UpEllipticRegions*(T, Mx, Mi, f), the algorithm updates the elliptic bound for all n users, where *ellipregions* represents the elliptic search regions $\{E_i, E_2, \ldots, E_n\}$ of the users. The bounds for the elliptic search regions are determined using both Theorem 4.3.1 and 4.3.2 for aggregate function f = SUM. For aggregate function f = MAX, to determine the bounds for the elliptic search regions our algorithm uses both Theorem 4.3.1 and 4.3.3. In Algorithm 3, we show the pseudocode of the function *UpEllipticRegions*(T, Mx, Mi, f) which we will explain shortly.

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 aggregate trip overhead distance.

Now we will elaborately explain pseudocode of two important functions $UpDynTables(n, \mathbb{C}, \mathcal{V}, f)$ and *UpEllipticRegions*(T, Mx, Mi, f) that we have used in Algorithm 1.

Algorithm 2 shows the pseudocode of the function $UpDynTables(n, \mathbb{C}, \mathcal{V}, f)$ which updates combined member columns of the dynamic tables from ν_1 to ν_m based on the logic described in Section 4.3.4. The function takes number of group members n, the set of required POI types $\mathbb{C}$, the set of all dynamic tables $\mathcal{V}$ and the aggregate function f 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 4.3.2 for both aggregate function SUM and MAX. The function uses y variable to keep track of the current dynamic table to update the combined member columns. Variable *maxcol* stores the number of maximum combined member columns of the current dynamic table ν_y. As we already mentioned that the final dynamic table ν_m has one more extra column than the other dynamic tables and using *maxcol* variable we keep track of the column count.

For each row of current dynamic table ν_y, the function update all the cells of combined member columns of that row one by one. The algorithm uses variable i to keep track of the current combined member column which is going to be updated. To compute the cell of current combined member column $\{u_1 \ldots u_i\}$, the algorithm uses single member column $\{u_i\}$ and already computed sin-

Algorithm 2: *UpDynTables*$(n, \mathbb{C}, \mathcal{V}, f)$

```
input : $n, \mathbb{C}, \mathcal{V}, f$
output: $T$,$Mx$,$Mi$
for $y \leftarrow 1$ to $|\mathbb{C}|$ do
    if $y = |\mathbb{C}|$ then
        $maxcol = n$;
    else
        $maxcol = n - 1$;
    foreach member $p_c$ of ${}^{\mathbb{C}}C_y$ do
        for $i \leftarrow 2$ to $maxcol$ do
            $fcol = \{u_1 \dots u_{(i-1)}\}$; $scol = \{u_i\}$; $dist_{min} \leftarrow 0$;
            for $g \leftarrow 0$ to $y$ do
                foreach member $q_c$ of ${}^{\mathbb{C}}C_g$ do
                    $d_t \leftarrow 0$;
                    if $q_c \subseteq p_c$ then
                        $p_{df} \leftarrow p_c - q_c$;
                        if $f = \textsc{sum}$ then
                            if $g = 0$ then
                                $d_t \leftarrow ((\nu_g[q_c][fcol] - \nu_0[\emptyset][fcol]) + \nu_{|p_{df}|}[p_{df}][scol])$;
                            else if $g = y$ then
                                $d_t \leftarrow (\nu_g[q_c][fcol] + (\nu_{|p_{df}|}[p_{df}][scol] - \nu_0[\emptyset][scol]))$;
                            else
                                $d_t \leftarrow (\nu_g[q_c][fcol] + \nu_{|p_{df}|}[p_{df}][scol])$;
                        else if $f = \textsc{max}$ then
                            if $g = 0$ then
                                $d_t \leftarrow \max((\nu_g[q_c][fcol] - \nu_0[\emptyset][fcol]), \nu_{|p_{df}|}[p_{df}][scol])$;
                            else if $g = y$ then
                                $d_t \leftarrow \max(\nu_g[q_c][fcol], (\nu_{|p_{df}|}[p_{df}][scol] - \nu_0[\emptyset][scol]))$;
                            else
                                $d_t \leftarrow \max(\nu_g[q_c][fcol], \nu_{|p_{df}|}[p_{df}][scol])$;
                    if $d_t < dist_{min}$ then
                        $dist_{min} \leftarrow d_t$;
            $\nu_y[p_c][\{u_1 \dots u_i\}] \leftarrow dist_{min}$;
```

gle/combined member column $\{u_1 \dots u_{(i-1)}\}$, upto the previous group member u_{i-1}. The algorithm uses $fcol$ and $scol$ variables to keep track of the columns that are required to compute the cell of current combined member column $\{u_1 \dots u_i\}$. Note that, $fcol$ can be both single (e.g. $\{u_1\}$) and combined member columns where $scol$ can be only single member column. Using variable $dist_{min}$ the algorithm computes the minimum aggregate trip overhead distances among all candidate trip overhead distances of the possible combinations of any cell.

To compute the combined member columns of current dynamic table ν_y, the algorithm uses all the dynamic tables from ν_0 to ν_y and to keep track of that the algorithm uses variable g. Each time we compare the POI type combinations of different dynamic tables to find out the candidate combinations and pick the minimum one which gives the minimum trip overhead distance of that user and POI type combination.

Note that, based on the aggregate function f, the calculation of combined member columns of the dynamic table differs. If $f = \textsc{sum}$, the combined member columns of the dynamic tables store the total value of the corresponding column's multiple users' trip overhead distances. Otherwise the combined member columns stores the minimum value of maximum trip overhead distances of the corresponding column's multiple users' trips, if $f = \textsc{max}$. Note that, the cells of single member columns of table ν_0 store the trip distances instead of trip overhead distances where the combined member columns of table ν_0 and the single and combined member columns of all other tables stores the trip overhead distances of the combined group member. So for the single member columns of table ν_0, we deduct the distance between the source and destination locations of a group member from the trip distance to get the trip overhead distance while updating the cells of combined member columns.

Algorithm 3 represents the pseudocode of function $UpEllipticRegions(T, Mx, Mi, f)$ which is uses to update the elliptic bounds for all n users elliptic search regions $\{E_1, E_2, \dots, E_n\}$ using the search region refinement techniques that has been described in Section 4.3.2. It takes T, Mx, Mi and f as input and updates the updates the elliptic bound of each group members elliptic search regions. We have already mentioned that, T represents the set of n scheduled trips $\{T_1, T_2, \dots, T_n\}$, Mx and Mi represents the minimum and maximum trip distances, $\{T_{max_1}, \dots, T_{max_n}\}$ and $\{T_{min_1}, \dots, T_{min_n}\}$, respectively. In Section 4.3.2 for both aggregate function SUM and MAX, T_{max_i} and T_{min_i} for $1 \leq i \leq n$ are defined. At first step, the function computes the aggregate trip overhead distance of the group members using T and Mi. Here *AggOverheadDist* stores the aggregate trip overhead distance for a group member. At first step of the algorithm, *AggOverheadDist* initializes to 0.0. After that, the algorithm computes the aggregate trip overhead distance for all n group members and stores the value to *AggOverheadDist*. $TripDist_i$ represents the trip distance of trip T_i When f = SUM, it stores the total trip overhead distances of the n trips. For both aggregate function SUM and MAX, T_{max_i} be one bound $Bound_1$ using Theorem 4.3.1. Using Theorem 4.3.2 for aggregate function f = SUM and Theorem 4.3.3 for aggregate function f = MAX another bound $Bound_2$ is $AggOverheadDist + T_{min_i}$. Finally the algorithm chooses and updates the major axis of the elliptic search region with the best bound which gives the smaller bound between $Bound_1$ and $Bound_2$.

Algorithm 3: *UpEllipticRegions*(T, Mx, Mi, f)

```
input : $T, Mx, Mi, f$

$AggOverheadDist \leftarrow 0.0$;
for $i \leftarrow 1$ to $n$ do
    if $f$ = SUM then
        $AggOverheadDist \leftarrow AggOverheadDist + (TripDist_i - T_{min_i})$;
    else if $f$ = MAX then
        $AggOverheadDist \leftarrow \max(AggOverheadDist, (TripDist_i - T_{min_i}))$;
for $i \leftarrow 1$ to $n$ do
    $Bound_1 \leftarrow T_{max_i}$;
    $Bound_2 \leftarrow AggOverheadDist + T_{min_i}$;
    if $Bound_1 < Bound_2$ then
        $E_i.MajorAxis \leftarrow Bound_1$;
    else
        $E_i.MajorAxis \leftarrow Bound_2$;
```

5.2 UGTS Approach

Algorithm 4: *UGTS_Approach*$(S, D, \mathbb{C}, e, f)$

```
input  : $S, D, \mathbb{C}, e, f$
output: A set of trips, $T$
$Initialize()$;
$InitDynTablesUniform(e, |S|, |\mathbb{C}|, \mathcal{V})$;
$Enqueue(Q_p, root, MinD(G, root))$;
while $Q_p$ is not empty do
    if $end = 1$ then
        break;
    $\{p, d_{min}(p)\} \leftarrow Dequeue(Q_p)$;
    $r \leftarrow d_{min}(p)$;
    if $p$ is not a POI then
        foreach child node $p_c$ of $p$ do
            $Enqueue(Q_p, p_c, MinD(G, p_c))$;
    else if $\tau(p) \in \mathbb{C}$ and $p \in \bigcup_{i=1}^{n} E_i$ then
        $P \leftarrow InsertPOI(p)$;
        if $init = 0$ and $CheckInclude(P, \mathbb{C})$ then
            $ComputeTrip(S, D, \mathbb{C}, e, P, \mathcal{V})$;
            $init \leftarrow 1$;
            $isup \leftarrow true$;
        else if $init = 1$ then
            $isup \leftarrow UpdateTrip(\tau(p), S, D, \mathbb{C}, e, p, \mathcal{V})$;
    if $isup = true$ and $init = 1$ then
        $\{T, Mx, Mi\} \leftarrow UpDynTablesUniform(e, |S|, \mathbb{C}, \mathcal{V}, f)$;
        $ellipregions \leftarrow UpEllipticRegions(T, Mx, Mi, f)$;
    if $IsInCircle(G, r, ellipregions)$ then
        $end \leftarrow 1$;
return $T$
```

Algorithm 4 shows the pseudocode of the proposed approach to evaluate Uniform GTS (UGTS) queries in spatial databases. It takes the set of source and destination locations, S and D, respectively for a group of n members, the set of required m POI types $\mathbb{C}$, the number of POI types e that each member visits and aggregation function f which may be either SUM or MAX as input. The output is the set of n scheduled trips $T = \{T_1, T_2, \ldots, T_n\}$ where the n trips combinedly visit all POI types in $\mathbb{C}$.

As first step, Algorithm 4 initializes all required variables that are needed throughout the UGTS query processing. It does this initialization using function $Initialize()$ where the function declares and initializes geometric centroid G of all n source-destination pairs, data structure for traversing R^*-tree, priority queue Q_p, $ellipregions = \{E_1, E_2, \ldots, E_n\}$ for n users where each ellipse E_i has foci at user's source and destination location and major axis is equal to ∞ and initializes the required variables $end = 0$, $init = 0$, $r = 0$, $isup = false$. The variable end represents the algorithm termination indicator and $init$ represents a variable flag that is used to keep track between compute and update trips operations. The variables r and $isup$ represent the radius of current known region and a flag that indicates any user trips is updated or not respectively.

