

MASTER'S THESIS

RECOMMENDING PLANS FOR VISITING
TOURISTIC ATTRACTIONS
EXPLOITING GPS-BASED PUBLIC
TRANSPORTATION SYSTEM

KEMAL CAGIN GÜLSEN

APRIL 2017

ALBERT-LUDWIGS UNIVERSITÄT FREIBURG
DEPARTMENT OF COMPUTER SCIENCE
CHAIR OF DATABASES AND INFORMATION SYSTEMS

Candidate

Kemal Cagin Gülsen

Matr. number

3957131

Working period

21. 10. 2016 – 21. 04. 2017

First Examiner

Prof. Dr. Georg Lausen

Second Examiner

Prof. Dr. Peter Fischer

Supervisors

Dr. Marco Muñoz - Aalborg University

Victor Anthony Arrascue Ayala - University of Freiburg

Abstract

When tourists visit new cities or countries, they might be interested in visiting local attractions. To achieve this, they can approach a local tourist information center from which they typically obtain a list of those sites. The user can then determine, based on the places' descriptions, which attractions are relevant and can search for a way to reach those places. This task is typically very time-consuming, especially the latter, since the user is in an unknown city and needs to understand how local public transportation functions.

Recommender systems (RS) are tools which aim to assist the user in a decision process and seek to alleviate information overload. Whereas there are several RS focused on touristic recommendations, not many of them integrate the information with the public transportation network. When it comes to developing countries, the situation is typically worse. In these countries there exist several challenges that prevent tools to produce reliable trip plans, such as traffic jams, accidents, construction work and unorganized transportation networks.

The goal of this Thesis is to combine two different paradigms, i.e. that of Recommender Systems and Route Planning, to create a tool that makes it possible to produce trip plans even when delays or other events are considered. To demonstrate the efficacy of our tool, we make use of real data from the city of Izmir. In addition to this, we carry extensive evaluation which shows that in terms of performance our tool is able to satisfy the initial requirements.

Kurzfassung

Wenn Touristen neue Städte oder Länder besuchen, sind sie daran interessiert einheimische Sehenswürdigkeiten zu besichtigen. Dazu können sie sich eine Liste der Standorte beschaffen, in dem sie sich einem örtlichen Touristeninformationszentrum wenden. Der Benutzer kann somit auf Grundlage der Ortsbeschreibungen beschließen, welche Sehenswürdigkeiten relevant sind und können nach einem Weg suchen, um diese Orte zu erreichen. Diese Aufgabe gestaltet sich in der Regel als sehr zeitaufwendig, besonders die letztere, da der Benutzer sich in einer unbekannten Stadt befindet und die Funktionsweise der einheimischen öffentlichen Verkehrsmittel verstehen muss.

Recommender Systems (RS) sind Werkzeuge, die darauf abzielen den Benutzer bei einem Entscheidungsprozess zu unterstützen und die Informationsüberlastung zu lindern. Obwohl mehrere Recommender Systems existieren, die sich auf Empfehlungen für die Touristen konzentrieren, integrieren viele von ihnen nicht die Informationen der öffentlichen Verkehrsnetze. Wenn es sich dabei um die Entwicklungsländer handelt, ist diese Situation typischerweise schlechter. In diesen Ländern gibt es mehrere Herausforderungen wie etwa Verkehrstaus, Unfälle, Baustellen und unregelmäßige Verkehrsnetze, die diese Werkzeuge daran hindern zuverlässige Reisepläne zu entwickeln.

Das Ziel dieser Thesis ist es, zwei Paradigmen zu kombinieren, die Recommender Systems und die Routenplanung, um ein Werkzeug zu erschaffen, das in Anbetracht der Verzögerungen oder anderen Ereignissen, Reisepläne erstellen kann. Ergänzend dazu, führen wir umfangreiche Auswertungen durch, die zeigen, dass das Werkzeug hinsichtlich der Leistung in der Lage ist die ursprünglichen Anforderungen erfüllen kann.

Contents

| | |
|---|------------|
| Abstract | II |
| Kurzfassung | III |
| List of Tables | VII |
| 1 Introduction | 1 |
| 1.1 Initial goal and contributions | 2 |
| 1.2 Thesis outline | 4 |
| 2 Preliminaries | 5 |
| 2.1 Transit Route Planning | 6 |
| 2.1.1 Graphs and Dijkstra’s Shortest Path Algorithm | 7 |
| 2.1.2 Time-Dependent Model | 8 |
| 2.1.3 Time-Expanded Model | 11 |
| 2.1.4 Optimization Techniques | 12 |
| 2.2 Recommender Systems | 19 |
| 2.2.1 Recommendation Techniques | 20 |
| 2.2.2 Knowledge-Based Recommender Systems | 21 |
| 2.3 Related Work | 23 |
| 2.3.1 Route Planning in Public Transit Networks | 23 |
| 2.3.2 Tourist Trip Plan Generation | 24 |
| 3 Problem Formalization | 26 |
| 3.1 Problem statement | 26 |

| | | |
|----------|---|-----------|
| 3.1.1 | Tourist Trip Design Problem (TTDP) | 26 |
| 3.1.2 | Route Planning Problem | 28 |
| 3.1.3 | Tourist Trip Design Problem with Route Planning . . | 28 |
| 3.2 | Solution | 29 |
| 3.2.1 | POI Recommendation using Orienteering Problem with Time Windows (OPTW) | 29 |
| 3.2.2 | OPTW with Transfer Patterns | 37 |
| 3.2.3 | Validating POI List | 37 |
| 3.2.4 | Handling Infeasible Trip Plans | 40 |
| 4 | Implementation | 42 |
| 4.1 | Datasets | 42 |
| 4.1.1 | Izmir Bus Network Data | 42 |
| 4.1.2 | POI Data | 44 |
| 4.2 | Architectural overview | 45 |
| 4.3 | Description of components | 45 |
| 4.4 | Android Client | 46 |
| 4.5 | Instantiation of the system | 46 |
| 4.6 | Simulator | 47 |
| 4.7 | eJabberd Server | 48 |
| 4.7.1 | Filtering Module | 48 |
| 4.7.2 | Broadcasting Module | 49 |
| 4.8 | Back-end Server | 49 |
| 4.8.1 | POI Recommender | 50 |
| 4.8.2 | Route planner | 51 |
| 5 | Evaluation | 52 |
| 5.1 | Setup | 52 |
| 5.2 | Execution Times | 53 |
| 5.2.1 | Performance of Route Planner | 53 |
| 5.2.2 | Performance of Poi Recommender | 54 |
| 5.2.3 | Overall Performance of the System | 54 |
| 5.3 | Recommendations Evaluation | 55 |
| 5.3.1 | Handling Infeasible Trip Plans Results | 57 |
| 6 | Conclusion and Future Work | 60 |

List of Figures

| | | |
|-----|---|----|
| 1.1 | Personalized Electronic Tourist Guide ¹ | 1 |
| 1.2 | VAG - Example Route Planner from a developed country . . . | 2 |
| 1.3 | Izmir - the third most populous city of Turkey ² | 3 |
| 2.1 | Time-Dependent Graph Example with 2 Stations and 2 Lines | 10 |
| 2.2 | Cost Function from Station A to B in Figure 2.1 | 10 |
| 2.3 | Time-Expanded Graph Example | 12 |
| 2.4 | Query Graph Example for the transfer pattern $\{A, B, D\}$. . | 19 |
| 3.1 | OPTW Example | 34 |
| 3.2 | OPTW Insertion Example | 34 |
| 3.3 | OPTW Shake Example | 36 |
| 3.4 | Example Trip - Start and End locations are represented as user icons | 40 |
| 4.1 | System Architecture | 45 |
| 4.2 | Android Client - Label Selection | 47 |
| 4.3 | Android Client - Request Answer | 48 |
| 4.4 | MySQL Database Tables | 49 |

List of Tables

| | | |
|-----|--|----|
| 2.1 | Example Direct-Connection Table | 17 |
| 2.2 | Example Line Lists | 17 |
| 4.1 | Lines Sheet - Example Line 101 | 43 |
| 4.2 | Stations Sheet - Stations of Example Line 101 | 43 |
| 4.3 | Schedules Sheet - Schedule of Example Line 101 | 43 |
| 5.1 | No Delay Scenario | 56 |
| 5.2 | Random Delay Scenario | 57 |
| 5.3 | Repairing Infeasible Trip Plans, No Delay Scenario | 58 |
| 5.4 | Repairing Infeasible Trip Plans, Random Delay Scenario | 59 |

Chapter 1

Introduction

Normally, if tourists visit a place for the first time, they do not know where to visit and what to do in that place. The tourists look for information and try to design a trip plan. This is a time-consuming task. They end up wasting lots of time or go to a local tourist information center and get some help. In local tourist organizations, staff provides up-to-date point of interest (POI) recommendations depending on the interests of the tourists.



Figure 1.1: Personalized Electronic Tourist Guide¹

A Personalized Electronic Tourist Guide (PET) should be able to perform the tasks that local tourist information centers accomplish. A PET should recommend the most interesting POIs for the tourists. Meanwhile, there are some attributes that should be taken into account such as trip duration, opening and closing times of the POIs, schedule of the local transportation units. Furthermore, generating travel plans between POI visits is

¹<http://dbis.informatik.uni-freiburg.de>

an optional task of a PET.

Moreover, generating travel plans task in developed countries has been widely studied and there are nearly no delays in the transportation schedule. Unfortunately, this is not the case in developing countries where there are lots of events happening in the transportation network such as traffic jams, accidents, many construction works. In addition, these countries lack well-structured transportation networks. Because of these reasons, there is a need for accurate real-time route planners.

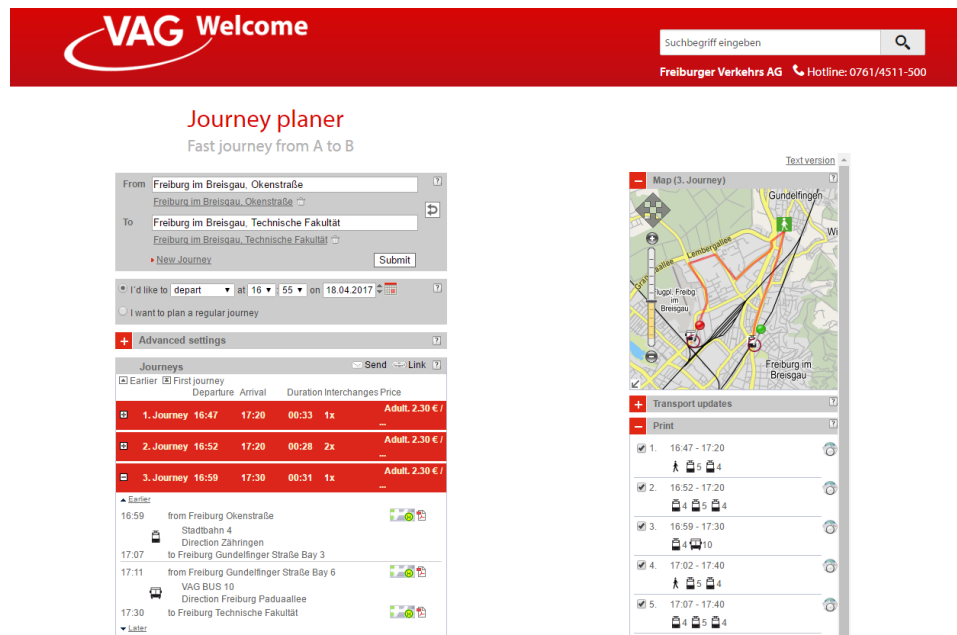


Figure 1.2: VAG - Example Route Planner from a developed country

Furthermore, if we look from the tourist trip aspect again, the tourists have limited amount of time. Wasting time on delays might make their trip plans unachievable and miss some of the attractions resulting in undesired experiences. This increases the necessity of real-time route planning even when schedules are available.

1.1 Initial goal and contributions

The goal of the thesis is to create a system that produces personalized tourist trip plans with a sequence of POIs which can be visited in a pre-defined

time span. In addition to the sequence of POIs, the system should provide a public transportation plan to move from one POI to the next one. Public transportation plan should consider real-time positions of the transportation units. Furthermore, an Android Application should be implemented as a user interface. Finally, these personalized trip plans should be computed in a reasonable time so that the system can be used in real-life. The city that is picked for the project is Izmir (Turkey) which has a population of 2,847,691 (2014).



Figure 1.3: Izmir - the third most populous city of Turkey²

The main contributions of this thesis can be summarized as follows.

First, we implemented a personalized POI recommender that makes use of real-time and GPS-based transportation data for the travel plans and can be hence used in scenarios where delays are recurrent (developing countries). Even when delays are drastically affecting the trip plan, our approach makes it even possible to recompute the plan again.

The second contribution consisted in using Transfer Patterns Algorithm with the POI recommender, which to the best of our knowledge has not been previously done. Transfer Patterns Algorithm is an optimization technique for route planning in public transit networks.

Finally, the third contribution was the proposal of two strategies to fix a trip plan in case of delays or changes in the plan, e.g. when the user decides to stay longer in a location. Both strategies were proved to be applicable and perform a good job in keeping the overall trip plan score close to the original proposed plan.

²<https://www.flickr.com/photos/79971562@N00/3866997955>

1.2 Thesis outline

Chapter 2 gives background on how to model transit networks as graph and recommender systems. Additionally, the chapter provides information on related work.

Chapter 3 states the Tourist Trip Desing Problem along with Route Planning Problem and shows how the problem is solved.

Chapter 4 describes the architecture of the system and explains how components are implemented.

Chapter 5 shows how the evaluation was done for our work. We first reveal our setup for the evaluation. Then execution times for the different components of the system are given and analyzed. We define *feasible* trip plans as the plans which have visits that can be completed on time and *infeasible* if the visits can not be completed on time. In Section 5.3, we examine how many of the trip plans are feasible. For the infeasible plans, we execute the failing queries and generate the trip plans again with our simplistic solutions and see if the trip plans become feasible.

Finally, in the last chapter, conclusions and future work are presented.

Preliminaries

The goal of this work is to implement a PET by designing an algorithm which allows generating the point of interest (POI) recommendations for a user. The requirements of the problem a PET has to solve can be modeled as the Tourist Trip Design Problem (TTDP) [19]. In addition to this, we aimed at providing feasible route plans to make it possible for the user to move from one POI to the next one according to the plan. We assume that buses may have delays and do not always stick to timetables which are the cases in many developing countries. Before explaining our approaches and how these are integrated together to solve the main problem, we introduce the foundations.

The first section explains the foundations of transit route planning, which is the task of computing the most cost-effective route involving transportation. This section covers two main strategies to model timetable information: time-expanded model and time-dependent model. In addition, we explain optimization techniques and especially transfer patterns method.

In the second section, a categorization of recommender systems (RS) is presented. In our work, we focus on particular kind of recommender systems called knowledge-based recommender systems.

In the third section, we present some work related to POI recommendation combined with transit route planning. Different approaches have been proposed for transit route planning and designing tourist trip plans. We will explain them and point out the main differences with respect to our work.

2.1 Transit Route Planning

The aim of route planning is to compute the most cost-effective route by minimizing the distance traveled and/or time taken. To achieve this, the transportation network is modeled as a graph. In order to find travels with the most cost-effective routes, shortest path algorithms are used on top of the graph model.

There are two major types of route planner systems; in-vehicle and out-of-vehicle route planners. In-vehicle route planners combine route determination with real-time guidance to the driver of the vehicle on picking paths to reach the destination. Out-of-vehicle route planners give the user the possibility to plan one or more routes and makes it possible for him to choose the best one considering the source and target points of the user. This work focuses on out-of-vehicle route planning.

In addition to this kind of categorization, route planning can be divided into two categories; route planning in road networks and in public transit networks. In route planning in road networks, the network is modeled as a graph in which intersections are represented by nodes and the roads as edges. The model is designed for vehicles and also pedestrian traffic. In public transit route planning, public transportation network is modeled as a graph in which stations and stops are additionally represented as nodes. There exist models or works which combine different aspects of these two categories. Our work focuses on public transit route planning only.

The main goal of the route planner is to find one or more optimal routes between the source location and the destination location of the user. The optimality of a plan might depend on different factors such as:

- **Minimum time**
- **Minimum distance**
- **Minimum number of transfers, etc.**

We consider minimum time as the optimal routes. Hence, our problem for the route planning part in this thesis is the so-called earliest arrival problem.

Moreover, to provide optimum routes to the user, the route planner requires the timetable information that represents transportation network of

the city, region or area as input. Nowadays, the *General Transit Feed Specification* (GTFS) is the de facto standard format to serve public transportation networks for the route planner systems [37].

Input data that we used for this thesis is provided by the General Directorate of ESHOT [14], the public bus transportation corporation of the Municipality of Izmir (Turkey), on our request. The data is not structured considering GTFS or any other specification. Instead, data is provided as Microsoft Excel worksheets. More details about the dataset are provided in Section 4.1.1.

2.1.1 Graphs and Dijkstra's Shortest Path Algorithm

Graphs

Graphs are the most common data structure used in route planning [2]. The different elements from a timetable such as stations, stops, arrivals, departures, transfers are modeled as nodes in the graph. There are multiple ways to store information in graphs. The ones used in this thesis will be explained next in the subsequent sections.

A graph $G = (V, E)$ is defined by a set of *nodes* or *vertices* V and a set of *edges* E . Every edge $e \in E$ connects two nodes, the *source node* $u \in V$ and *target node* $v \in V$, $e = (u, v)$. For all $e = (u, v)$, we can call v as the neighbor of u . A graph is undirected if $(u, v) \in E \implies (v, u) \in E$ holds for every edge $e \in E$ and directed otherwise. A *weighted graph* associates a weight (cost) with every edge in the graph.

A path in G is a sequence of edges $\langle e_1, \dots, e_k \rangle$ and can be represented as a sequence of nodes $\langle u_1, \dots, u_{k+1} \rangle$ where u_i is the source node and u_{i+1} is the target node of e_i for all $1 \leq i \leq k$. This only works under the assumption that there is at most one edge connecting two nodes. The target node of e_i is the source node of e_{i+1} for all $1 \leq i \leq (k - 1)$. The source of path is the first node in the sequence u_1 and the target of path is the last node in the sequence u_{k+1} .

The cost of a path is defined as the sum of the edge costs in the path. The *shortest path* between the source node u and the target node v is the path which starts from u and ends at v and has the least path cost among all paths.

Dijkstra's Shortest Path Algorithm

Since we need to solve earliest arrival problem as mentioned in Section 2.1, we find the shortest path using the time to move from one node to another one as the edge cost. Dijkstra's shortest path algorithm [13] was the first algorithm implemented to find shortest paths in the graphs with positive edge costs and the algorithm is still used widely in many graph problems. Dijkstra's algorithm solves the one-to-all nodes shortest path problem. Algorithm 1 shows the pseudo-code of Dijkstra's algorithm. The input of the algorithm is a weighted graph and a source node. The algorithm returns the cost and the sequence of nodes in optimal paths from the source node as output.

The algorithm maintains a priority queue Q and a list $\text{dist}[\]$ with the distances to all nodes $u \in V$ from the source node s . At the beginning, all distances are initialized to infinity except $\text{dist}[s] = 0$. In each iteration algorithm dequeues the node with minimum distance u and then checks distances to each neighbor node v . The distance to v is calculated by $\text{cost}[v] = \text{dist}[u] + \text{length}(u, v)$ and if this value is less than $\text{dist}[v]$, $\text{dist}[v]$ is updated by the value. This step is called *relaxation*. Q is also updated in relaxation-phase.

There is another list $\text{prev}[\]$ for keeping track of the *previous* nodes. Initially, this list is empty for every node. The previous node of v is node u at relaxation step. The list $\text{prev}[\]$ can be used for traversing from any target node to source node s .

The complexity of the algorithm depends on the type of the priority queue. The complexity of the algorithm is $\mathcal{O}((|V| + |E|) \log |V|)$ when binary heaps [43] are used. When Fibonacci heaps [18] are used, the complexity of the algorithm improves to $\mathcal{O}(|E| + |V| \log |V|)$. We used Fibonacci heaps for our shortest path algorithms.

2.1.2 Time-Dependent Model

The time-dependent model was introduced for modeling timetable information [8]. In this model, each station or stop (for simplicity, term 'station' will be used alone from now on instead of both) in the transportation network corresponds to one node in the graph model. There exists an edge

Algorithm 1 Dijkstra's Shortest Path Algorithm

```
1: function DIJKSTRA(Graph  $G$ , source  $s \in V$ )
2:
3:    $Q \leftarrow$  priority queue contains all nodes
4:
5:   for all  $u \in V$  do
6:      $\text{dist}[u] \leftarrow \infty$ 
7:      $\text{prev}[u] \leftarrow \text{null}$ 
8:      $Q.\text{update}(u, \infty)$ 
9:   end for
10:
11:    $\text{dist}[s] \leftarrow 0$ 
12:    $Q.\text{update}(s, 0)$ 
13:
14:   while  $Q$  is not empty do
15:      $u \leftarrow Q.\text{dequeueMin}()$   $\triangleright$  node with the least distance
16:
17:     for all neighbor  $v$  of  $u$  do
18:        $\text{alt} \leftarrow \text{dist}[u] + \text{length}(u, v)$ 
19:       if  $\text{alt} < \text{dist}[v]$  then
20:          $Q.\text{update}(v, \text{alt})$   $\triangleright$  update distance value
21:          $\text{dist}[v] \leftarrow \text{alt}$ 
22:          $\text{prev}[v] \leftarrow u$ 
23:       end if
24:     end for
25:   end while
26:
27:   return  $\text{dist}[\ ]$ ,  $\text{prev}[\ ]$ 
28: end function
```

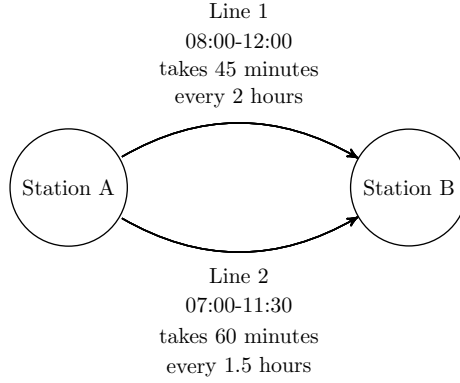


Figure 2.1: Time-Dependent Graph Example with 2 Stations and 2 Lines

from u to v if there is at least one elementary connection i.e. there is no other stop in-between, for corresponding stations. For each edge, timetable information such as departure and arrival times from the source station to the target station are attached as a cost function. These cost functions take a time t as input (departure from that station) and return the earliest possible arrival time for that elementary connection as output. The cost function for each edge $e = (u, v)$ can be denoted as $cost(u, v, t)$, where u is the source station, v is the target station and t is travel starting time. A minimal time-dependent model example with stations A and B is given in Figure 2.1. Resulting cost function from station A to B is given in Figure 2.2.

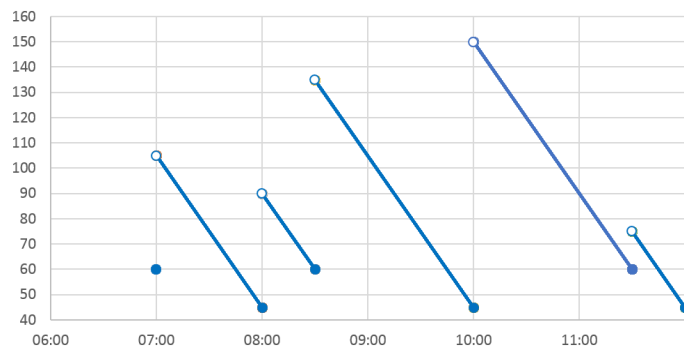


Figure 2.2: Cost Function from Station A to B in Figure 2.1

For station-to-station queries, Dijkstra's algorithm can be applied by choosing the source station, the target station as the parameters and using

the cost function instead of $length(u, v)$ in Algorithm 1.

This model allows one to store the timetable information in a few number of nodes in the graph, therefore it is a compact representation. Moreover, there are more sophisticated extensions of the time-dependent model which consider transfer buffers i.e. minimum time needed to change a vehicle [33] or walking between stations [15].

2.1.3 Time-Expanded Model

The following definitions were taken from [1] and adapted to our problem.

In the time-expanded model, there are three kinds of nodes; *arrival*, *departure* [31, 36] and *transfer* [32, 33, 34] nodes which represent an event of a transportation unit at a certain station and time.