Function $InitDynTablesUniform(e, |S|, |\mathbb{C}|, \mathcal{V})$ initializes the dynamic tables $\mathcal{V} = \{\nu_e, \nu_{2e}, \ldots \nu_m\}$ which we have mentioned in Section 4.3.4. For the dynamic table ν_e, the function also initializes Mx and Mi, where Mx and Mi represent the set of n maximum $\{T_{max_1}, \ldots, T_{max_n}\}$ and minimum $\{T_{min_1}, \ldots, T_{min_n}\}$ bounds, which has been mentioned in Section 4.3.2. The function initializes Mx by ∞ and Mi by $Dist(s_i, d_i)$ for any group member u_i.

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. The priority queue Q_p stores elements in order of their distance from G which determined by function $MinD(G, p)$. The function $MinD(G, p)$ calculates and returns the minimum Euclidean distance between geometric centroid, G and p, where p represents a POI or a minimum bounding rectangle of a R^*-tree node for both Euclidean space and road networks. After that the algorithm removes an element p from the priority queue Q_p along with $d_{min}(p)$ which represents the minimum distance of p computed from the query point G. 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 if they might contain any candidate answer set. On the other hand, if p is a POI then it is added to candidate POI set P with tracking of the POI type of this POI. The algorithm uses function $\tau(p)$ to determine the POI type of a POI p. Before adding POI p to candidate POI set P, the algorithm also checks if the POI p lies inside any user's elliptic search region or not too.

Function $CheckInclude(P, \mathbb{C})$ checks if the POI set P contains at least one POI from each POI types in $\mathbb{C}$. When initial POI set has been found, the algorithm initially compute trips for all possible cases for each users using function $ComputeTrip(S, D, \mathbb{C}, e, P, \mathcal{V})$. In short, this function computes required trips and populates the single member columns of the dynamic table ν_e. The algorithm updates *isup* by *true* and *init* by 1. After computing trips, if the algorithm retrieves any new POI p, it uses function $UpdateTrip(\tau(p), S, D, \mathbb{C}, e, p, \mathcal{V})$ to update the trips in table ν_e which only can be updated with the newly retrieved POI p and updates *isup* accordingly.

Function $UpDynTablesUniform(e, n, \mathbb{C}, \mathcal{V}, f)$ updates all other tables from ν_{2e} to ν_m which we have been mentioned in Section 4.3.4. Algorithm 5 shows the pseudocode for updating the dynamic tables based on the logic we have described in Section 4.3.4. The function takes e, n, $\mathbb{C}$, $\mathcal{V}$ and f as input and returns T, Mx and Mi, where T represents the scheduled trips, Mx and Mi represent the set of n maximum $\{T_{max_1}, \ldots, T_{max_n}\}$ and minimum $\{T_{min_1}, \ldots, T_{min_n}\}$ bounds, which has been mentioned in Section 4.3.2. Function $UpdateMinMaxDist(\nu_e)$ updates Mx and Mi for all n users.

Then using function $UpEllipticRegions(T, Mx, Mi, f)$ the algorithm updates the elliptic bound for all n users where *ellipregions* represents the elliptic search region of the users. The bound for the elliptic search regions are determined using both Theorem 4.3.1 and 4.3.2 for aggregate function SUM and using both Theorem 4.3.1 and 4.3.3 for aggregate function MAX. The algorithm checks the terminating condition of our GTS queries using function $IsInCircle(G, r, ellipregions)$. This function checks if all n elliptic search regions is included by the current circular known region or not. If the terminating condition becomes true, the algorithm updates the terminating flag *end* to 1and returns the uniformly scheduled trips T for n users that provide the minimum aggregate trip overhead distance.

Now we will elaborately explain pseudocode of one of the important functions $UpDynTableUniform(e, n, \mathbb{C}, \mathcal{V}, f)$. We skip to explain the another important function $UpEllipticRegions(T, Mx, Mi, f)$ which is exactly similar which we already explained in previous Section 5.1.

Algorithm 5 shows the pseudocode of the function $UpDynTableUniform(e, n, \mathbb{C}, \mathcal{V}, f)$ which updates combined member columns of the dynamic tables from ν_{2e}, ν_{3e} to ν_m based on the logic described in Section 4.3.4. The function takes number of POI types that each group member visits e, the number of group members n, the set of required POI types $\mathbb{C}$, the set of all dynamic tables $\mathcal{V}$ and the aggregate function f 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 4.3.2 for both aggregate function SUM and MAX.

Initially the function updates Mx and Mi using the function $UpdateMinMaxDist(\nu_e)$ which take the dynamic table ν_e as input and updates Mx and Mi for the group members. After that function uses i variable to keep track of the current dynamic table (e.g. ν_{ie}) to update the only combined member column. For each row of current dynamic table ν_{ie}, we update all the cells of the combined member column. To compute the cells of current combined member column $\{u_1 \ldots u_i\}$ of dynamic table ν_{ie}, the algorithm uses single member column $\{u_i\}$ of table ν_e and already computed combined member column $\{u_1 \ldots u_{(i-1)}\}$ of dynamic table $\nu_{(i-1)e}$. The algorithm uses $fcol$ and $scol$ variables to keep track of the columns that are required to compute the cell of current combined member column $\{u_1 \ldots u_i\}$. Note that, $fcol$ can be both single (e.g. $\{u_1\}$) and combined member column where $scol$ can be only single member column. Using variable $dist_{min}$ the algorithm computes the minimum aggregate trip overhead distances among all candidate trip overhead distances of the possible combinations of any cell.

Each time we compare the POI type combinations of different dynamic tables to find out the candidate combinations and pick the minimum one which gives the minimum trip overhead distance of that user and POI type combination. Note that, based on the aggregate function f, the calculation of combined member columns of the dynamic table differs. If f = SUM, the combined member columns of the dynamic tables store the total value of the corresponding column's multiple users' trip overhead distances. Otherwise the combined member columns stores the minimum value of maximum trip overhead distances of the corresponding column's multiple users' trips, if f = MAX. Note that,

Algorithm 5: $UpDynTablesUniform(e, n, \mathbb{C}, \mathcal{V}, f)$

```
input  : $e, n, \mathbb{C}, \mathcal{V}, f$
output: $T$,$Mx$,$Mi$
$UpdateMinMaxDist(\nu_e)$;
for $i \leftarrow 2$ to $n$ do
    $fcol = \{u_1 \dots u_{(i-1)}\}$;
    $scol = \{u_i\}$;
    foreach member $p_c$ of $\{^{\mathbb{C}}C_{ie}\}$ do
        $dist_{min} \leftarrow 0$;
        foreach member $q_c$ of $\{^{\mathbb{C}}C_{(i-1)e}\}$ do
            $d_t \leftarrow 0$;
            if $q_c \subset p_c$ then
                $p_{df} \leftarrow p_c - q_c$;
                if $f =$ SUM then
                    if $fcol = \{u_1\}$ then
                        $d_t \leftarrow (\nu_e[q_c][fcol] - T_{min_1}) + (\nu_e[p_{df}][scol] - T_{min_i})$;
                    else
                        $d_t \leftarrow (\nu_{(i-1)e}[q_c][fcol] + (\nu_e[p_{df}][scol] - T_{min_i}))$;
                else if $f =$ MAX then
                    if $fcol = \{u_1\}$ then
                        $d_t \leftarrow \max((\nu_e[q_c][fcol] - T_{min_1}), (\nu_e[p_{df}][scol] - T_{min_i}))$;
                    else
                        $d_t \leftarrow \max(\nu_{(i-1)e}[q_c][fcol], (\nu_e[p_{df}][scol] - T_{min_i}))$;
            if $d_t \leq dist_{min}$ then
                $dist_{min} \leftarrow d_t$;
        $\nu_{ie}[\{p_c\}][\{u_1 \dots u_i\}] \leftarrow dist_{min}$;
return $T$,$Mx$,$Mi$
```

the cells of single member columns store the trip distances instead of trip overhead distances where the combined member columns stores the trip overhead distances of the combined group members. So for the single member columns, we deduct the distance between the source and destination locations of a group member from the trip distance to get the trip overhead distance while updating the cells of combined member columns.

5.3 Extensions

A group may impose different types of constraints like dependencies among POIs, dependencies among members and POIs in both GTS and UGTS queries. We can extend our proposed algorithms that we have described in Section 5.1 for GTS queries and in Section 5.2 for UGTS queries for GTS/UGTS queries having different types of constraints as well.

For having different types of constraints, some combinations of POI types or some combinations of members and POI types become invalid which we have mentioned in Chapter 4. These invalid POIs combinations or members and POIs combinations should be ignored if we need to schedule trips with constrains using our proposed approach. In our approach when we are initializing dynamic tables in both Algorithm 1 and Algorithm 4 using function $InitDynTables(|S|, |\mathbb{C}|, \mathcal{V})$ and $InitDynTablesUniform(e, |S|, |\mathbb{C}|, \mathcal{V})$, respectively, we can check the validity of the combination based on imposed constraints. In both algorithms, while we are computing and updating trips we have to check the validity of the POI types combinations or user and POI types combinations. For the invalid combinations, we do not need to perform compute or update trip operations. We also have to check validity while we are updating combined user columns of the dynamic tables. The invalid combinations have to deduce while we are computing aggregate trip overhead distances for each cells of the dynamic tables.

For the GTS queries with different types of constraints, we need to choose Mx and Mi bounds for different user's considering the validity of the POI type combinations or user and POI types combinations. For example, in GTS queries, we take $\nu_0[\emptyset][\{u_i\}]$ value which is the trip distance for user u_i without visiting any POI types as minimum bound for user u_i. Suppose in GTS queries with dependencies among user and POIs, user imposes constraint that user u_i should visit POI type c_j. In that case the cell $\nu_0[\{u_i\}]$ of table ν_0 will be invalid. It will not happen that user u_i will not visit any POI types. So we should take the value of the trip T_i of user u_i where the trip visits only POI type c_j as minimum bound of that user. In summary, we should take the value of the cell $\nu_1[\{c_j\}][\{u_i\}]$ of dynamic table ν_1 as the minimum bound for user u_i.

Chapter 6

A Straightforward Approach

To the best of our knowledge, we introduce GTS queries and its variant Uniform GTS (UGTS) in spatial databases and thus, there exists no approach to process GTS or UGTS queries in the literature. To validate the efficiency of our proposed approach in experiments, using existing trip planning algorithms, we develop straightforward approaches for processing GTS queries and UGTS queries, S-GTS and S-UGTS, respectively.

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

We organize this chapter as follows. The algorithms for processing S-GTS and S-UGTS queries have been presented and elaborately discussed in Sections 6.1 and 6.2, respectively. In Section 6.3, we discuss ways to extend our proposed algorithms for processing S-GTS and S-UGTS queries for having different types of constraints.

6.1 Algorithm for S-GTS Approach

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

Algorithm 6 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 6 initializes the dynamic tables ν_0 to ν_m using the function $InitDynTables(|S|, |\mathbb{C}|, \mathcal{V})$, which we mentioned in Section 4.3.4. After that $ComputeTable(\nu_0, f)$ computes single member columns and combined member columns of the first dynamic table ν_0 according to the aggregate function f. 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 with computed trip overhead distances which are computed by reducing the distance from source (s_i) to destination (d_i) for a group member u_i from the trip distances. 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 [3] 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 $UpDynTables(n, m, \mathcal{V}, f)$ 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 according to aggregate function f, 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 4, 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.