For each pair of stations A and B for which there is available public transportation, there exists a departure node $A_{d@t_1}$ at station A with the departure time t_1 and an arrival node $B_{a@t_2}$ at station B with the arrival time t_2 . Furthermore, there exists an edge between these two nodes $A_{d@t_1} \rightarrow B_{a@t_2}$ which represents the action of getting into the vehicle from A to B . If B is not the final station of the trip, the transportation unit will continue the trip to the next station. To model that, there exists the next departure node from B , $B_{d@t_3}$ with an edge $B_{a@t_2} \rightarrow B_{d@t_3}$.

A transfer represents the event of changing from a vehicle to another one when travelling. To model transfers, there exists a transfer node for each departure node with the same time, $B_{t@t_3}$ ¹. The transfer node is connected to the graph by adding an edge from transfer node to departure node $B_{t@t_3} \rightarrow B_{d@t_3}$ and another edge from arrival node (predecessor of $B_{d@t_3}$) to transfer node $B_{a@t_2} \rightarrow B_{t@t_3}$. Moreover, we put edges $B_{t@t} \rightarrow B_{t@t'}$ from a transfer node to the next transfer node at B that comes next in the ascending order of transfer times. In this way, transfer nodes of the same station form a sequence of connected nodes called *waiting chain*. Thanks to waiting chain nodes, connections to the trips of the same station at later times are allowed.

We can give the graph in Figure 2.3 as an example. We should note that this example is not related to the example in Figure 2.1. There are two

¹t (transfer) should not be confused with t (time)

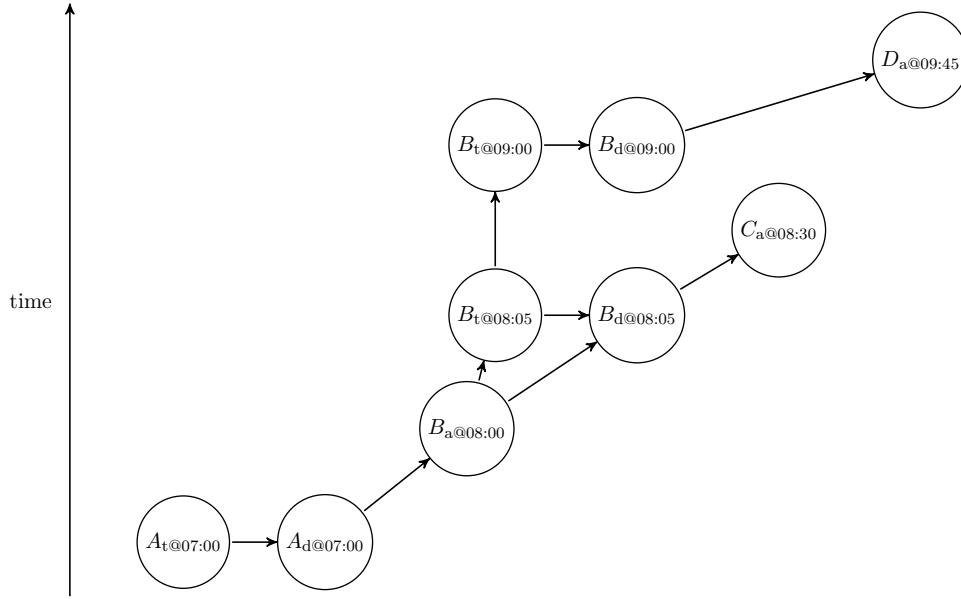


Figure 2.3: Time-Expanded Graph Example

lines in the graph, first line starts from station A visits B and then ends at C . The second line starts from station B and ends at D . The nodes are arranged vertically by their time. In the example, nodes $B_t@08:05$ and $B_t@09:00$ form a waiting chain at station B .

The times in the graph model increase unbounded from time 0 (midnight of a day) and a modulo of 24 hours is applied. As a result, all calculations are done for a day time, days of the week are not considered in the problem. Furthermore, the cost function represents the duration of trip only, and duration is the difference in time between the arrival and the departure time. Suppose a trip starts at the node $A_d@t_1$ and ends at the node $B_a@t_2$. The cost and the duration of the trip would be $t_2 - t_1$. In addition, we say cost c_1 *dominates* c_2 if $c_1 \leq c_2$.

2.1.4 Optimization Techniques

Optimizing the performance is a popular problem in route planning since transportation networks are growing every year. Because of this, there is a demand for efficient algorithms.

We have already defined the basic approaches of route planning; graphs

in general, Dijkstra’s shortest path algorithm and 2 most commonly used graph models: time-dependent model and time-expanded model. One can think that this knowledge might be enough to solve basic route planning problems. In our case, we implemented a simple route planning algorithm using these building blocks first. But execution times were too high as more than 20 seconds needed for finding only 1 shortest path. This was because our graph model for a whole day was large, it contained more than 3 million nodes. Therefore, we needed an optimization technique.

In the last fifteen years, many speed-up techniques are implemented. These optimization techniques can be divided into following groups [2]:

- **Goal-Directed Techniques** aim to guide the search towards the target and avoid scanning nodes in the other directions.
- **Separator-Based Techniques** aim to use small separators as in planar graphs [29, 11, 16].
- **Hierarchical Techniques**’ goal is to exploit the inherent hierarchy of road networks.
- **Bounded-Hop Techniques** try to precompute distances between node pairs and add shortcuts to the graph.
- **Combinations** aim to exploit different graph properties for additional speedups.

The new faster algorithms mostly have a precomputation step independent of queries. The precomputation step helps algorithms find solutions faster by doing required calculations in before. In our work, we applied an optimization technique developed for public transit networks called Transfer Patterns which requires a precomputation step.

Transit Routing using Transfer Patterns

Transfer Patterns Algorithm is an optimization method for fast routing in very large transportation networks [1]. The algorithm is based on two key observations. First, many shortest paths share the same *transfer pattern*. Second, *direct connections* without change of a vehicle can be looked up very

quickly. Using these observations, the respective data is precomputed. At query time, precomputed data is used. As a result, there are fewer nodes in the *query graph* scanned at the shortest paths algorithm. The main advantage of using this speedup technique is it drastically reduces execution times for queries from seconds to 1 millisecond. Furthermore, the technique supports extensions such as walking between stations, traffic days, multi-criteria costs.

Basic Concepts of Transfer patterns Before we introduce steps of the algorithm, we should define basic concepts. The following definitions were taken from [1] and adapted to our problem setting. Moreover, the **time-expanded** graph is used for the precomputation step. For query graph evaluation step, the **time-dependent** graph is used. For both of the steps, both of the models can be used but using them in this way is the most efficient method [1].

For the stations A and B , a **station-to-station query** at time t_1 is denoted as $A_{@t_1} \rightarrow B$. We use help of additional nodes, a source node S and a target node T to find optimal paths between stations A and B . For the time-expanded graph, we follow these steps:

1. Get the first transfer node $A_{t@t_2}$ with $t_2 > t_1$.
2. Extend the graph by a source node S .
3. Add an arc from S going to $A_{t@t_2}$ with the cost $t_2 - t_1$
4. Extend the graph by a target node T .
5. For all arrival nodes of B , add arcs outgoing to T with 0 cost.

Feasible connections for the query are the paths from S to T .

Optimal connections for the query are the paths from S to T , which are not dominated (Section 2.1.3) by any other path.

6. Because of the waiting chain, there might be multiple optimal connections and the one that departs latest is picked [1]. The *result* of the query is the remaining optimal connection.

For any path, consider the subsequence of nodes built by:

- the first node,
- each arrival node whose successor is a transfer node,
- the final node.

The sequence of stations of these nodes is the **transfer pattern** of the path.

Before explaining the precomputation step, we should also define the *optimal set of transfer patterns* for a pair of stations, A and B . For all possible $A_{@t} \rightarrow B$ queries at some time t , there is an optimal connection and a transfer pattern TP for that connection. Suppose a set S_{TP} of transfer patterns formed from all TP s considering all possible times. Most probably, there will be identical transfer patterns in S_{TP} . The **optimal set of transfer patterns** is the set S_{TP} with duplicate patterns removed.

To illustrate, we can use the time-expanded graph in Figure 2.3. For the station pair (A, D) , there is only 1 feasible connection, therefore that connection is the optimal connection. The path of that connection starts from $A_{t@07:00}$, passes through $B_{a@08:00}$ (arrival node whose successor is a transfer node) and ends at $D_{a@09:45}$. Therefore, the sequence of stations $\{A, B, D\}$ is the only transfer pattern for (A, D) pair. In a more complex example, there could be other optimal connections for different times. In that case, there would be more transfer patterns for (A, D) pair.

We first explained which type of graphs are used for the different stages of the algorithm. Then we defined station-to-station queries. Finally, we presented the transfer patterns concept. Now we can continue by describing the steps of the algorithm.

Transfer patterns precomputation In the precomputation step, the optimal set of transfer patterns for every pair of stations is computed by using station-to-station queries. The result saved to a file to be used for building query graphs. To precompute respective data, for each station A , following steps are applied:

1. Get all transfer nodes of A as the set of source nodes of Dijkstra's shortest path algorithm. All of the nodes in the set of source nodes have a distance of 0. Then execute the shortest path algorithm.
2. For all B stations,
 - Order arrival nodes of B by distinct times $t_1 < t_2 < \dots$, resulting in arrival nodes a_1, a_2, \dots
 - Let T_{i-1} and T_i be the travel times of the shortest paths to a_{i-1} and a_i respectively.
 - If $T_{i-1} + (t_i - t_{i-1}) \leq T_i$, replace the travel time T_i at a_i by $T_{i-1} + (t_i - t_{i-1})$ and make a_{i-1} the predecessor on the shortest path to a_i .
3. Trace back all paths for the arrival nodes in Step 2 and get transfer patterns by traversing them. Write the transfer patterns to the file.

Transfer patterns precomputation is quadratic itself with respect to the vertices in the time-expanded graph. This might lead precomputation step to take too much time for large networks. Therefore speedup techniques for this part are introduced but not used in this work.

Direct-connection queries Direct-connection queries allow fast look-ups for connections without transfers. Since we use the time-dependent graph for the query graph, every edge in the graph represents a direct connection. Thus, we can use direct-connection queries as cost functions of the shortest path algorithm.

For a *direct-connection query* $A_{@t} \rightarrow B$, the feasible connections are defined as before. The only exception is that connections with transfers are not allowed. Therefore all feasible connections are direct. The result of the direct-connection query is the optimal connection without transfers. There are two data structures introduced for evaluating direct-connection queries.

The first data structure, a **direct-connection table** is a table constructed using trip departure and arrival times for the stations. In other words, direct-connection tables are special representations of a timetable for fast schedule look-ups. To construct the table, trips are precomputed by calculating all departure and arrival times for every station on its line.

Table 2.1: Example Direct-Connection Table

| Line 121 | A | B | C | D |
|------------|-------|-------------|-------------|-------------|
| Trip@06:00 | 06:00 | 06:01 06:02 | 06:04 06:04 | 06:07 06:08 |
| Trip@06:14 | 06:14 | 06:15 06:16 | 06:18 06:18 | 06:21 06:22 |
| Trip@06:28 | 06:28 | 06:29 06:30 | 06:32 06:32 | 06:35 06:36 |

Table 2.2: Example Line Lists

| | | | | |
|-----|---------|---------|----------|----------|
| A | L121, 1 | L123, 1 | L326, 26 | L400, 14 |
| B | L101, 1 | L121, 2 | | |
| C | L121, 3 | L125, 3 | L,303, 3 | |
| D | L121, 4 | L125, 4 | L,326, 2 | L350, 15 |

After that, trips are grouped by lines. These grouped trips should have the same travel times between any two stations and should not overtake each other. Then for each line, a table is formed using the stations since all trips on the line share the same sequence of stations. Finally, the trips of each line are sorted considering start time. Table 2.1 shows the direct-connection table for the line 121. Creating tables this way lets us have a more compact representation for the timetable look-ups.

The second data structure, a **line list** is a sorted list of lines for a specific station. For each station, lines incident to it are gathered, then sorted. The position of the station on the line is attached to each line. Let us consider the station C in Table 2.1. Since C is at the third position in the line, the name of the line with the position 3 (L121,3) is added to the line list of C (Table 2.2).

After having computed the direct-connection tables and line lists, a direct-connection query can be answered. For a direct-connection query $A_{@t} \rightarrow B$, the answer is found by intersecting the line lists of A and B . Then, a list of feasible trips is formed by looking up direct-connection tables of the lines that A appear before B . Among the feasible trips, the trip with the earliest arrival is selected as the answer.

Let us consider the direct-connection query $A_{@06:10} \rightarrow D$ as an example. The positions 1 and 4 are found on Line 121 by using line lists of A and D . Since there is no other line found, we only check the table of Line 121. The

trip that arrives earliest starts at 06:14 and arrives the destination at 06:21.

Query graph construction and evaluation At this point, we assume the transfer patterns for all station pairs are computed. Also, we assume direct-connection tables and line lists are constructed.

A station-to-station query of $A_{@t} \rightarrow B$ is evaluated by first building a query graph as follows:

1. Fetch precomputed transfer patterns for station A .
2. Among these patterns, get the ones ending with station B . Remaining patterns will be transfer patterns of the pair (A, B) .
3. Assume a transfer pattern has m successor nodes with labels C_1, \dots, C_m . Add the arcs $(C_1, C_2), \dots, (C_{m-1}, C_m), (C_m, B)$ to the query graph. Do not add the arcs that are already in the graph.
4. Do the previous step for all selected patterns.

After the query graph is constructed, the query is evaluated by executing time-dependent Dijkstra search on the graph. For cost functions of each station pair in the graph, direct-connection queries are used. With correctly answered direct-connection queries, the result is optimal since all possible optimal paths connecting stations A and B are already in the query graph.

Again, let us consider the example in Figure 2.3. For the station pair (A, D) , we have transfer pattern of $\{A, B, D\}$. For a station-to-station query $A_{@t} \rightarrow D$, we build the query graph as in Figure 2.4. Since station C does not appear in any of the optimal connections, it is not in the transfer patterns and it is not in the query graph. As a result, we have fewer nodes and query graph evaluation evaluated quicker.

Izmir bus network has more than 7000 stations. In a station-to-station query, thanks to transfer patterns, our query graphs have less than 50 stations. The performance of our system can be seen in Chapter 5.

Dealing with Delays To keep Transfer Patterns Algorithm up-to-date, the direct connection data structure should be updated in real-time. For a single specific trip, the bus delay is calculated. Then, the trip's line is found

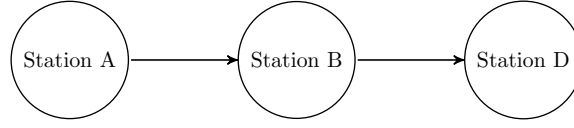


Figure 2.4: Query Graph Example for the transfer pattern $\{A, B, D\}$

and stop times are changed. If the trip does not overtake previous or next trip, the process is over. Otherwise, the trip is removed and a new trip is inserted to the same line instead. This operation is repeated for all active trips.

The optimality of Transfer Patterns Algorithm in the presence of delays is tested with experiments in the work of Bast et. al. [4].

2.2 Recommender Systems

Recommender systems are special kind of information filtering systems that provide suggestions for items that are most likely of interest to a particular user [9, 24]. These suggestions help users to decide which goods, services or information to purchase or consume. The importance of good recommenders increases every year as the number of possible choices increase.

There are 3 kinds of elements in the recommender systems:

Item

Item is the general name for the objects that are recommended by the system. Items might be songs, movies, mobile phones or even jobs, financial investments. In our work, items are the POIs in Izmir. Some of the properties they have are opening-closing hours, ratings, coordinates and times need to visit them.

For suggestions, items should be characterized by their value or utility to the user. The value of the items might be positive if the item is beneficial to the user, and negative if the item is not necessary. Recommenders aim is to suggest items that have the highest value to the user.

User

Users might have various goals and attributes. Recommenders try to exploit user information and personalize recommendations as much as possible. There are multiple ways of structuring user information depending on the recommendation approaches.

In one of the most popular techniques, *collaborative filtering*, recommenders predict the items that the user might like based on similarity of the user behavior to other users'. In this approach, it is assumed that users who agreed in the past will agree again in the future because they have similar tastes on the items. Users can also be characterized by their behavior patterns such as the way they use websites or search their patterns.

Transactions and Feedback

Transactions are recorded interactions of the users. Every feedback of users can be recorded for future item predictions. We can give an example of a movie rating web-site where items are movies. Users of the web-site rate the movies that they watched on a numerical rating scale such as 1 to 10. These interactions can be recorded by the system and used for future recommendations. As already explained for the users, these interactions may be user behavior patterns such as movies searched in a session. These movies can be related somehow and a recommender might use this information when recommending movies.

2.2.1 Recommendation Techniques

Most common approaches are:

- **Collaborative Filtering:** As mentioned before, recommenders which use this technique try to take advantage of the users with similar tastes [22]. Similarity is computed based on the similarity of users' transactions for example similarity of users' movie ratings.
- **Content-Based Filtering:** is another approach where recommenders filter items based on their features and a profile on the user's prefer-

ence. The system tries to learn items that are similar to the ones user already liked and recommend them [30].

- **Knowledge Based Recommender Systems:** will be explained in the next section for more details.
- **Demographic:** In this approach, recommenders provide recommendations based on the demographic profile of the user [6].
- **Hybrid Recommender Systems:** These systems combine approaches that we mentioned so that they tackle the weaknesses and problems occur when using only one technique.

2.2.2 Knowledge-Based Recommender Systems

Whereas some recommendation techniques work well for recommending products like movies, songs, etc., but they might be insufficient when it comes to recommending more complex recommendations such as tourist trip plans or financial services. One of the reasons of that is gathering ratings and user behavior can be very challenging in these complex domains. Another point is that out-of-date item ratings or preferences might result in recommendations which are not up-to-date. As a result, users of these recommender systems would not be satisfied.

Knowledge-based recommender systems make suggestions based on specific domain knowledge about a user's needs and preferences. Therefore, these systems have more tailored recommendations thanks to the knowledge sources that are not exploited by other approaches. Since requirements are obtained at runtime, these systems do not have cold-start problems. However, without any learning methods used, their performance tend to be surpassed by other techniques, since others benefit from recorded ratings and patterns.

Knowledge-based recommenders can be grouped into two different categories. *Case-based* recommenders [7] solve new problems by transferring and adapting solutions that were already used for similar problems previously. New recommendations can be made by transferring and adapting solutions that were used in the past for similar cases.

On the other side, in *Constraint-based* recommenders [17], recommendation is viewed as a process of constraint satisfaction. Some of the constraints are obtained from the user at the beginning of the process and the rest of the constraints come from the item domain. Items which satisfy these constraints are considered to be relevant recommendations. Our work falls into this category.

We obtain travel constraints from the user and use them to filter the POIs. Then we will give the user a set of POIs which satisfy the constraints as an output while we try to maximize user satisfaction.

While using travel constraints with POI information, we needed a ranking strategy. In need of rankings, many recommender systems use retrieval models such as *Vector Space Model*.

Vector Space Model

POI input file is described in details in Section 4.1.2. Let us consider *labels* feature of the POIs. To combine POI labels with users' interest, we ask the users to pick labels they are interested in. For ranking POIs with respect to these interests, our recommender uses Vector Space Model (VSM).

VSM is an algebraic representation of the text documents. In VSM, each document is represented by a vector in a n -dimensional space. In the model, each dimension corresponds to a term from given text document. In our case, POIs are the documents and labels are the terms. Suppose there are N number of POIs, let $P = \{P_1, P_2, \dots, P_N\}$ denote a set of POIs and $L = \{L_1, L_2, \dots, L_N\}$ denote a set of labels. Each POI P_j is represented as a vector in a n -dimensional vector space, $\vec{P}_j = \langle w_{1j}, w_{2j}, \dots, w_{nj} \rangle$ and w_{kj} is the weight for the label L_k for the POI P_j .

The labels that user is interested in can be represented as a vector as well $q = \langle w_{1q}, w_{2q}, \dots, w_{nq} \rangle$. Relevance rankings can be calculated using document similarities. This is done by comparing the deviation of angles between each document vector and the query vector. It is easy to calculate *cosine* of the vectors and find the answer with *cosine similarity*. The cosine of the angle between the vectors:

$$\cos(P_j, q) = \frac{P_j \cdot q}{\|P_j\| \cdot \|q\|} \quad (2.1)$$

Using the cosine similarity formula (2.1), similarity can be calculated as:

$$\text{sim}(P_j, q) = \frac{P_j \cdot q}{\|P_j\| \cdot \|q\|} = \frac{\sum_{i=1}^n (w_{i,j} w_{i,q})}{\sqrt{\sum_{i=1}^n (w_{i,j}^2)} \sqrt{\sum_{i=1}^n (w_{i,q}^2)}} \quad (2.2)$$

Using similarity between POIs' labels and users' interests (2.2), a ranking can be done between POIs.

2.3 Related Work

We will examine related work in two different sections since our work combines two topics and benefits from both. First, we will explain the contributions on route planning in public transit networks, then we will explain works address the problem of tourist trip design.

2.3.1 Route Planning in Public Transit Networks

The groundwork for route planning is established by E. Dijkstra in 1959 with his shortest path algorithm [13]. Many state of the art algorithms still use derivations of this algorithm for their graph models. One of these algorithms is a goal-directed technique called *A** algorithm which uses heuristics that improves running time while preserving optimality. Some of the more advanced shortest path finding techniques introduced latterly are *Arc Flags* [26], *Contraction Hierarchies* [21] and *SHARC* [5]. These examples have significantly improved execution times and they are currently being used for the route planning applications.

As explained in Sections 2.1.2 and 2.1.3, there are two important graph models for route planning in public transit networks. In *Time-expanded* model a graph is constructed by using timetable information. In the graph, every node corresponds to a specific time event at a certain station. The basic versions of the model are introduced and discussed by Möhring in [31] and Schulz et al. in [36]. Müller-Hannemann and Schnee extended the model with multi-criteria pareto search including minimum change times [32]. Pyrga et al. used *transfer* vertices in [34]. Delling et al. reduced the query times by using less number of arcs for the search [12].

Time-dependent model has been used for instance in the work of Brodal and Jakob [8]. They proposed using time-dependent network instead and

proved by their theoretical analysis that their approach is more efficient than the time-expanded approach. As mentioned before, this model uses stations as nodes in the graph. Disser et al. [15] introduced the concept of walking between stations to the model. Pyrga et al. [33] extended the model by enabling minimum change times and in the same work they found out that the time-expanded model is more robust for modeling more complex scenarios whereas the time-dependent approach shows a better performance.

Speedup techniques have been developed for both graph models. Bast et al. developed a technique called *Transfer Patterns* [1] which consists in finding optimal journeys and detecting the sequence of stops where transfers occur. These patterns are computed at precomputation time and saved to be used later at query time. By using precomputed transfer patterns, query times highly improve. For improving precomputation times, they used the concept of hub-stops. Later on Bast et al. [4] examined the robustness of their state-of-the-art speed-up technique. They found out that the algorithm still almost always finds the optimal solution even in different delay scenarios. In the *Scalable Transfer Patterns*, Bast et al. [3] reduced precomputation time and space consumption for large networks. Finally, Bast et al. [2] report recent advances in algorithms for route planning in transportation networks in a survey.

2.3.2 Tourist Trip Plan Generation

In the tourist trip design problem (TTDP) [41] the main goal is to select POIs that maximize tourist satisfaction. Solutions must take into account parameters and constraints of the tourist. The model can be modeled as the traveling salesperson problem with profits (TSPP). The difficulty of solving TSPP comes from two conflicting objectives, collected profit should be maximized while minimizing the travel cost [28]. For POIs, profit can be defined as the degree of satisfaction of the tourist which is the score of visited POI.

The orienteering problem (OP)(also known as the selective traveling salesperson problem) is introduced by Tsiligirides [38] which is a single-criterion variant of TSPP. OP tries to find the tour that maximizes the profit while keeping the travel cost under a value. The most basic version of TTDP corresponds to the OP.

An extension of OP is the OP with time windows (OPTW) where the opening and closing hours of POIs are considered as time windows. Kantor and Rosenwein [27] were the first to solve OPTW by a straightforward insertion heuristic. They used depth-first search trees that construct partial tours, along with the heuristic. The infeasible routes are abandoned if they are not likely to give the best total score. Righini and Salani [35] introduced an algorithm where they apply bi-directional dynamic programming which gives the exact solutions to OPTW. There exist different approaches to solve OPTW, such as in the work of Verbeeck et al. [42] solved the problem using an ant colony system.

The Team Orienteering Problem was defined by Chao et al. [10] as the extension of OP where the number of tours is increased from single to multiple. The TOP with time windows (TOPTW) was introduced by Vansteenwegen [39]. Later, the Iterated Local Search (ILS) heuristic proposed by Vansteenwegen et al. [40] became the fastest known algorithm for TOPTW. We used a similar version of the ILS heuristic for solving OPTW. ILS heuristic will be discussed in the Section 3.