6.2 Algorithm for S-UGTS Approach

Algorithm 7 shows the pseudocode of the S-UGTS approach to evaluate UGTS queries in spatial databases. As input it takes the following parameters : the set of source and destination locations, S and D, respectively, for a group of n members, the set of required m POI types $\mathbb{C}$, and the number of POI types e that each member visits. 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 $\mathbb{C}$.

```
Algorithm 7: $S\text{-}UGTS\_Approach(S, D, \mathbb{C}, e, f)$
input  : $S, D, \mathbb{C}, e, f$
output: A set of trips, $T$
$m \leftarrow |\mathbb{C}|$;
$n \leftarrow |S|$;
$InitDynTablesUniform(e, |S|, |\mathbb{C}|, \mathcal{V})$;
for group member $u_i$ do
    foreach member $t_c$ of $\{{}^{\mathbb{C}}C_e\}$ do
        $\nu_e[\{t_c\}, \{u_i\}] \leftarrow GTP(s_i, d_i, t_c)$;
$\{T, Mx, Mi\} \leftarrow UpDynTablesUniform(e, n, \mathbb{C}, \mathcal{V}, f)$;
return $T$
```

In the first step, Algorithm 7 initializes the dynamic tables $\nu_e, \nu_{2e}, \ldots, \nu_m$ using the function $InitDynTablesUniform(e, |S|, |\mathbb{C}|, \mathcal{V})$, which we mentioned in Section 4.3.4. For the dynamic table ν_e, the function also initializes Mx by ∞ and Mi by $Dist(s_i, d_i)$ for any group member u_i, where Mx and Mi represent the set of n maximum trip distances of visiting any e number of POI types.

After that for each member of the group, the algorithm calculates trips for ${}^{\mathbb{C}}C_e$ possible set of e POI types using function $GTP(s_i, d_i, t_c)$, and populates the dynamic table ν_e. The function takes the source and destination locations of u_i, and a set of e POI types from C as input and returns the optimal trip distance 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.

Finally, the algorithm uses function $UpDynTablesUniform(e, n, \mathbb{C}, \mathcal{V}, f)$ to select the final n scheduled trips for the group. The function updates tables from ν_{2e} to ν_m, which we discussed in Section 4.3.4, and returns T, Mx and Mi, where T is the n scheduled trips. Mx and Mi represent the set of n maximum $\{T_{max_1}, \ldots, T_{max_n}\}$ and minimum $\{T_{min_1}, \ldots, T_{min_n}\}$ bounds, respectively, which has no use for the S-UGTS approach.

Similar to S-GTS approach, in S-UGTS approach, any existing trip planning algorithm or group trip planning algorithm (by assuming one group member) can be used for the function $GTP(s_i, d_i, t_c)$. In our experiment, we use the most recent and efficient group trip planning algorithm [3] for this purpose.

Although for the S-UGTS approach, we also apply the similar dynamic programming that we use for our UGTS approach in Chapter 4, the two approaches are different. In the S-UGTS approach, we use the dynamic programming technique once to find the final scheduled n trips from the already calculated optimal sub trips of users. Based on n scheduled trips, we do not perform any optimization, whereas in the UGTS approach, we apply the dynamic programming technique multiple times. The UGTS approach incrementally retrieves POIs from the database, calculates the sub trips of users, and the dynamic programming technique is applied to compute n scheduled trips. Based on the computed n scheduled trips, the UGTS approach refines the search region, retrieves POIs, updates sub trips until we find the optimal n scheduled trips for UGTS queries.

6.3 Extension of Straightforward Approach for GTS and UGTS Queries with Constraints

In a GTS query or a Uniform GTS (UGTS) query, group may impose different types of constraints like dependencies among POIs, dependencies among members and POIs. For having different types of constraints, some combinations of POI types or some combinations of members and POI types become invalid which we have mentioned in Chapter 4. In straightforward approach, the invalid POIs combinations or members and POIs combinations should be ignored as well while we are computing single trips using function $GTP(s_i, d_i, t_c)$. After computing single trips for all valid combinations, we use similar dynamic programming approach for constraints which schedules multiples trips considering only valid combinations only once.

Chapter 7

Experiments

In this chapter, we evaluate the performance of our approach for processing GTS and UGTS queries through extensive experiments. Since there is no existing work for GTS or UGTS queries in the literature, we compare our proposed GTS and UGTS approaches with the straightforward approaches S-GTS and S-UGTS, respectively, that have been discussed in Chapter 6 by varying a wide range of parameters.

We evaluate our approaches in both Euclidean and road network dataspaces using synthetic and real world datasets for both aggregate functions SUM and MAX. For the real dataset, we used California [1] 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 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 and UGTS 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 and UGTS and S-UGTS approaches require the same amount of storage for storing dynamic tables, we do not show them in our experiments.

To present the experimental results, we organize this chapter as follows. In Section 7.1, we show the experimental results of GTS queries for aggregate functions SUM and MAX in both Euclidean space and road networks. The experimental results of UGTS queries for aggregate functions SUM and MAX in both Euclidean space and road networks have been shown and discussed in Section 7.2.

7.1 GTS Queries

GTS queries for both aggregate function SUM and MAX, we performed several set of experiments by varying the following parameters:

(i) the group size n
(ii) the number of specified POI types m
(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)

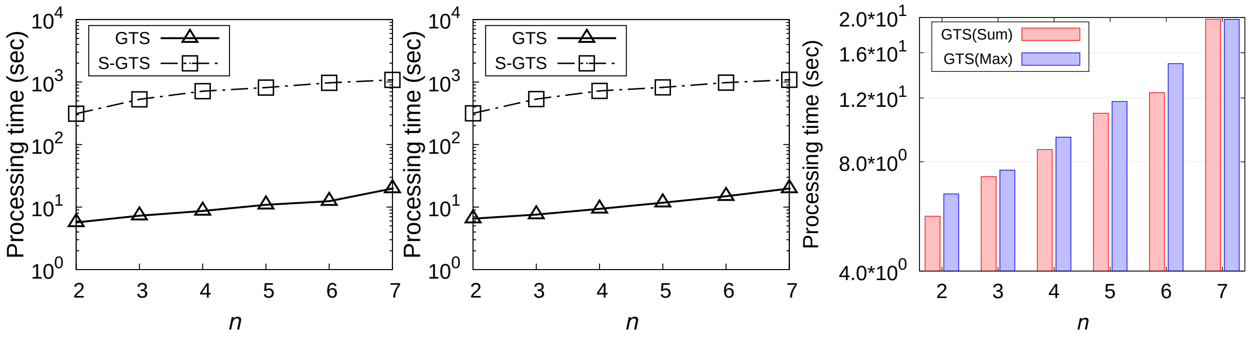
Table 7.1: Parameter settings for GTS queries

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	-

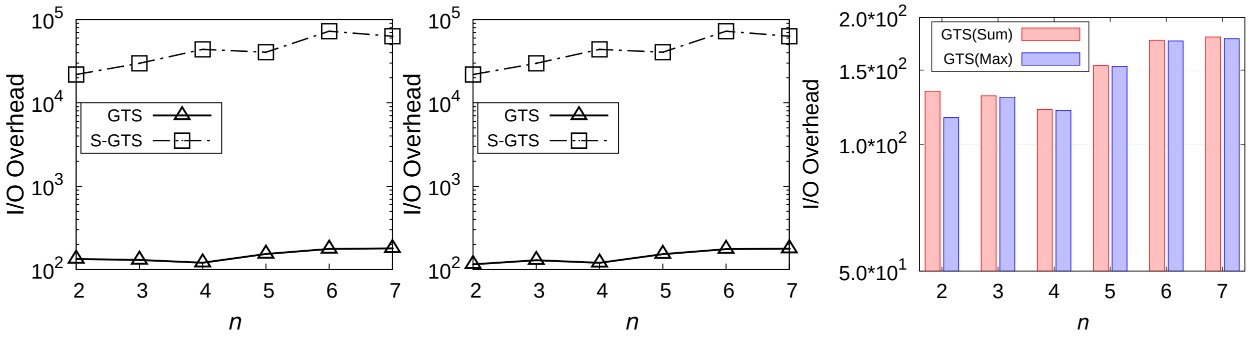
Table 7.1 shows the range and default values used for each parameter. To observe the effect of a parameter in an experiment, the value of the parameter is varied within its range, and other parameters are set to their default values.

7.1.1 Euclidean Space

7.1.1.1 Effect of Group Size (n)



(a) SUM (b) MAX (c) SUM and MAX



(d) SUM (e) MAX (f) SUM and MAX

Figure 7.1: Effect of group size (n) in Euclidean space (California dataset)

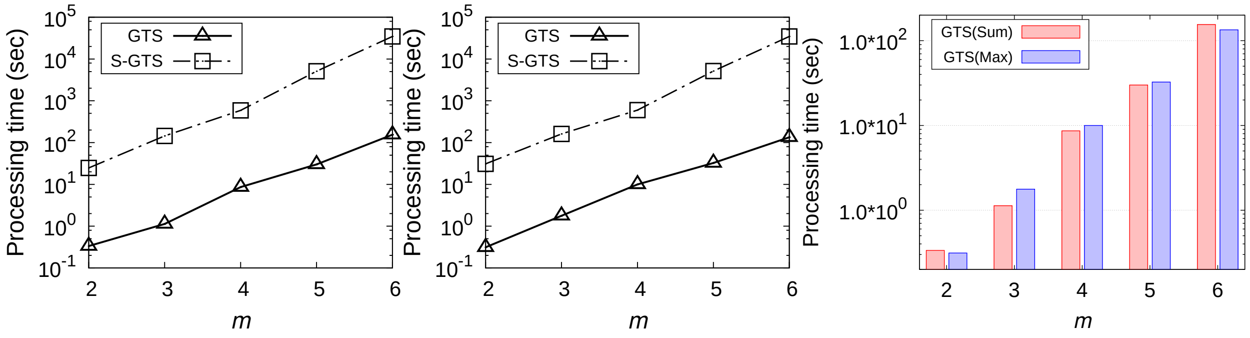
We study the impact of group size on the performance of GTS query by varying the group size using 2, 3, 4, 5, 6 and 7 and measuring the required processing time and number of I/O access from the POI R^*-tree for both aggregate functions SUM and MAX. Figures 7.1(a) and 7.1(b) show the processing time and for aggregate functions SUM and MAX, respectively, for our GTS and S-GTS approaches. For both approaches Figures 7.1(d) and 7.1(e) show the I/O access for aggregate functions SUM and MAX. 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.

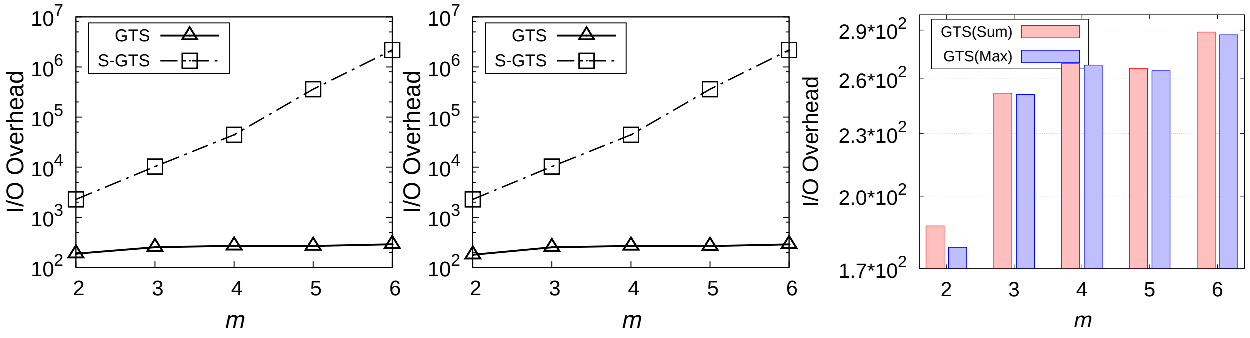
In Figures 7.1(c) and 7.1(f), we show a comparative view of the aggregate functions SUM and MAX for both metrics the processing time and I/O access, respectively. For both metrics, aggregate functions SUM and MAX show almost similar changes with the increase of group size. The reason behind this is, for GTS queries with different aggregate functions, the bound of each group member's elliptic region changes which impacts both metrics, processing time and I/O access. In a GTS query for aggregate function MAX, with minimizing the maximum trip overhead of a group member, it may reduce the bound for that group member which may increase the bound for other group members who may have smaller bound GTS queries for aggregate function SUM. Thus on average for both cases we have almost similar trends for the metrics.