The applications of the TTDP can be called Personalized Electronic Tourist guides (PETs). PETs work as knowledge-based recommender systems and derive personalized tourist routes to users. The work of Garcia et al. [19] can be considered an example of PET. They used ILS heuristic for their TOPTW which is implemented for the city of San Sebastian (Spain).

Problem Formalization

Tourist trip design problem (TTDP) is modeled in our work as the orienteering problem with time windows (OPTW). In the OPTW, a set of locations (POIs) each with a score, a service time and a time window is given. The goal is to maximize the sum of the collected scores considering problem constraints. As already showed in the work of Vansteenwegen et. al [40], it takes too much time to solve the problem in real-time using public transportation directly for the OPTW solution. Moreover, in the work, they used the city of San Sebastian for the experiments and San Sebastian is approximately 10 times smaller than Izmir. As a result, execution times might probably be unsatisfactory for our project. Therefore the problem is solved in 2 fine-grained steps.

3.1 Problem statement

3.1.1 Tourist Trip Design Problem (TTDP)

Notation

The following definitions were taken from [40] and adapted to our problem setting.

A visitor is a tourist and the user of the program. The visitor wants to have a trip plan. A set of n points-of-interest (POI) P_1, \dots, P_n , is given with

each location having an index $i = 1, \dots, n$, a score $Score_i$, a visiting time T_i which is the time a visitor typically spends visiting the POI and a time window $[O_i, C_i]$ with opening time O_i and closing time C_i . The starting point and the end point of every tour are fixed. Time needed to travel from P_i to P_j , c_{ij} , is known for all POIs. Available time of the tourist is limited to a given time budget T_{max} . s_i represents the starting time and $s_i + T_i$ represents the ending time of the visit at P_i .

Furthermore, $x_{ij} = 1$ if a visit to P_i is followed by a visit to P_j and 0 otherwise; $y_i = 1$ if P_i is visited and 0 otherwise. M is a large constant. Variables $Score_i$, T_{max} , s_i , x_{ij} , y_i are specific to a single user.

Problem

The aim of OPTW is to determine a route r , limited by T_{max} , maximizing the total collected score. Here are the problem constraints:

$$\text{Max} \sum_{i=2}^{n-1} Score_i y_i \quad (3.1)$$

(3.1) is the objective function and it maximises the total score.

$$\sum_{j=2}^{n-1} x_{1j} = \sum_{i=2}^{n-1} x_{in} = 1 \quad (3.2)$$

Constraint (3.2) guarantees that a tour starts from P_1 and ends at P_n .

$$\sum_{i=1}^{n-1} x_{ik} = \sum_{j=2}^n x_{kj} = y_k \quad , \quad \forall k \in [2, n-1] \quad (3.3)$$

$$s_i + T_i + c_{ij} - s_j \leq M(1 - x_{ij}) \quad , \quad \forall i, j \in [1, n] \quad (3.4)$$

Constraints (3.3) and (3.4) determine the connectivity and the timeline of each tour.

$$y_k \leq 1 \quad , \quad \forall k \in [2, n-1] \quad (3.5)$$

Constraint (3.5) makes sure that every POI is visited at most once.

$$\sum_{i=1}^{n-1} \left(T_i y_i + \sum_{j=2}^n c_{ij} x_{ij} \right) \leq T_{max} \quad (3.6)$$

Constraint (3.6) makes sure that total time spent is less than time limit.

$$O_i \leq s_i \quad , \quad \forall i \in [1, n] \quad (3.7)$$

$$s_i + T_i \leq C_i \quad , \quad \forall i \in [1, n] \quad (3.8)$$

Constraints (3.7) and (3.8) restrict the start and end of the service to the time window.

Result of the program should be a path of POIs (P_1, \dots, P_{k+1}) which maximizes total collected score, along with visitation times for the each POI, $((t_{1,s}, t_{1,e}), \dots, (t_{k,s}, t_{k,e}))$; $t_{1,s}$ denote visitation starting time and $t_{1,e}$ denote visitation end time for P_1 .

3.1.2 Route Planning Problem

Route Planning problem aims to find travel plans with the lowest cost. As stated in Section 2.1, we only use time t for the cost. In other words, we try to minimize travel time. The input of the problem is the timetable of public transportation system. In addition, there are travel plan queries between stations with starting times called *station-to-station queries*. These queries are used to find the travel plans between stations with the lowest costs (Section 2.1.4 for the detailed explanation). The travels with lowest costs are called *optimal connections* (Section 2.1.4). The output of the problem is the optimal connections for given query.

3.1.3 Tourist Trip Design Problem with Route Planning

This problem is the goal of the thesis. Using the output of Tourist Trip Design Problem, the path of POIs (P_1, \dots, P_{k+1}) with visitation times for each POI $((t_{1,s}, t_{1,e}), \dots, (t_{k,s}, t_{k,e}))$, a query $P_l @ t_{l,e} \rightarrow P_{l+1}$ is constructed, where $1 \leq l \leq k$. Result for each query is denoted as r_l where $1 \leq l \leq k$. After all, the main output should be the sequence of the results of the route planning queries, (r_1, \dots, r_k) .

3.2 Solution

3.2.1 POI Recommendation using Orienteering Problem with Time Windows (OPTW)

As stated in the work of Golden et al. [23], OP is NP-hard. Therefore, trying to solve OPTW in polynomial time can be too ambitious. Since a tourist trip plan designer should return the results in a few seconds, we solved the problem using a straightforward heuristic called iterated local search (ILS) [40]. ILS handles the problem in two different steps. First, there is an insertion step which adds a new visit to the trip. Insertion step is applied multiple times until the total trip score is maximized. Second, there exists a shaking step which is applied right after insertion step and it removes visits from the trip plan to escape from local optima. We will cover ILS heuristic in three stages. We will first explain insertion and shake steps, then finally we will be dealing with heuristic itself.

After we explain ILS, we introduce a *validation step*. We should mention that c_{ij} values are calculated using average travel times between POIs and known before ILS. Since recommended POI List is computed considering average travel times without considering up-to-date bus timetable and the current locations of the buses, timings between POIs might mismatch with average travel times. Scheduled bus times can let user arrive earlier or make them arrive later than expected. Furthermore, in the presence of delays, travel times can take even longer than predicted.

Arriving at a POI later than expected can lead the users of the system to spend less time at that place or even worse, completely miss the visit and make the whole trip plan a disaster. As a result, a validation mechanism needed. Infeasible trip plans can be detected and repaired thanks to the validation.

Insertion Step

In the insertion step, a new visit is added to the tour. Since time is limited by T_{max} , every time we do an insertion, we should verify that existing visits still fit in their time window. To avoid spending much time for checking other visits feasibility, variables that store times, *Wait* and *MaxShift* are introduced. The first variable *Wait* is for keeping the waiting time if the

arrival at a location a_i happens before the time window because a visit cannot start before opening time. Otherwise, if the arrival is after the opening time and before the closing time, $Wait$ has a value of zero.

The second variable $MaxShift$ is the maximum time that the visit can be delayed. Since we are dealing with time windows, we will shift the visit times towards the end of the time window of the visit. We do that in order to find the best score. $MaxShift$ is, of course, limited by the time window, otherwise visit becomes infeasible. Another condition that we check is the sum of $Wait$ and $MaxShift$ of the next location, $i+1$. This is because a visit can not be shifted more than the $MaxShift$ of the next visit. The minimum value of these two conditions equal to the value of $MaxShift$.

$$Wait_i = \max[0, O_i - a_i] \quad (3.9)$$

$$MaxShift_i = \min[C_i - s_i - T_i, Wait_{i+1} + MaxShift_{i+1}] \quad (3.10)$$

Thanks to these variables, we check the feasibility of the insertion step in constant time, not linear time. If we do not use these variables, we need to check the timings of the next visits for the current visit every time and this can be done in linear time.

Moreover, the total time consumption to insert visit j is defined as $Shift_j$. Again, $Shift$ is a variable that stores time. To determine $Shift_j$ between locations i and k , travel costs from i to j and from j to k should be added to wait time $Wait_j$ and visit time T_j . Then the travel cost from i to k should be subtracted. For feasibility, $Shift_j$ should be less than the sum of $Wait_k$ and $MaxShift_k$. This gives us 3.11. We should also remind that the constraint 3.8 should be satisfied for j .

$$Shift_j = c_{ij} + Wait_j + T_j + c_{jk} - c_{ik} \leq Wait_k + MaxShift_k \quad (3.11)$$

$Shift$ value depends on the i and k locations for each j . Therefore, for each possible visit, lowest $Shift$ value is calculated to find the best insertion.

For finding best insertion, we use the score of j , S_j , and $Shift_j$ value and calculate $Ratio_j$. We favor score and use the square of S_j . This gives better results for the overall score [40].

$$Ratio_j = (Score_i)^2 / Shift_i \quad (3.12)$$

The visit with highest ratio value is picked for the insertion. After insertion, arrival a , start s , $Wait$, $MaxShift$ and $Shift$ values of the visits after j need an update since we inserted a new visit and changed the timings for other visits. Updates on visit k will be represented as k' .

$$Shift_j = c_{ij} + Wait_j + T_j + c_{jk} - c_{ik} \quad (3.13)$$

$$Wait_{k'} = \max[0, Wait_k - Shift_j] \quad (3.14)$$

$$a_{k'} = a_k + Shift_j \quad (3.15)$$

$$Shift_{k'} = \max[0, Shift_j - Wait_k] \quad (3.16)$$

$$s_{k'} = s_k + Shift_k \quad (3.17)$$

$$MaxShift_{k'} = MaxShift_k - Shift_k \quad (3.18)$$

$Shift_k$ and other updates must be computed one by one towards ending visit until $Shift$ becomes 0. Then $MaxShift_j$ should be updated using 3.10. Same $MaxShift$ update must be applied visits before j again one by one but this time to the beginning visit direction (See Algorithm 2).

Shake Step

The shake step removes at least one visit from the given tour. The purpose of this step is to escape local optima (See Algorithm 3). Other than currently selected visits, variables R and S are required for this step. R is the number of visits that will be consecutively removed from the selected visits. S indicates the index to start removing visits in the selected visits list. If

Algorithm 2 Insertion Step of ILS

```
1: function INSERTIONSTEP(SelectedVisits,AllVisits)
2:
3:   NonIncludedVisits  $\leftarrow$  AllVisits - SelectedVisits
4:   Ratios  $\leftarrow$  list for NonIncludedVisits
5:
6:   for all  $j \in$  NonIncludedVisits do
7:      $Shift_j = \text{findMinShift}(j)$ 
8:      $Ratio_j = (S_j)^2 / Shift_j$ 
9:     Ratios.add( $j$ ,  $Ratio_j$ )
10:  end for
11:
12:   $j = \text{findVisitWithHighestRatio}(\text{Ratios})$ 
13:  SelectedVisits.insert( $j$ )
14:   $j.\text{calculateArrive}(), j.\text{calculateStart}(), j.\text{calculateWait}()$ 
15:
16:  for all visit  $k \in$  SelectedVisits after  $j$ ,  $Shift_k$  not equals 0 do
17:     $k.\text{updateAll}()$   $\triangleright$  update  $Arrive, Start, Wait, MaxShift, Shift$ 
18:  end for
19:   $j.\text{updateMaxShift}()$ 
20:  for all visit  $i \in$  SelectedVisits before  $j$  do
21:     $i.\text{updateMaxShift}()$ 
22:  end for
23: end function
```

the end visit is reached in the removal operation, removal is continued from the start. S variable helps algorithm to remove visits at different positions, therefore it increases to chance to escape from local optima.

Algorithm 3 Shake Step of ILS

```

1: function SHAKESTEP(SelectedVisits,  $R$ ,  $S$ )
2:
3:    $i \leftarrow \text{SelectedVisits.findVisitByIndex}(R)$ 
4:   SelectedVisits.deleteVisits( $i$ ,  $S$ )
5:
6:   for all visit  $k \in \text{SelectedVisits}$  after  $i$ ,  $Shift_k$  not equals 0 do
7:      $k.updateAll()$   $\triangleright$  update  $Arrive$ ,  $Start$ ,  $Wait$ ,  $MaxShift$ ,  $Shift$ 
8:   end for
9:
10:  for all visit  $k \in \text{SelectedVisits}$  before  $i$  do
11:     $k.updateMaxShift()$ 
12:  end for
13: end function

```

After removal, following updates are made: visits after removed visits should be shifted towards the beginning to avoid unnecessary waiting. Some of the visits may not be shifted because of the time window constraint. The visits after those non-shiftable visits should remain unchanged. The second update is similar to the one in insertion step; visits after removed visits should be updated for all variables. For the visits before removed visits, only $MaxShift$ should be updated.