7.1.1.2 Effect of Number of POI Types (m)

In our experiments, we study the impact of number of POI types on the performance of GTS query by varying the number of POI types using 2, 3, 4, 5 and 6 and measuring the required processing time and number of I/O access from the POI R^*-tree for both aggregate functions SUM and MAX. Figures 7.2(a-b) and 7.2(d-e) show that the processing time and I/O access, respectively, for both aggregate function SUM and MAX, 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 Figures 7.2(d-e) 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 $AggTripOvDist$ and T_{max_i} for any user u_i increase by only a small amount.



(a) SUM (b) MAX (c) SUM and MAX



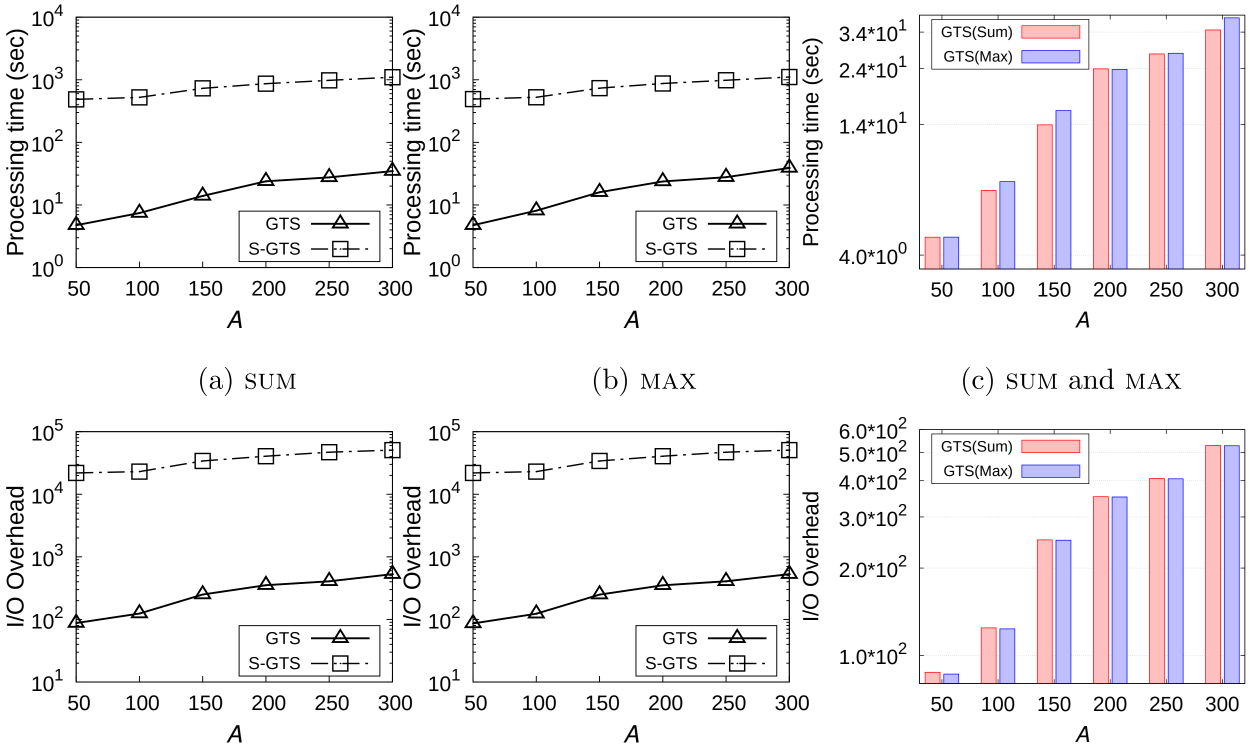
(d) SUM (e) MAX (f) SUM and MAX

Figure 7.2: Effect of number of POI types (m) in Euclidean space (California dataset)

For both metrics, the processing time and I/O access, Figures 7.2(c) and 7.2(f) show a comparative view of the aggregate functions SUM and MAX, respectively. We observe that for both metrics, aggregate functions SUM and MAX show almost similar changes with the increase of number of POI types. For having different aggregate functions, in a GTS query, bound for each group members elliptic search region changes but on average search region remains same. Thus both aggregate functions show similar trends for the processing time and I/O access.

7.1.1.3 Effect of Query Area (A)

We vary the query area by 50 × 50, 100 × 100, 150 × 150, 200 × 200, 250 × 250 and 300 × 300 sq. units in our experiments to observe the impact on the performance of GTS query and measure the required processing time and the number of I/O access from the POI R^*-tree for both aggregate functions SUM and MAX. Figures 7.3(a-b) and 7.3(d-e) show experimental results for different values of the query area A for both aggregate functions. 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 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.



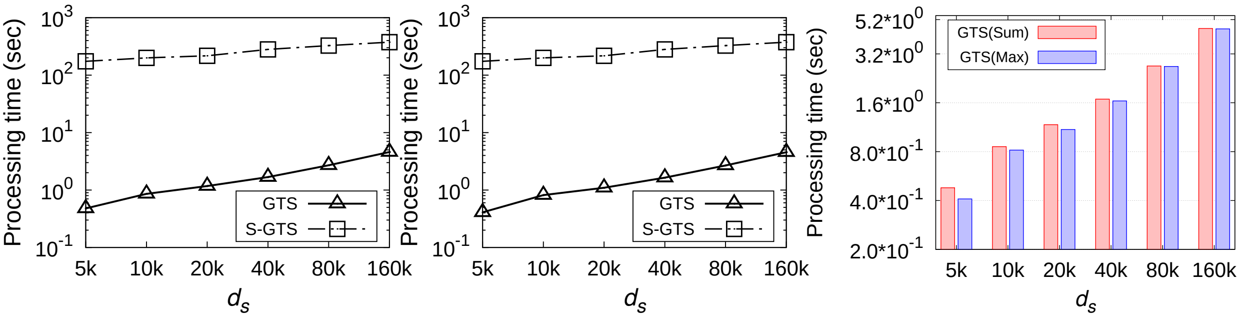
(a) SUM (b) MAX (c) SUM and MAX

(d) SUM (e) MAX (f) SUM and MAX

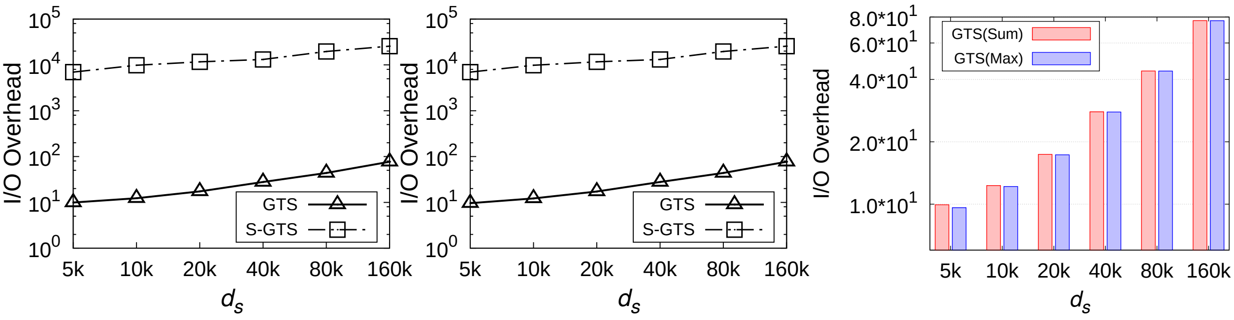
Figure 7.3: Effect of query area (A) in Euclidean space (California dataset)

Figures 7.3(c) and 7.3(f) show a comparative chart of the aggregate functions SUM and MAX for metrics the processing time and I/O access, respectively. Both aggregate functions show similar trends for the processing time and I/O access for the similar reason that we have described in Section 7.1.1.1 and 7.1.1.2.

7.1.1.4 Effect of Dataset Size (d_s)



(a) SUM (b) MAX (c) SUM and MAX



(d) SUM (e) MAX (f) SUM and MAX

Figure 7.4: Effect of dataset size (d_s) in Euclidean space (Synthetic dataset)

In this experiment, we varied the size of synthetic dataset from $5k$ to $160k$ ($5k, 10k, 20k, 40k, 80k, 160k$). To show the effect of dataset size(d_s), we run experiments using synthetic datasets generated using uniform distributions. The corresponding experimental results are shown in Figures 7.4(a-b) and 7.4(d-e) for both aggregate functions SUM and MAX. In this experiment, we examine the performance difference of the two approaches with respect to data set size (d_s). Figures 7.4(a-b) and 7.4(d-e) show that as the size increases, processing time and I/O access increases 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.

Both aggregate functions SUM and MAX show similar trends for the processing time and I/O access which we can observe deeply in Figures 7.4(c) and 7.4(f) for the processing time and I/O access, respectively. The reason behind this is similar that we have described in Section 7.1.1.1, 7.1.1.2 and 7.1.1.3.

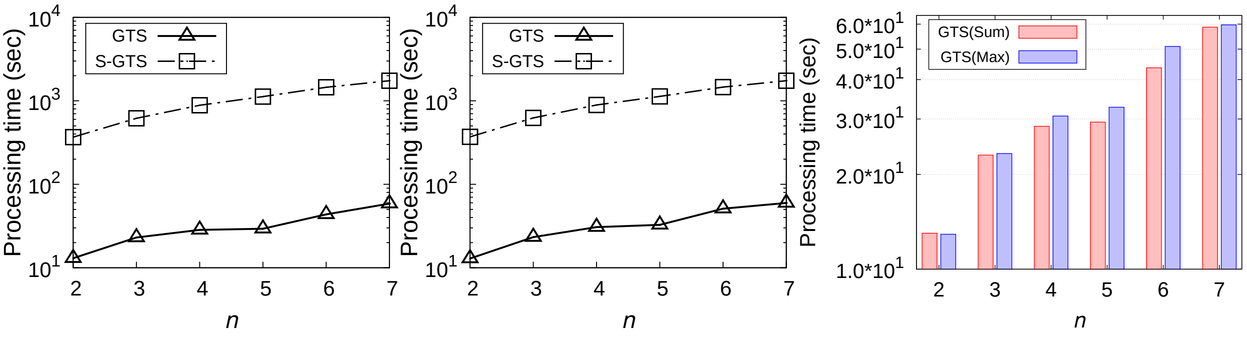
7.1.2 Road Networks

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 for both aggregate functions SUM and MAX.

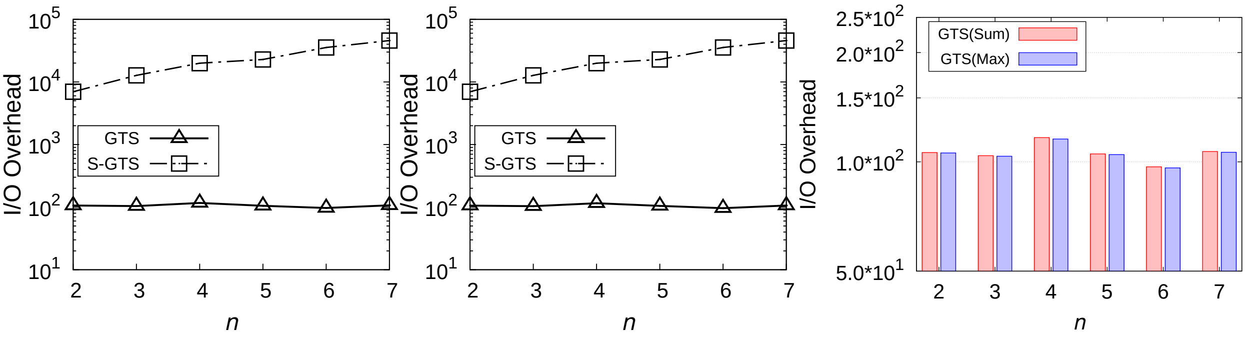
7.1.2.1 Effect of Group Size (n)

To analysis the impact of group size on the performance of GTS query, we vary the group size from 2 to 7 $(2, 3, 4, 5, 6, 7)$. For both aggregate functions SUM and MAX, Figures 7.5(a-b) and 7.5(d-e) show that the query processing time increases 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.

In Figures 7.5(c) and 7.5(f), we show a comparative chart of the aggregate functions SUM and MAX for both metrics, the processing time and I/O access, respectively. For both metrics, aggregate functions SUM and MAX shows almost similar changes with the increase of group size. The reason behind this is, for GTS queries with different aggregate functions, the bound of each group member's elliptic region changes which impacts both processing time and I/O access. In a GTS query for aggregate function MAX, with minimizing the maximum trip overhead of a group member, it may reduce the bound for that group member which may increase the bound for other group members who may have smaller bound GTS queries for aggregate function SUM. Thus on average for both cases we have almost similar trend for the metrics.



(a) SUM (b) MAX (c) SUM and MAX



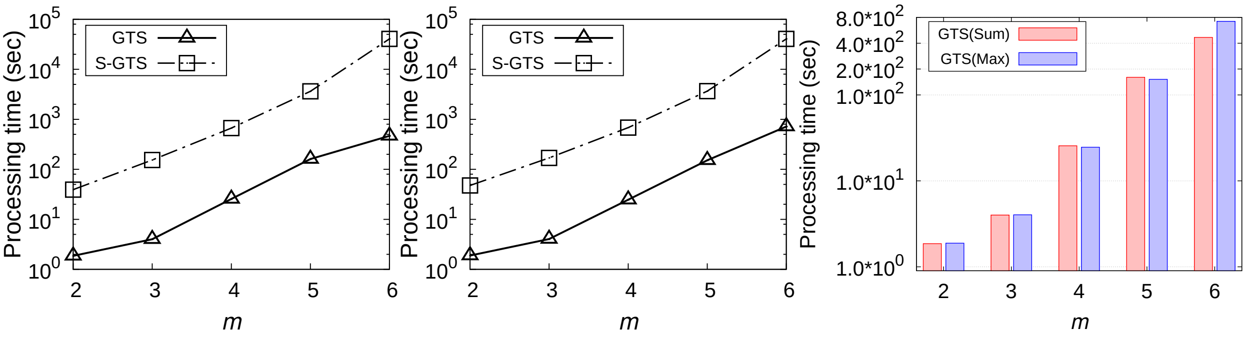
(d) SUM (e) MAX (f) SUM and MAX

Figure 7.5: Effect of group size (n) in road networks (California dataset)

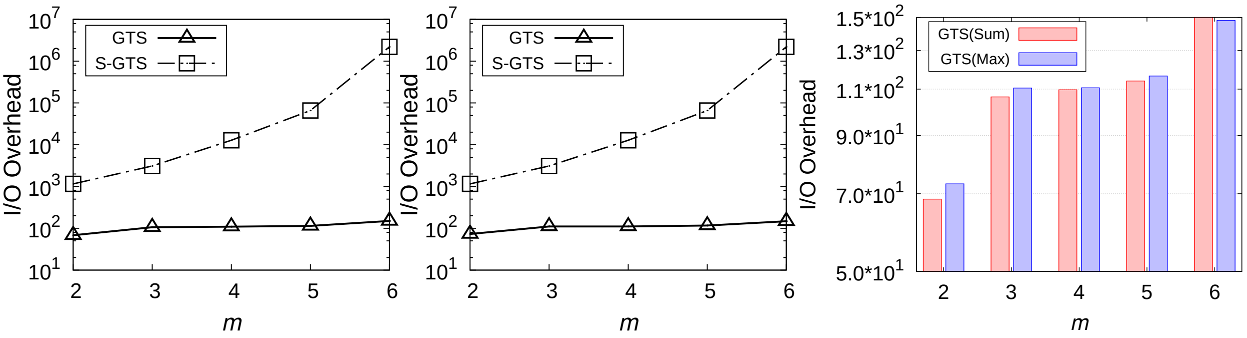
7.1.2.2 Effect of Number of POI Types (m)

Figures 7.6(a-b) and 7.6(d-e) show the performance of the GTS approach and the S-GTS approach for varying the total number of POI types m for both aggregate functions SUM and MAX. In this experiment, we varied the number of POI types from 2 to 6 $(2, 3, 4, 5, 6)$. We observe that the performance trends are similar to those for the Euclidean space. For any number of POI types, the GTS approach outperforms the S-GTS approach in terms of both I/O access and processing time.

For metrics the processing time and I/O access, Figures 7.6(c) and 7.6(f) show a comparative chart of the aggregate functions SUM and MAX, respectively. We observe that for both metrics, aggregate functions SUM and MAX show almost similar changes with the increase of number of POI types. For having different aggregate functions, in a GTS query, bound for each group member's elliptic search region changes but on average search region remains same. Thus both aggregate functions shows similar trends for the processing time and I/O access.



(a) SUM (b) MAX (c) SUM and MAX

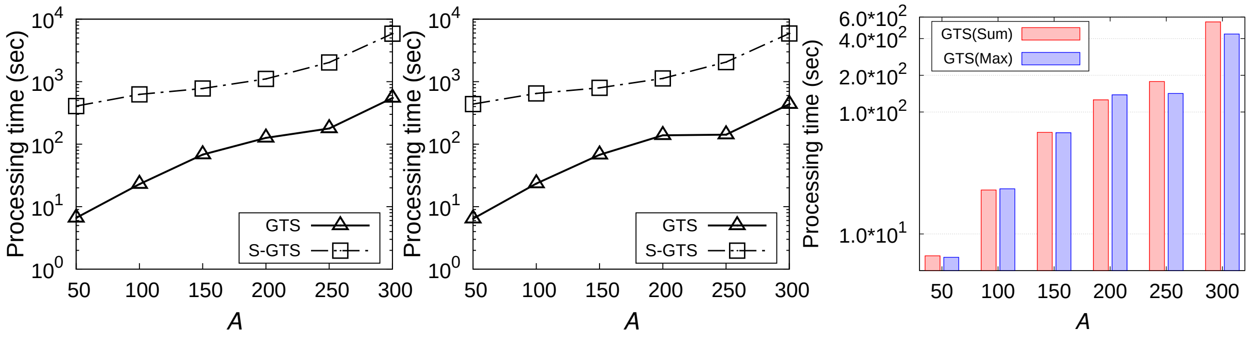


(d) SUM (e) MAX (f) SUM and MAX

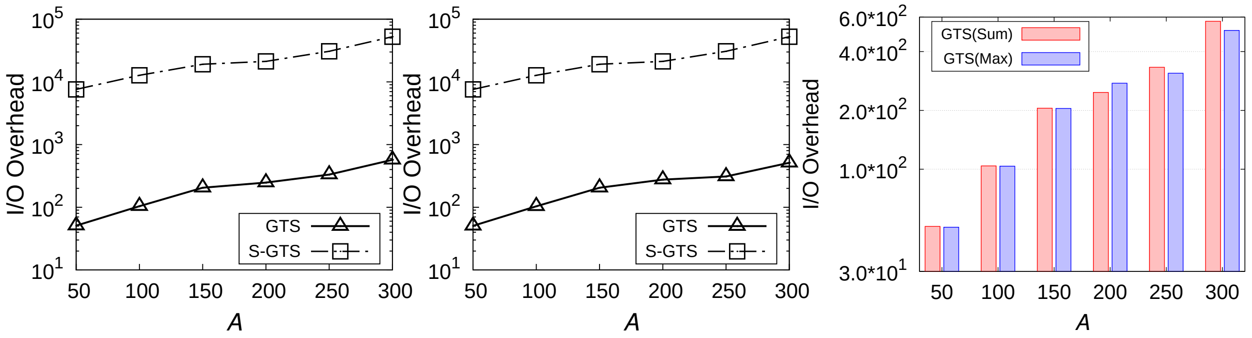
Figure 7.6: Effect of number of POI types (m) in road networks (California dataset)

7.1.2.3 Effect of Query Area (A)

We vary the query area by 50×50, 100×100, 150×150, 200×200, 250×250 and 300×300 sq. units in our experiments to observe the impact on the performance of GTS query and measure the required processing time and number of I/O access from the POI R^*-tree for both aggregate functions SUM and MAX. Figures 7.7(a-b) and 7.7(d-e) 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. Figures 7.7(c) and 7.7(f) show a comparative chart of the aggregate functions SUM and MAX, respectively. We observe that for both metrics, aggregate functions SUM and MAX shows almost similar changes with the increase of number of POI types for the similar reason that we already described in Section 7.1.2.1 and in Section 7.1.2.2.



(a) SUM (b) MAX (c) SUM and MAX



(d) SUM (e) MAX (f) SUM and MAX

Figure 7.7: Effect of query area (A) in road networks (California dataset)

7.2 UGTS Queries

For UGTS queries where every group member visit uniform number of POIs, we performed similar set of experiments that we performed for GTS queries by varying the following parameters for both aggregate functions SUM and MAX:

(i) the group size n

(ii) the number of specified POI types m

(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)

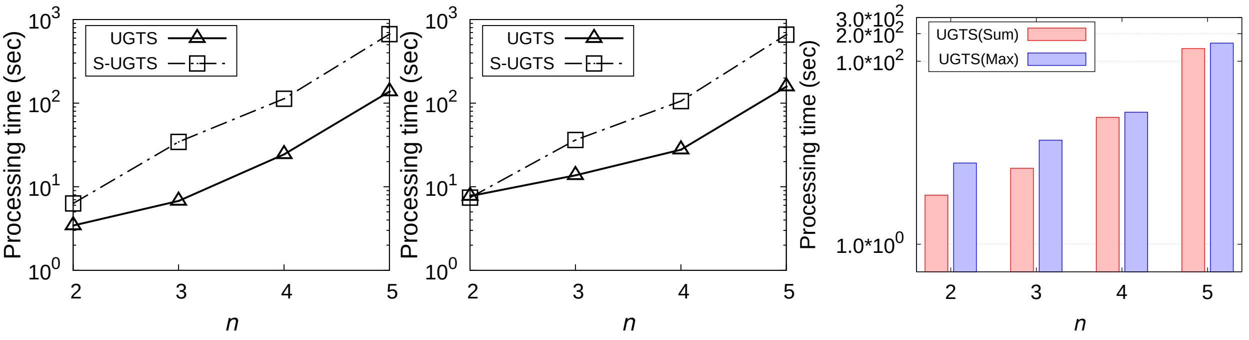
Table 7.2: Parameter settings for UGTS queries

Parameter	**Values**	**Default**
Group size(n)	2, 3, 4, 5	3
Number of POI types (m)	3, 6, 9	6
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	-

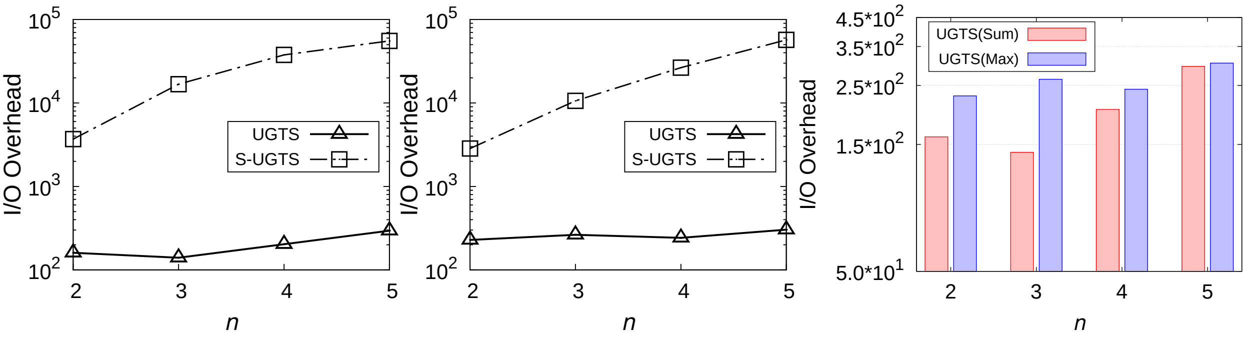
Table 7.2 shows the range of values of different parameters used and the default value of each parameter. A parameter was set to the default value in experiments where any other parameter was being varied.