Insertion and Shake Steps Example

Before moving on to the heuristic, we can give an example using insertion (Figures 3.1, 3.2) and shake (Figures 3.2, 3.3) steps. All travel costs between any points i and j , c_{ij} are 1. Following visits are initially in the tour (Figure 3.1):

- Visit a with $O_a = 8$, $C_a = 14$, $T_a = 1$, $a_a = s_a = 8$
- Visit c with $O_c = 10$, $C_c = 20$, $T_c = 3$, $a_c = s_c = 10$

- Visit d with $O_d = 17$, $C_d = 23$, $T_d = 2$, $a_d = 14$, $s_d = 17$

$Shift_b$ is computed to check if b (with $O_b = 11$, $C_b = 15$, $T_b = 2$) can be inserted between a and c :

$$\begin{aligned} Shift_b &= c_{ab} + Wait_b + T_b + c_{bc} - c_{ac} \\ &= 1 + 1 + 2 + 1 - 1 \leq 0 + 7 = Wait_c + MaxShift_c \end{aligned} \quad (3.19)$$

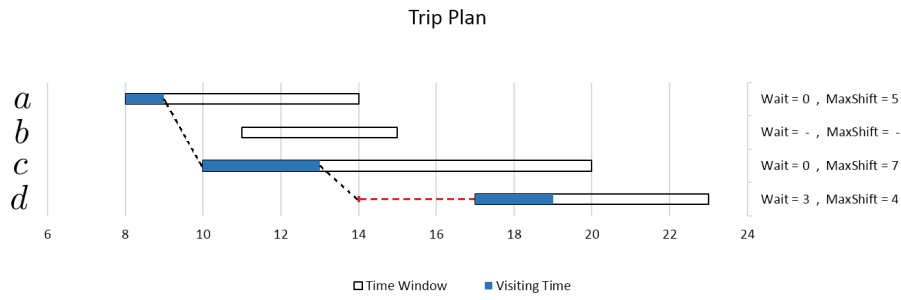


Figure 3.1: OPTW Example

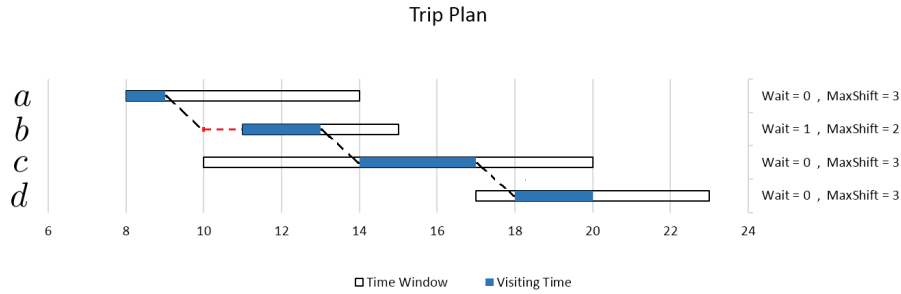


Figure 3.2: OPTW Insertion Example

Since the equation holds, b can be inserted to trip plan. As a results, visits c and d need to be updated. The arrival of c , a_c , need to be delayed by $Shift_b$ which is the total time spent for the insertion of b . This makes a_c 14 (3.20). $Wait_c$ value does not change, it is 0 again (3.21). Since new value of $Shift_c$ is 4 (3.22), s_c is delayed to 14 (3.23). $MaxShift_c$ is updated to 3, from 7. Same updates are done for visit d (3.25 - 3.29). $Shift_d$ is calculated as 1, but if it was 0, we could say that visits after d do not need to be updated (Insertion Step, Algorithm 3).

$$a_{c'} = a_c + Shift_b = 10 + 4 = 14 \quad (3.20)$$

$$Wait_{c'} = \max[0, Wait_c - Shift_b] = \max[0, 0 - 4] = 0 \quad (3.21)$$

$$Shift_{c'} = \max[0, Shift_b - Wait_c] = \max[0, 4 - 0] = 4 \quad (3.22)$$

$$s_{c'} = s_c + Shift_c = 10 + 4 = 14 \quad (3.23)$$

$$MaxShift_{c'} = MaxShift_c - Shift_c = 7 - 4 = 3 \quad (3.24)$$

$$a_{d'} = a_d + Shift_c = 14 + 4 = 18 \quad (3.25)$$

$$Wait_{d'} = \max[0, Wait_d - Shift_c] = \max[0, 3 - 4] = 0 \quad (3.26)$$

$$Shift_{d'} = \max[0, Shift_c - Wait_d] = \max[0, 4 - 3] = 1 \quad (3.27)$$

$$s_{d'} = s_d + Shift_d = 17 + 1 = 18 \quad (3.28)$$

$$MaxShift_{d'} = MaxShift_d - Shift_d = 4 - 1 = 3 \quad (3.29)$$

For visit b and other visits before b , $MaxShift$ value is updated. $MaxShift_b$ value is updated as 2 and $MaxShift_a$ value is updated as 3 (3.30, 3.30).

$$MaxShift_b = \min[C_b - s_b - T_b, Wait_c + MaxShift_c] = \min[15 - 11 - 2, 0 + 3] = 2 \quad (3.30)$$

$$MaxShift_a = \min[C_a - s_a - T_a, Wait_b + MaxShift_b] = \min[14 - 8 - 1, 1 + 2] = 3 \quad (3.31)$$

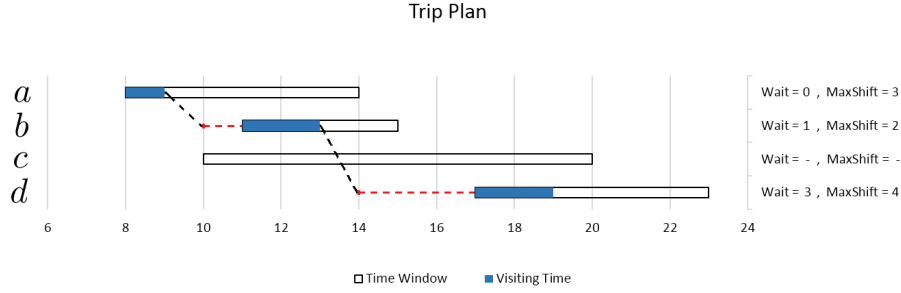


Figure 3.3: OPTW Shake Example

Shake step applied to the visit list in Figure 3.2 can be seen in Figure 3.3. R equals to 1 and S equals to 3. As a result, the visit c is removed. The visits come after c , in this case, it is only the visit d is shifted towards the beginning of its time window. The arrival, start of the service, wait, shift and maxshift values are updated for d . For the visits come before c , the visits a and b , only *MaxShift* values are updated.

Iterated Local Search Heuristic

Since we defined the insertion and shake steps, we can finally have a look at the ILS heuristic (Algorithm 4). The heuristic starts with initializing variables and it continues with finding solutions. The best solution is kept in a *BestSolution* list. The search is stopped if there is no improvement in the solution for fixed number of times. The fixed number is picked as 150. There is a tradeoff between the execution time and the quality of the solution. If the value is less, the algorithm works faster but the average total score gap between the best solutions found and the best solution that is possible increases. If fixed number is greater than 150, the algorithm works slower but solutions have higher scores.

Insertion step is used at the beginning of the heuristic loop. Visits are inserted one by one until it is not possible to add more visits. Then, if the new solution found is better than the best solution, the best solution is replaced by the new solution, R is reset to one, no improvement counter is reset to 0. If the new solution is worse than the best solution, no improvement counter is increased by one.

After checking the new solution, the shake step is applied. Then the S

value is increased by the R value, the R value is increased by one for the next iteration. Finally, if the value of S is greater than the size of the smallest tour that is found, the S value is subtracted by the size. R value is reset if its value equals to $SelectedVisits/3$. These operations are done to change the variables of the shake step and as a result, change the visits that are removed. R value reset threshold can be changed to other values but it does not really affect the quality of the result. As stated in the studies [20, 25], ILS works well for the time window problems.

3.2.2 OPTW with Transfer Patterns

ILS heuristic finds a recommended visits with average travel times. In reality, travel times are different. Hence, we calculate travel times using Transfer Patterns Algorithm and also considering walking. Meanwhile, we validate if trip is still feasible (details are in Section 3.2.3). How we process the trip plan request query can be seen in Algorithm 5.

ILS heuristic is executed first and ILS method returns best trip plan found. For each visit, we validate the visit. If the validation is not successful, the operation is aborted. If the validation is successful, travel time is calculated using Transfer Patterns Algorithm. Then visit start and end times are updated for the next iteration. A result step containing timing information is created and added to the result steps list which is the output of the OPTW with Transfer Patterns.

After all validation calls are successful, result steps list is returned as the answer. An example can be seen in Figure 4.3. For each result step, the way of traveling (walking or taking a line, with line id), the names of the start and end locations, and arriving time are included. An example trip drawn to the map manually can be seen in Figure 3.4.

3.2.3 Validating POI List

We introduce two exceptions for validation (see Section 4.8.1) and two very simple methods for the repair. After computation of real-time travel times between two points, timings are updated (Algorithm 5). With updated timings, visits are validated if they still satisfy their time window constraint. Validation is done for each visit after updating the timings.

Algorithm 4 ILS Heuristic

```

1: function ILS(AllVisits)
2:
3:    $R, S \leftarrow 1$ 
4:    $NoImprovementCount \leftarrow 0$ 
5:    $BestScore \leftarrow 0$ 
6:    $SelectedVisits, BestSolution \leftarrow \emptyset$ 
7:
8:   while  $NoImprovementCount < 150$  do
9:     while not local optimum do
10:      InsertionStep(SelectedVisits, AllVisits)
11:    end while
12:
13:    if  $getScore(SelectedVisits) > BestScore$  then
14:       $BestScore \leftarrow getScore(SelectedVisits)$ 
15:       $BestSolution \leftarrow SelectedVisits$ 
16:       $R \leftarrow 1$ 
17:       $NoImprovementCount \leftarrow 0$ 
18:    else
19:       $NoImprovementCount \leftarrow NoImprovementCount + 1$ 
20:    end if
21:
22:    ShakeStep(SelectedVisits,  $R, S$ )
23:     $S \leftarrow S + R$ 
24:     $R \leftarrow R + 1$ 
25:
26:    if  $S \geq SizeOfSmallestTour$  then
27:       $S \leftarrow S - SizeOfSmallestTour$ 
28:    end if
29:    if  $R == SelectedVisits.size()/3$  then
30:       $R \leftarrow 1$ 
31:    end if
32:  end while
33:  return  $BestSolution$ 
34: end function

```

Algorithm 5 Processing Query

```
1: function PROCESSQUERY(RequestPOIs, RequestTime)
2:
3:   BestSolution  $\leftarrow$  ILS(RequestPOIs)
4:   BSV  $\leftarrow$  BestSolution.getVisits()  $\triangleright$  BestSolutionVisits
5:   ResultSteps  $\leftarrow$   $\emptyset$ 
6:   VisitStartTime  $\leftarrow$  RequestTime
7:   VisitEndTime  $\leftarrow$  RequestTime
8:
9:   for  $i := 1$  to BSV.size() - 1 do
10:    BSV.get( $i$ ).validate(VisitStartTime)
11:
12:    WalkingTime = calculateWalkingTime(SV.get( $i$ ),
13:                                         BSV.get( $i + 1$ ), VisitEndTime)
14:    BusTime = transferPatternsQuery(SV.get( $i$ ),
15:                                     BSV.get( $i + 1$ ), VisitEndTime)
16:    if WalkingTime  $\leq$  BusTime then
17:      VisitStartTime  $\leftarrow$  VisitStartTime + WalkingTime
18:      VisitEndTime  $\leftarrow$  VisitStartTime +
19:        BSV.get( $i+1$ ).getRequiredTime()
20:      ResultSteps.add( getWalkingStep(WalkingTime) )
21:    else
22:      VisitStartTime  $\leftarrow$  BusTime
23:      VisitEndTime  $\leftarrow$  VisitStartTime +
24:        BSV.get( $i+1$ ).getRequiredTime()
25:      ResultSteps.add( getBusStep(BusTime) )
26:    end if
27:  end for
28:
29:  return ResultSteps
30: end function
```