7.2.1 Euclidean Space

7.2.1.1 Effect of Group Size (n)



(a) SUM (b) MAX (c) SUM and MAX



(d) SUM (e) MAX (f) SUM and MAX

Figure 7.8: Effect of group size (n) in Euclidean space (California dataset)

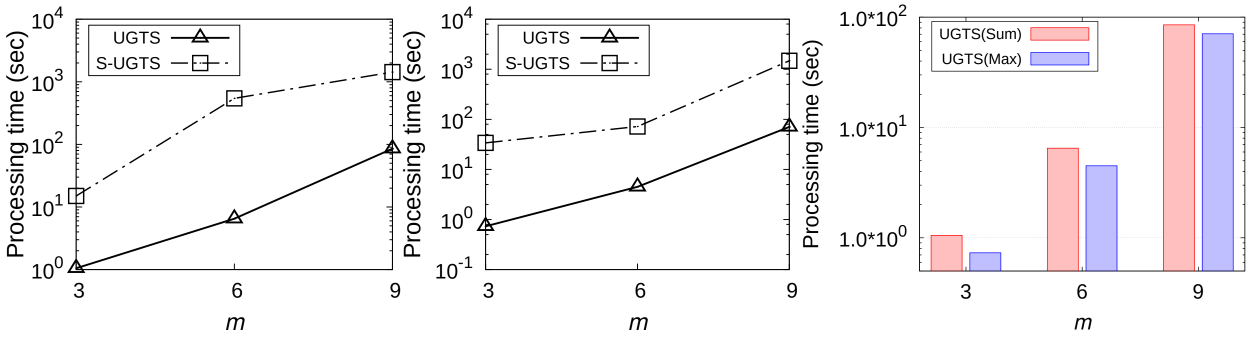
To study the impact of group size on the performance of UGTS query we vary the group size from 2 to 5 $(2, 3, 4, 5)$. With the increase of group size in UGTS queries, the number of POI types increases thus our default POI types is 6 which follows that the number of uniform POI type is 2. So with the increase of group size, 2, 3, 4 and 5, the number of POI types become 4, 6, 8 and 10. For different values of group sizes we measure the required processing time and number of I/O access from the POI R^*-tree for both aggregate functions SUM and MAX. Figures 7.8(a-b) and 7.8(d-e) show the processing time and I/O access for aggregate functions SUM and MAX, respectively, for our UGTS and S-UGTS approaches. We observe that both processing time and I/O access slightly increase with the increase of the group size. Our UGTS approach requires significantly less processing time and I/O access than the S-UGTS approach, which is expected. The S-UGTS approach computes the optimal trips for each group member and for every possible combination of POI types independently having uniform number of POI types, and thus, accesses the same POIs multiple times in the database. On the other hand, our UGTS 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.

In Figures 7.8(c) and 7.8(f), we show a comparative chart of the aggregate functions SUM and MAX for both metrics the processing time and I/O access, respectively. For both metrics, aggregate functions SUM and MAX shows almost similar changes with the increase of group size. The reason behind this is, for UGTS queries with different aggregate functions, the bound of each group member's elliptic region changes which impacts both processing time and I/O access. In a UGTS query for aggregate function MAX, with minimizing the maximum trip overhead of a group member, it may reduce the bound for that group member which may increase the bound for other group members who may have smaller bound UGTS queries for aggregate function SUM. Thus on average for both cases we have almost similar trend for the metrics.

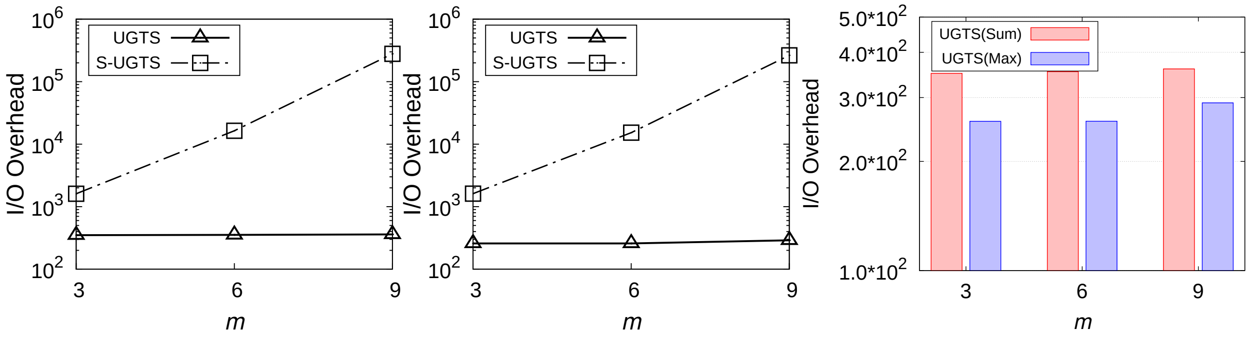
7.2.1.2 Effect of Number of POI Types (m)

In Figures 7.9(a-b) and 7.9(d-e), we show the performance of our proposed UGTS and straightforward S-UGTS approach when total number of POI types m is varied from 3 to 9 for both aggregate functions SUM and MAX. The results show that for any number of POI types our proposed approach outperform S-UGTS by a large margin in terms of I/O access and processing time. We estimated that our efficient incremental GTS approach takes on the average approximately 33 times less processing time and 299 times less I/O access than the S-GTS approach.

In Figures 7.9(c) and 7.9(f) we observe a chart of the aggregate functions SUM and MAX for metrics the processing time and I/O access, respectively. Both aggregate functions show similar trends for the processing time and I/O access for the similar reason that we mentioned for the experiments of varying n in Section 7.2.1.1.



(a) SUM (b) MAX (c) SUM and MAX

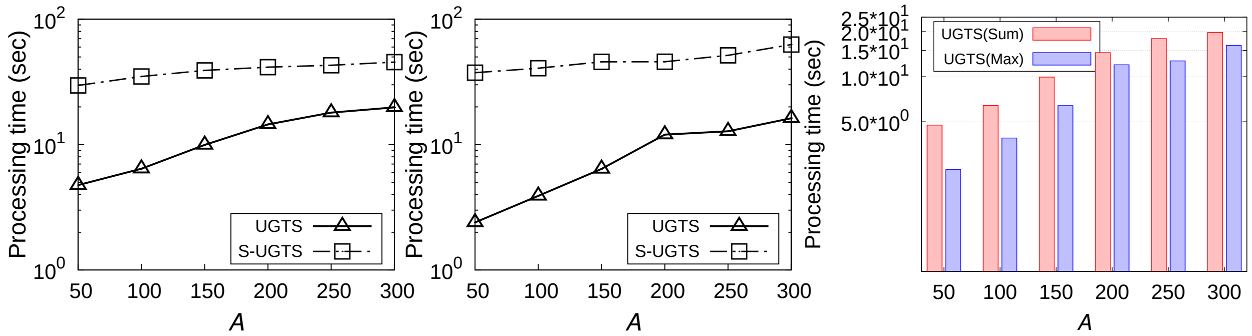


(d) SUM (e) MAX (f) SUM and MAX

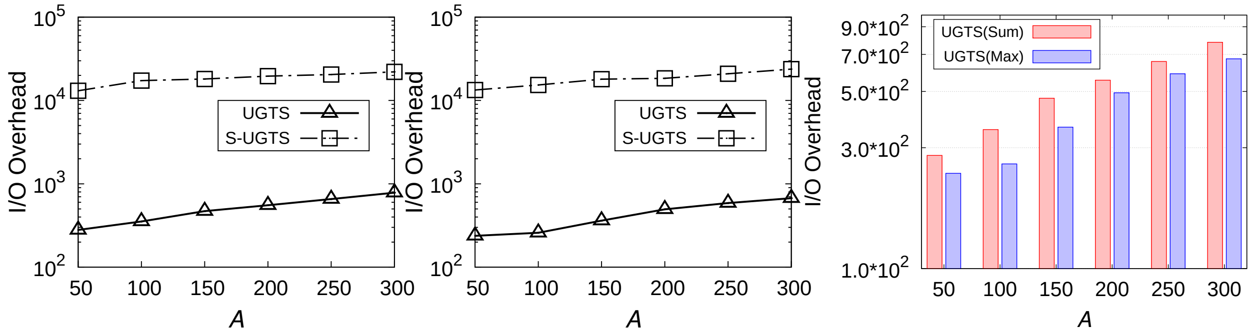
Figure 7.9: Effect of number of POI types (m) in Euclidean space (California dataset)

7.2.1.3 Effect of Query Area (A)

In this experiment to observe the impact on the performance of UGTS queries, we vary the query area by 50 × 50, 100 × 100, 150 × 150, 200 × 200, 250 × 250 and 300 × 300 sq. units and measure the required processing time and number of I/O access from the POI R^*-tree for both aggregate functions SUM and MAX. Figures 7.10(a-b) and 7.10(d-e) shows experimental results for different values of query area A for both aggregate functions SUM and MAX. We see that for both approaches, the processing time and I/O access increases with the increase of A, although the rate of increase is less than that of Figures 7.8 and 7.9. For both metrics, our UGTS approach outperforms the S-UGTS approach significantly.



(a) SUM (b) MAX (c) SUM and MAX



(d) SUM (e) MAX (f) SUM and MAX

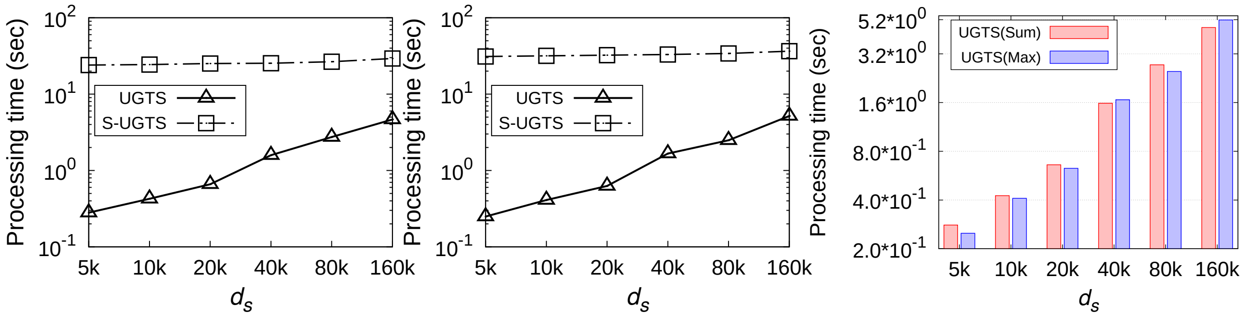
Figure 7.10: Effect of query area (A) in Euclidean space (California dataset)

Figures 7.10(c) and 7.10(f) show a comparative chart of the aggregate functions SUM and MAX for metrics the processing time and I/O access, respectively. Both aggregate functions show similar trends for the processing time and I/O access for the similar reason that we mentioned for the experiments of varying n in Section 7.2.1.1.

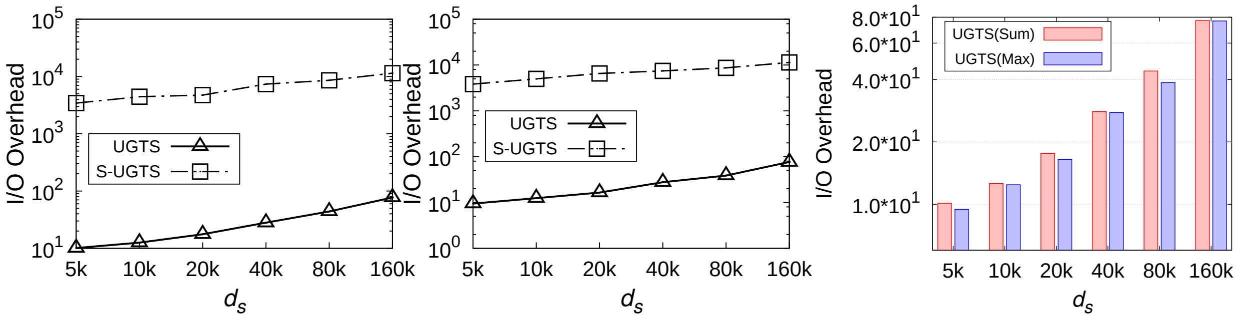
7.2.1.4 Effect of Dataset Size (d_s)

In this experiment, we examine the performance difference of the two approaches with respect to data set size (d_s). We varied the size of synthetic dataset from $5k$ to $160k$ ($5k, 10k, 20k, 40k, 80k160k$). To show the effect of dataset size(d_s), we run experiments using synthetic datasets generated using uniform distributions. The corresponding experimental results are shown in Figures 7.11(a-b) and 7.11(d-e) which shows that as size increases, processing time and I/O access increases for both approaches. But incremental approach takes much less processing time and I/O access than the straightforward approach.

Both aggregate functions SUM and MAX show similar trends for the processing time and I/O access that we deeply observe in Figures 7.11(c) and 7.11(f) for the processing time and I/O access, respectively. The reason behind this is similar that we have described in Section 7.2.1.1.



(a)SUM (b)MAX (c)SUM and MAX



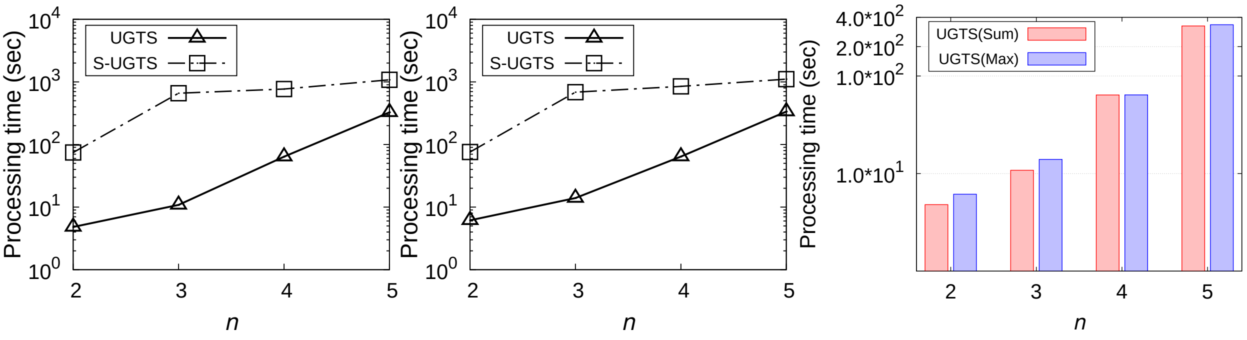
(d) SUM (e) MAX (f) SUM and MAX

Figure 7.11: Effect of dataset size(d_s) in Euclidean space (Synthetic dataset)

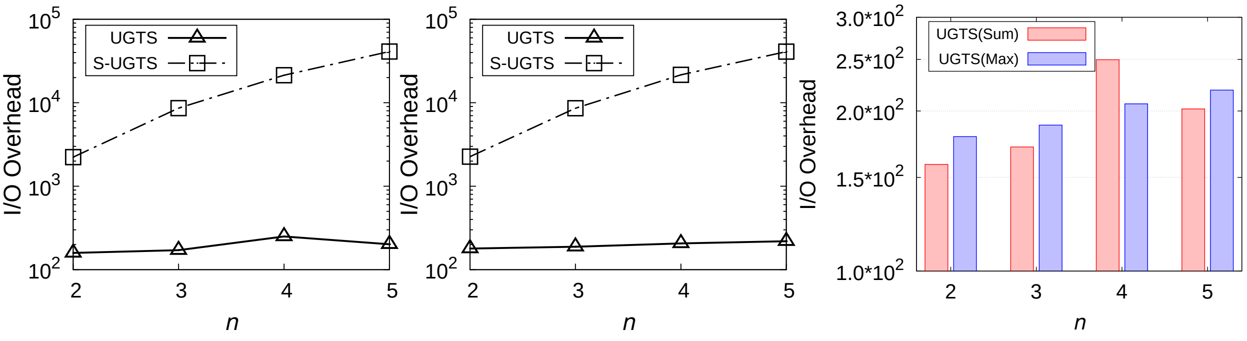
7.2.2 Road Networks

7.2.2.1 Effect of Group Size (n)

Figures 7.12(a-b) and 7.12(d-e) show the processing time and I/O access, respectively, for our proposed UGTS approach and the S-UGTS approach. We observe that, with the increase of group size n, for our UGTS approach I/O access slightly changes where for the S-UGTS approach I/O access increases with significant amount. For both approaches, query processing time increases with the increase of group size n. In Figures 7.12(c) and 7.12(f) we observe that for both aggregate functions SUM and MAX, both processing time and I/O access metrics show almost similar trends which is expected.



(a) SUM (b) MAX (c) SUM and MAX

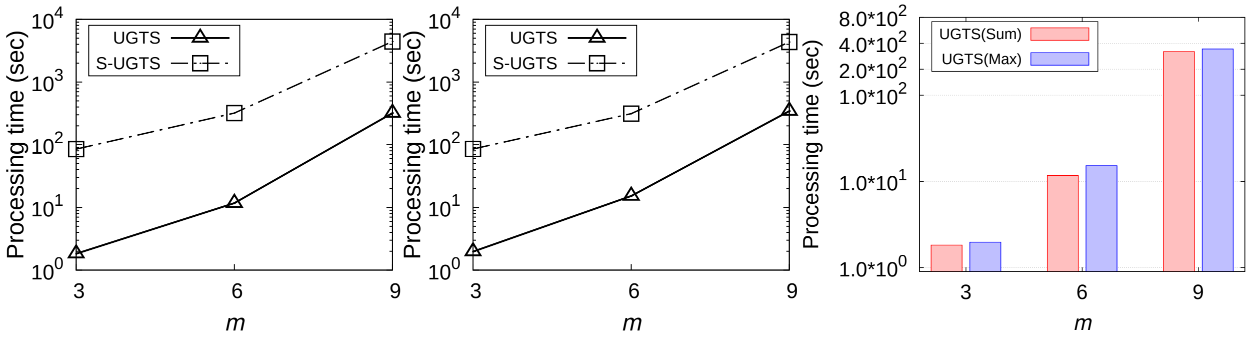


(d) SUM (e) MAX (f) SUM and MAX

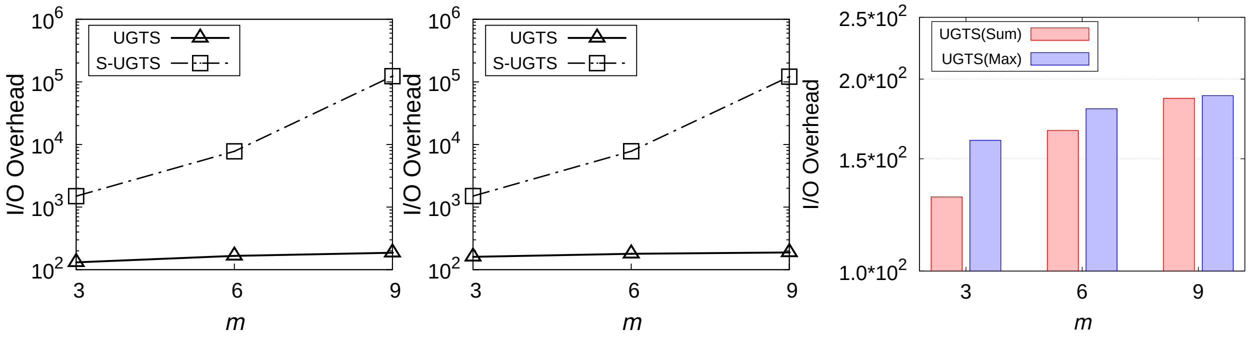
Figure 7.12: Effect of group size (n) in road networks (California dataset)

7.2.2.2 Effect of Number of POI Types (m)

In Figures 7.13(a-b) and 7.13(d-e), we show the performance of our proposed UGTS approach and the S-UGTS approach by varying the total number of POI types m. The results show that for any number of POI types our proposed approach, outperform S-GTS by in terms of I/O access and processing time. We observe that the performance trends are similar to those for the Euclidean space. For any number of POI types, the UGTS approach outperforms the S-UGTS approach in terms of both I/O access and processing time. For metrics the processing time and I/O access both aggregate functions SUM and MAX shows almost similar changes with the increase of number of POI types in Figures 7.13(c) and 7.13(f), respectively, similar to other experiments.