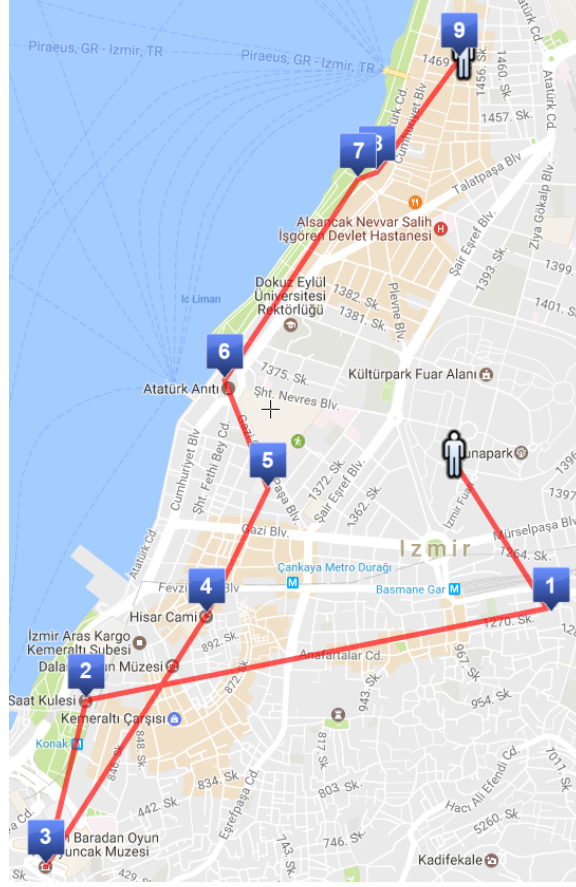


Figure 3.4: Example Trip - Start and End locations are represented as user icons

For queries that giving infeasible trip plans and therefore failing at validation, ILS heuristic is executed again but this time with one of the repairing methods as explained in the next section.

3.2.4 Handling Infeasible Trip Plans

We implemented 2 pretty simple and straightforward approaches for the repair. Score comparison of the methods for no delay and random delay scenarios can be found in Section 5.3.1. Both methods are used for only infeasible trip plans and repairing them since eventually, they should give trip plans with less total scores.

Disabling POIs Method (DP)

In this method, POI of the visit that fails validation is detected. The score of this POI is changed to 0, which results in the POI to have a *Ratio* value of 0 all the time. As a result, POI is excluded and not considered for the trip plan.

Increased Average Travel Times Method (IATT)

After excluding a POI, there is no guarantee that ILS heuristic finds a feasible solution. The result can still be an infeasible trip plan. Moreover, there might be events happening in the city that affects all the bus lines and this might end up all lines having delays, such as bad weather, special events, celebrations, etc. This can result excluding most of the POIs if Disabling POIs Method is used. In addition, a trip plan might not be found at all in the case of delays.

As a second repairing method, we increase average travel times and run ILS heuristic again. We would sacrifice some of the scores, but we can be more optimistic about the validation. Time window constraints can be met and validation will be successful. To achieve that, we multiply every average travel cost, c_{ij} , with a constant value. We picked the constant value as 1.175 because it gives the highest amount of feasible trip plans and the highest scores for the plans at the same time. For example $c_{ij} = 60$ minutes and multiplying it will result as following: $c_{ij} \times 1.175 = 70.5$ minutes. Fewer visits will be found for the trip plan but it will be easier to meet time window constraint. If the validation fails with this method, average travel times can be multiplied by a greater constant and ILS heuristic can be executed again. Eventually, a solution is found.

Implementation

In this chapter, we present the key features of the system and how they are implemented. First, we explain the datasets that we used. Then, we have a look at the system as a whole and continue by describing each component one by one.

4.1 Datasets

4.1.1 Izmir Bus Network Data

As stated before, we requested the bus network data of Izmir from ESHOT, the municipal transport company of the city of Izmir and thankfully they provided the data for our research. The data was not in GTFS format, it was in excel sheets. We used 3 of them as in Tables 4.1, 4.2 and 4.3.

Table 4.1 represents an example for the line 101. *Sequence Number* column denotes the order of the station in a trip. There are two directions for two opposite ways and they are denoted as *G* and *D* initial letters for the Turkish words. All lines have two directions. The stations of the same line for two directions can be different or same. *Line ID* shows the identification number of the line. *Station ID* is the station identification number for that step of the trip. In the table, the line 101 and its stations are shown. For both directions of the line, there are only two stations. A trip starting from the Station 50321 ends at the Station 50322. On the return trip, it starts at the Station 50322 and ends at the Station 50321.

Table 4.1: Lines Sheet - Example Line 101

| Sequence Number | Direction | Line ID | Station ID |
|-----------------|-----------|---------|------------|
| 1 | G | 101 | 50321 |
| 2 | G | 101 | 50322 |
| 1 | D | 101 | 50322 |
| 2 | D | 101 | 50321 |

Table 4.2: Stations Sheet - Stations of Example Line 101

| Station ID | Station Name | Latitude | Longitude |
|------------|-----------------|-----------|-----------|
| 50321 | Karsiyaka Carsi | 38.455232 | 27.119530 |
| 50322 | Bostanli | 38.457001 | 27.098327 |

Table 4.3: Schedules Sheet - Schedule of Example Line 101

| Line ID: 101 | | | | | |
|--------------|-------|----------|-------|--------|-------|
| Weekdays | | Saturday | | Sunday | |
| 50321 | 50322 | 50321 | 50322 | 50321 | 50322 |
| 08:00 | 07:50 | 09:00 | 09:00 | 09:00 | 09:00 |
| 08:20 | 08:10 | 12:00 | 12:00 | 12:00 | 12:00 |

Table 4.2 shows the station details. Two stations that appear in line 101 are shown in the table. Stations have name, latitude and longitude values.

In Table 4.3, a schedule for the line 101 presented. Schedules sheet has schedules of all lines for weekdays, Saturdays, and Sundays but we only used schedule for weekdays. In the columns, schedules for two directions are presented separately.

All of the tables are converted to JSON files and used by the Simulator and the back-end server (components will be explained in details). For all subsequent station pairs of the lines, travel times and event probabilities are added for the events traffic jams and accidents. Construction work event is added as a true or false variable.

4.1.2 POI Data

We created our POI data input file manually by selecting 75 tourist attractions in Izmir. POI elements have the following fields:

- **Identification number**
- **Name**
- **Description:** Brief description about the location and its history
- **Categories,** example: art, museum, historical place, etc.
- **Tags:** Specific 1 or 2 word information
- **Latitude**
- **Longitude**
- **Rating:** Score of the location. Initially, a POI has a score that is written in the POI file. At query time, the rating of the POI is replaced with the score that is calculated with respect to user's interests (Section 2.2.2).
- **Opening-Closing hours:** Times that location operates
- **Required Time:** Time needed to visit that location

POI file is a JSON file used by the Android Client for personalization purposes. The back-end server uses POI file for creating recommendations.

4.2 Architectural overview

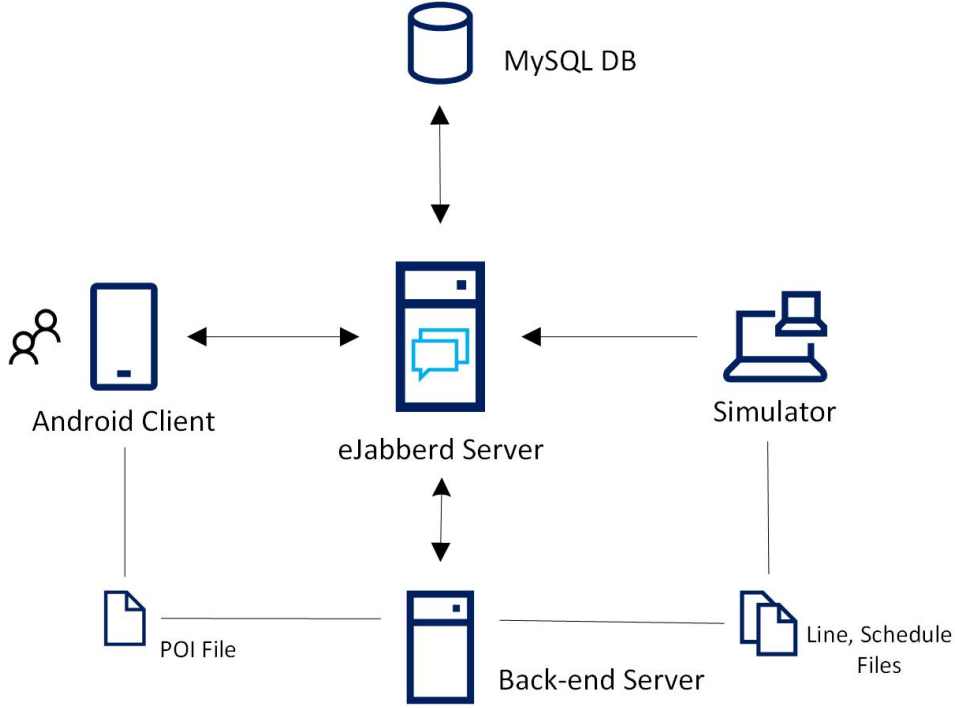


Figure 4.1: System Architecture

We implemented our system following the principles of the client-server model. The system has 2 types of clients, 2 servers, and 1 database components. Figure 4.1 visualizes our system. The input files used by different parts of the system are directly connected to the corresponding components.

4.3 Description of components

We used eJabberd Chat Server (Section 4.7) for the communications of the system and extended it with 2 custom modules. All other components are connected only to the eJabberd Server. Our second server component is back-end server. Functionalities related to the solution of the problem as stated in Chapter 3 reside in the back-end server. We explain two main subcomponents of this server; POI Recommender and Route Planner in Section 4.8.

To create real-time GPS data, we implemented a Simulator (Section 4.6).

It generates bus locations with delays and sends these locations to eJabberd Server. The eJabberd Server stores GPS data in MySQL Database (Section 4.5). The Simulator is a client in our system and it is implemented in a way that it can be exchanged with real bus clients. The second client is an Android Client (Section 4.4) which is used as a front-end application.

4.4 Android Client

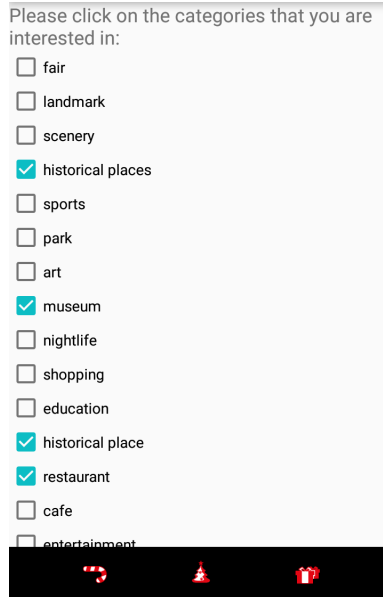
An Android Application is implemented as the user interface. Users can pick from different labels (Figure 4.2). If no label is picked, POI scores remain same. If at least one of the labels is picked, using cosine similarity, POI scores are normalized. Then by selecting trip duration and start and end locations, trip plan request is sent to eJabberd Server as a chat message. eJabberd Server sends HTTP request to the back-end server and the request is handled there. The result is sent back to the Android Client as a chat message.

The Android Client parses the answer and shows the result message in the screen showed as in Figure 4.3. The client allows personalization with labels, trip duration, start and end locations. There might be many more features implemented for more personalization.

4.5 Instantiation of the system

At the beginning, we first installed eJabberd Chat Server from the source code available. The version of the eJabberd is 15.10. After installation, the internal database of the chat server is changed to MySQL database from the default Mnesia database. Required settings have been saved to the eJabberd configuration file. We ran the server in one of the virtual machines in our department as a public server.

After setting up eJabberd Server, a populated database was needed. To achieve that, we used Line and Schedule files and created Location and Bus tables (see Figure 4.4). Since there are multiple buses running at the same line, Bus Table is implemented. Bus ID and Line ID are unique identification numbers for the bus and line units respectively. Together they form the primary key of the table.



Please click on the categories that you are interested in:

- ☐ fair
- ☐ landmark
- ☐ scenery
- ☒ historical places
- ☐ sports
- ☐ park
- ☐ art
- ☒ museum
- ☐ nightlife
- ☐ shopping
- ☐ education
- ☒ historical place
- ☒ restaurant
- ☐ cafe
- ☐ entertainment

At the bottom, there are three red icons: a candy cane, a Christmas tree, and a gift.