(a) SUM (b) MAX (c) SUM and MAX

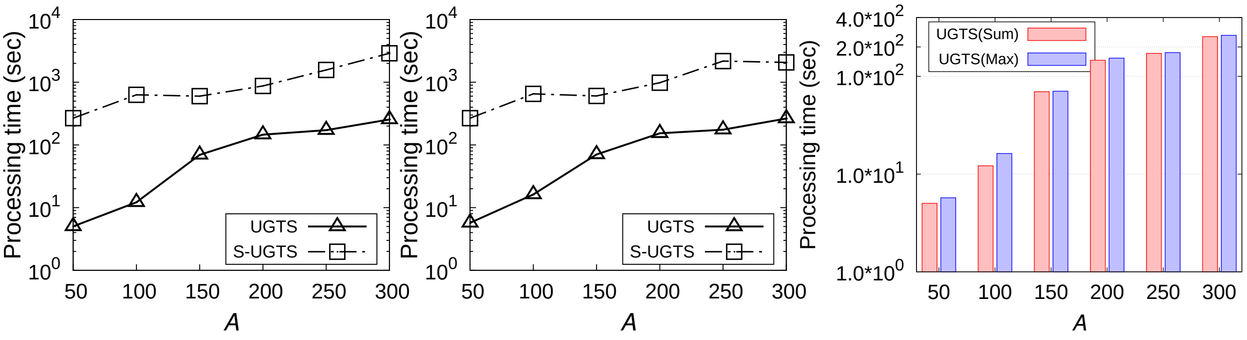


(d) SUM (e) MAX (f) SUM and MAX

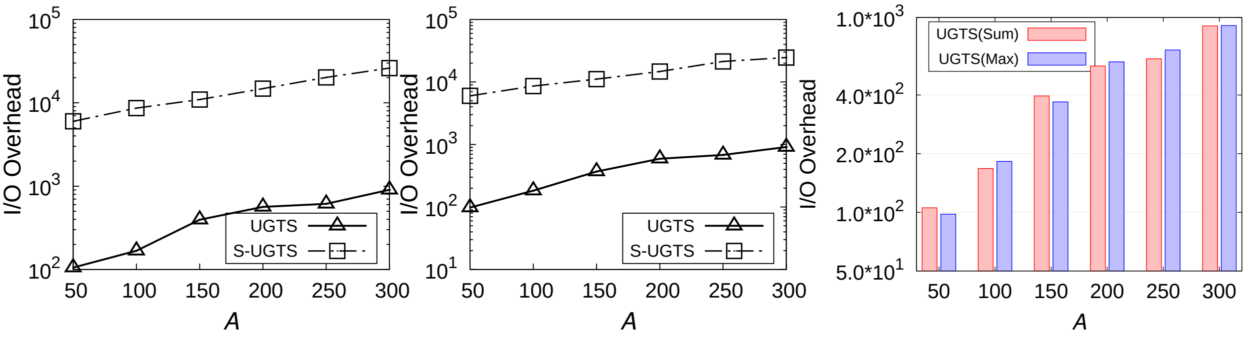
Figure 7.13: Effect of number of POI types (m) in road networks (California dataset)

7.2.2.3 Effect of Query Area (A)

Figures 7.14(d-e) and 7.14(a-b) show the comparison of required I/O access and query processing time between our proposed UGTS approach and the S-UGTS approach by varying the query area (A). We vary the query area by 50×50, 100×100, 150×150, 200×200, 250×250 and 300×300 sq. units in our experiments to observe the impact on the performance of UGTS query. We estimated that, for the GTS approach, both query processing time and required I/O access increases slightly with the increase of query area (A). For the S-UGTS approach, I/O access increases slightly with the change of query area but changes in query processing time is not visible so much. In Figures 7.14(c) and 7.14(f), we observe that for both metrics the processing time and I/O access aggregate functions SUM and MAX show almost similar changes with the change of area size.



(a) SUM (b) MAX (c) SUM and MAX



(d) SUM (e) MAX (f) SUM and MAX

Figure 7.14: Effect of query area (A) in road networks (California dataset)

Chapter 8

Conclusions

In this thesis, 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 aggregate trip overhead distance of the group members. We propose the first solution to evaluate GTS queries in both Euclidean space and road networks. To schedule trips among group members, in GTS queries, we consider two different aggregate trip overhead distances. The aggregate trip overhead distance can be either the total or the maximum of the trip overhead distances of group members that we measured using aggregate functions SUM and MAX, respectively.

Specifically, we have proposed refinement techniques for the POI search space and a dynamic approach to schedule trips among group members, which are the key ideas behind the efficiency of our approach. We have exploited geometric properties to refine the POI search space and prune POIs to reduce the number of possible combinations of trips among group members. To schedule trips among group members, we have developed an efficient dynamic programming technique that eliminates the trip combinations that can not be a part of the optimal query answer.

We have proposed a variant of GTS queries, a uniform GTS (UGTS) query that schedules trips by uniformly distributing the required POI types among group members, i.e., each trip visits equal number of POI types and the aggregate trip overhead distance is minimum. In this thesis, we have provided an efficient solution for processing UGTS queries in both Euclidean and road networks. In addition to fixing the number of POI types, we have extended our approach for processing GTS and

UGTS queries with constraints like the dependencies among POIs, and/or dependencies among POIs and group members.

Since there exists no approach to process GTS or UGTS queries in the literature, to validate the efficiency of our proposed approach in experiments, we have developed straightforward approaches for processing GTS queries (S-GTS) and UGTS (S-UGTS) queries using existing trip planning algorithms. We have performed extensive experimental evaluation of the proposed techniques and provided an comparative analysis of experimental results using both real and synthetic datasets. Our experimental results show the performance analysis of our proposed approach for different parameters. Experiments show that our GTS approach is on average 107 and 113 times faster and requires on average 635 and 668 times less I/Os for aggregate function SUM and MAX, respectively, than the straightforward approach for the Euclidean space. For road networks, we observed that our GTS approach requires on average 30 and 29 times less processing time and 1021 and 1033 times less I/O access for aggregate function SUM and MAX, respectively, than the straightforward approach.

In the future, we aim to protect location privacy [29–31] of users for GTS queries and variants. To protect location privacy, a user may reveal encrypted [32], false [33] or cloaked [34] locations to the LSP. The challenge is to find the query answer for the actual location of the user in real time based on encrypted, false or cloaked locations. In the literature, there exist a number of privacy preserving algorithms for processing variant spatial queries like nearest neighbor queries [35, 36], group nearest neighbor queries [37, 38], and trip planning queries [27]. However, these algorithms are not directly applicable for GTS queries.

In this thesis, we have only considered distance for finding GTS query answers. In reality, all POIs of a single POI type may not have the same rating. The ratings of POIs of a POI type like restaurant may vary based on the quality of service, and price. In the future we will focus on considering on rating of POIs in addition to the distance for evaluating GTS queries and variants.

References

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

[2] Haiquan Chen, Wei-Shinn Ku, Min-Te Sun, and Roger Zimmermann. The multi-rule partial sequenced route query. In *SIGSPATIAL*, pages 10:1–10, 2008.

[3] Tanzima Hashem, Sukarna Barua, Mohammed Eunus Ali, Lars Kulik, and Egemen Tanin. Efficient computation of trips with friends and families. In *CIKM*, pages 931–940, 2015.

[4] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On trip planning queries in spatial databases. In *SSTD*, pages 273–290, 2005.

[5] Hongga Li, Hua Lu, Bo Huang, and Zhiyong Huang. Two ellipse-based pruning methods for group nearest neighbor queries. In *GIS*, pages 192–199, 2005.

[6] Mehdi Sharifzadeh, Mohammad R. Kolahdouzan, and Cyrus Shahabi. The optimal sequenced route query. *VLDB J.*, 17(4):765–787, 2008.

[7] Tanzima Hashem, Tahrima Hashem, Mohammed Eunus Ali, and Lars Kulik. Group trip planning queries in spatial databases. In *SSTD*, pages 259–276, 2013.

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

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

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

[11] Jun Li, Qirui Sun, MengChu Zhou, and Xianzhong Dai. A new multiple traveling salesman problem and its genetic algorithm-based solution. In *SMC*, pages 627–632, 2013.

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

[13] Yutaka Ohsawa, Htoo Htoo, Noboru Sonehara, and Masao Sakauchi. Sequenced route query in road network distance based on incremental euclidean restriction. In *DEXA*, pages 484–491, 2012.

[14] Nirmesh Malviya, Samuel Madden, and Arnab Bhattacharya. A continuous query system for dynamic route planning. In *ICDE*, pages 792–803, 2011.

[15] Hossain Mahmud, Ashfaq Mahmood Amin, Mohammed Eunus Ali, Tanzima Hashem, and Sarana Nutanong. A group based approach for path queries in road networks. In *SSTD*, pages 367–385, 2013.

[16] Robert Geisberger, Michael N. Rice, Peter Sanders, and Vassilis J. Tsotras. Route planning with flexible edge restrictions. *ACM Journal of Experimental Algorithmics*, 17(1), 2012.

[17] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. Route planning with flexible objective functions. In *ALENEX*, pages 124–137, 2010.

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

[19] Samiha Samrose, Tanzima Hashem, Sukarna Barua, Mohammed Eunus Ali, Mohammad Hafiz Uddin, and Md. Iftekhar Mahmud. Efficient computation of group optimal sequenced routes in road networks. In *MDM*, pages 122–127, 2015.

[20] Dimitris Papadias, Qiongmao Shen, Yufei Tao, and Kyriakos Mouratidis. Group nearest neighbor queries. In *ICDE*, pages 301–312, Washington, DC, USA, 2004. IEEE Computer Society.

[21] Dimitris Papadias, Yufei Tao, Kyriakos Mouratidis, and Chun Kit Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.*, 30(2):529–576, 2005.

[22] Man Lung Yiu, Nikos Mamoulis, and Dimitris Papadias. Aggregate nearest neighbor queries in road networks. *IEEE Trans. Knowl. Data Eng.*, 17(6):820–833, 2005.

[23] Sansarkhuu Namnandorj, Hanxiong Chen, Kazutaka Furuse, and Nobuo Ohbo. Efficient bounds in finding aggregate nearest neighbors. In Sourav S. Bhowmick, Josef Kng, and Roland Wagner, editors, *DEXA*, volume 5181 of *Lecture Notes in Computer Science*, pages 693–700. Springer, 2008.

[24] Petrica C. Pop, Oliviu Matei, and C. Sabo. A new approach for solving the generalized traveling salesman problem. In *Hybrid Metaheuristics*, pages 62–72, 2010.

[25] Zhou Xu and Brian Rodrigues. A 3/2-approximation algorithm for multiple depot multiple traveling salesman problem. In *SWAT*, pages 127–138, 2010.

[26] Hongga Li, Hua Lu, Bo Huang, and Zhiyong Huang. Two ellipse-based pruning methods for group nearest neighbor queries. In *International Workshop on GIS*, pages 192–199, 2005.

[27] Subarna Chowdhury Soma, Tanzima Hashem, Muhammad Aamir Cheema, and Samiha Samrose. Trip planning queries with location privacy in spatial databases. *World Wide Web*, 2016.

[28] `http://rtreeportal.org/`.

[29] Chi-Yin Chow, Mohamed F. Mokbel, and Walid G. Aref. Casper*: Query processing for location services without compromising privacy. *ACM Trans. Database Syst.*, 34(4):24:1–24:48.

[30] B. Gedik and L. Liu. Protecting location privacy with personalized k-anonymity: Architecture and algorithms. *IEEE TMC*, 7(1):1–18, 2008.

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

[32] Gabriel Ghinita, Panos Kalnis, Ali Khoshgozaran, Cyrus Shahabi, and Kian-Lee Tan. Private queries in location based services: anonymizers are not necessary. In *SIGMOD*, pages 121–132, 2008.

[33] Man L. Yiu, Christian S. Jensen, Jesper Møller, and Hua Lu. Design and analysis of a ranking approach to private location-based services. *ACM TODS*, 36(2):10, 2011.

[34] Tanzima Hashem and Lars Kulik. “Don’t trust anyone”: Privacy protection for location-based services. *PMC*, 7:44–59, 2011.

[35] Ali Khoshgozaran and Cyrus Shahabi. Blind evaluation of nearest neighbor queries using space transformation to preserve location privacy. In *SSTD*, pages 239–257, 2007.

[36] Man L. Yiu, Christian S. Jensen, Xuegang Huang, and Hua Lu. Spacetwist: Managing the trade-offs among location privacy, query performance, and query accuracy in mobile services. In *ICDE*, pages 366–375, 2008.

[37] Tanzima Hashem, Mohammed Eunus Ali, Lars Kulik, Egemen Tanin, and Anthony Quattrone. Protecting privacy for group nearest neighbor queries with crowdsourced data and computing. In *UbiComp*, pages 559–562, 2013.

[38] Tanzima Hashem, Lars Kulik, and Rui Zhang. Privacy preserving group nearest neighbor queries. In *EDBT*, pages 489–500, 2010.