Figure 4.2: Android Client - Label Selection

Location Table keeps the position for the each bus and the starting time of the trip as 'Timestamp'. When buses are active and running, latitude, longitude and timestamp values are updated. When a bus finishes its trip, the fields of the corresponding record are reset.

4.6 Simulator

Since we did not have real-time GPS data coming from buses running actively, we needed to generate the data by ourselves. The *Simulator* uses lines and schedules JSON files and with respect to timings and event probabilities, imitates bus trips. Generated GPS positions are sent to eJabberd Server and kept in MySQL Database for the back-end server.

For each scheduled bus trip, Simulator generates travel times separately. It only sends latitude, longitude and starting time of the vehicle. The amount of delay is not sent to eJabberd Server because the back-end server computes the amount for each bus by itself. After a bus trip finishes, Simulator makes the bus inactive.

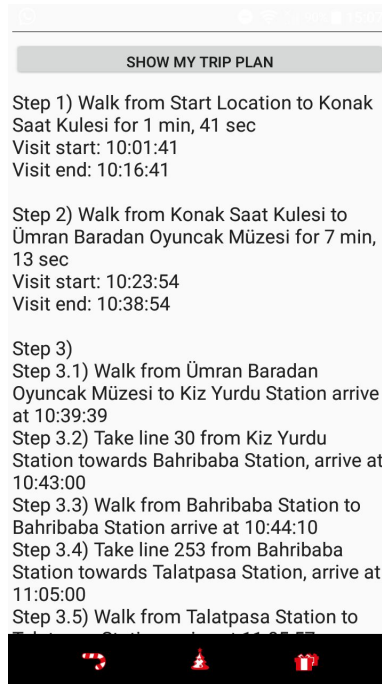


Figure 4.3: Android Client - Request Answer

4.7 eJabberd Server

Erlang Jabber Daemon (eJabberd) is a multi-platform XMPP application server. eJabberd has chosen due to extensibility, scalability and database management. The server is mostly written in functional language Erlang and it allows clients to communicate with each other using XMPP. Modular architecture offers an advantage for the development and gave us an opportunity to design a GPS system. We extended eJabberd with following custom modules.

4.7.1 Filtering Module

Filtering Module lets eJabberd filter chat messages. Chat messages are listened and filtered for two occasions:

- **Simulator Messages:** Simulator generates live GPS data and sends data as chat messages. Filtering Module detects these bus positions and stores them in MySQL Database.

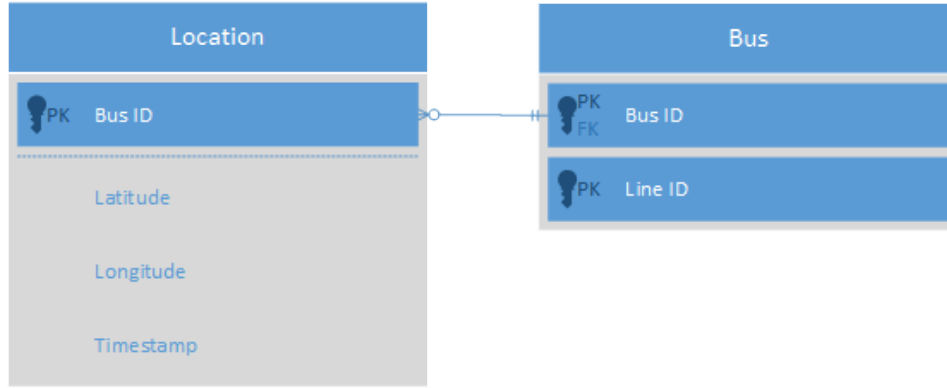


Figure 4.4: MySQL Database Tables

- **Trip Plan Request:** Android Client sends trip plan requests as chat messages. Filtering Module handles these messages and sends HTTP request to the back-end server. After receiving back the answer, the result is sent back to Android Client again as a chat message.

4.7.2 Broadcasting Module

Broadcasting Module gets a broadcast request like Filtering Module, listening chat messages. The difference is Broadcasting Module works only with chat groups. After receiving a broadcast request, the module broadcasts the positions and starting times of the active buses at that point of time. The messages are used for calculating the bus delays at the back-end server. Messages can also be used at clients for visualization. That was the main idea for using chat groups for this module.

4.8 Back-end Server

Back-end server's main functionality is to receive trip plan requests along with POI scores, trip duration, start and end locations. The server returns POIs that are recommended to visit, including travel plans in between. For this purpose, *POI Recommender* is implemented.

Another functionality of the back-end server is to accomplish precomputation for both Transfer Patterns and OPTW algorithms. Both of the precomputation tasks are carried out inside *Route Planner*. In addition,

route plan queries are evaluated by the Route Planner.

Both POI Recommender and Route Planner use same *Managers* inside the server. *Data Manager* is responsible for parsing input data and storing the data as objects. At the beginning of all operations carried out by the back-end server, Data Manager parses line, schedule, station and POI files. Data Manager object is used by two other managers.

Model Manager is responsible for using structured data in Data Manager and building the model. The most important functionality of the Model Manager is to build Graph Model of the transportation network as Time-Expanded Model.

Query Manager controls two other managers. For Transfer Patterns precomputation, Query Manager computes the patterns using the model from Model Manager. Using the same model, average travel times between every POIs are computed for OPTW solution. When the back-end server is running for answering queries, Model Manager has fewer roles since Time-Expanded Model is not needed anymore. Instead, Query Manager answers the query using precomputed data.

We can divide the back-end server into two main parts as POI Recommender and Route Planner. Both of them are implemented inside Query Manager class.

4.8.1 POI Recommender

POI Recommender is responsible for answering trip plan requests. Using the solution algorithms in Chapter 3, POI Recommender creates a list of recommended POIs. For each subsequent POI pair, the recommender uses Route Planner to generate travel plans.

The second main functionality of POI Recommender is to validate the trip plan. The trips are tested if they are feasible considering real-time bus locations and schedules. For this purpose, we introduce two exceptions.

Our first exception is *Visit Validation Exception* (VV). If time window of the visit can not be met, VV exception is thrown. The other exception is called *Out of Time Exception* (OOT), which is thrown if time limit of the trip plan is exceeded. Out of Time Exception is checked at after all travel times are updated with real-time ones and all visits are checked for Visit Validation Exception (for results see Section 5.3). As a result, Visit

Validation Exception will be thrown if the total time is exceeded and a visit failed for time window validation at the same time. For infeasible trip plans, recommender generates a new plan using repair methods as explained in Section 3.2.4.

All of the POI Recommender functionalities are carried out by Query Manager.

4.8.2 Route planner

Route Planner (RP) is used as a tool for the POI Recommender and it has three functionalities. First, RP computes Transfer Patterns for the precomputation process as mentioned in Section 2.1.4. Time-Expanded Model is used from Model Manager and Dijkstra is applied to the graph. Resulting Transfer Patterns are saved to files for query time usage.

Second, again for the precomputation process, RP calculates average travel times between POIs. This task uses same Time-Expanded Model in Model Manager and same Dijkstra execution. The difference is only travel times between POIs are computed, therefore it takes much less time for the execution and much less space when storing.

Finally, third main functionality is to answer route plan queries for the POI Recommender. For this purpose, after creating a query graph (see Section 2.1.4), the result is compared to walking time (see Algorithm 5 for the whole process). The best answer is returned to POI Recommender.

All of the Route Planner functionalities are carried out by Query Manager.

Evaluation

In this chapter, we show the results of our work. We first present execution times for query processing steps. Then we investigate the validation step for the recommendations.

5.1 Setup

For the experiments, we used the bus transportation network data of Izmir provided by ESHOT. Details can be seen in Section 4.1.1. The network consists of 333 working bus lines operating with respect to a schedule, in two opposite ways, making 25849 bus runs in a day. Furthermore, there are 7788 stations in the network. The information provided is not publicly available and it does not comply with any standard. For POI data, details can be seen in Section 4.1.2.

The machine that we used had 40 GB of memory and 64 bit Intel® Xeon® E5-2640, 2.5 GHz Processor. We ran all of our executions on a single core.

In the experiments, we used point of interest coordinates for queries. For example, for the start of the trip, we get latitude and longitude of POI A and for the end of the trip, we get latitude and longitude of POI B , ($A = B$ is possible). We ran our queries this way for simplicity.

We should note that we had 75 different POIs and we tried all combinations making 5625 different start and end location pairs for the queries. We picked 3 different trip time periods; trips of 4 hours, 6 hours and 8

hours with two different starting times 10:00 am and 12:00 noon, resulting 6 different trip periods:

- 10:00-14:00
- 10:00-16:00
- 10:00-18:00
- 12:00-16:00
- 12:00-18:00
- 12:00-20:00

We had 6 different trip time periods and each time period had 5625 different start and end location pairs. Thus, each time period has 5625 unique queries. We ran these queries with no delay and random delays scenarios. The time periods used for each experiment are explained in the descriptions of the experiments respectively.

5.2 Execution Times

In this section, we only measure the performance of our components for the execution times. Please notice that the optimality of transfer patterns has been demonstrated in the works [1, 4]. For the score performance of Iterated Local Search Heuristic (ILS), we refer the reader to read the work [40].

In this section, we will examine execution runs with only 8 hour long trips, since we can label a trip period with the longest time as the worst case scenario for the algorithm.

5.2.1 Performance of Route Planner

To examine the performance of the route planner, we measure the average time needed to answer a route planner query (public transportation). Route planner was used for every subsequent POI pairs in the POI recommendation sequence. For example, a recommendation with 3 POIs would be like in the following:

- Start Location

- POI *A*
- POI *B*
- POI *C*
- End Location

which results in 5 locations in the list with 4 travels for the subsequent pairs: Start Location \rightarrow POI *A*, POI *A* \rightarrow POI *B*, etc. In addition, we should note that our route planner considers 2 ways of traveling between POIs: walking and using public transportation. We used only 8 hour long trips, 10:00-18:00 and 12:00-20:00 in no delay and random delays scenarios. This resulted in 11250 unique queries per scenario.

For each trip plan query, travel computations took at most 3 milliseconds. For both of the scenarios, resulting trip plans had up to 3 public transportation and up to 12 walking travels (15 travels at most for a single trip). After all, a route planner query needed less than *1 millisecond* as denoted in Transfer Patterns Algorithm [1].

5.2.2 Performance of Poi Recommender

In this section, we measure the average time needed to recommend a POI list. Again, we used the same settings, 8 hour long trips, 10:00-18:00 and 12:00-20:00 in no delay and random delays scenarios.

For both of the scenarios, 11250 queries were completed between 108 and 146 seconds averaging 130 seconds. POI Recommender used $94.\bar{4}$ % of the overall execution time, leaving 122.77 seconds for creating POI recommendation list (7.33 seconds spent for the route planner). This makes 10.9135 milliseconds spent on average for creating a POI list. Moreover, there were 11.46 POIs per query result. Thus, it takes approximately 0.95 milliseconds for our recommender to recommend a POI.

5.2.3 Overall Performance of the System

The previous experiments only considered the time required by the back-end server to provide the user with a trip plan. However, as we showed in Chapter 4, the request/response process from and to the client is affected by

the network conditions. Therefore, we decided to measure the end-to-end execution time of the overall process.

Since we implemented an Android Client (AC), we measure $AC \rightarrow Server \rightarrow AC$ execution time. This time is computed as getting the time difference between sending request from the AC to the Server and receiving an answer at the AC.

We observed that the end-to-end execution time changed between 309 to 1436 milliseconds depending on the quality of the network. As a result, we can say that our system works in real-time in an acceptable time frame. Therefore, we achieved our goal of implementing a PET that works in real-time.

5.3 Recommendations Evaluation

The evaluation here focuses on the correctness of the trip plan rather than on the quality of recommendations for two reasons. First, it was not possible to carry out an off-line evaluation because we did not have any ground truth data. Secondly, an on-line evaluation with real users would have required a large amount of time and this was out of the scope of the thesis.

We explained how to recommend a trip plan in the Section 3.2.2. Since our problem is solved with average travel times, our plan might be infeasible in reality. Therefore, we need to validate time windows of the visits like we showed in Section 3.2.3. For infeasible plans, we proposed two simple approaches to fix these infeasible plans in Section 3.2.4; disabling infeasible POIs and increasing average travel times.

In this section, we investigate how many of the trip plans are feasible. For this purpose, we use all 6 time periods resulting in 33750 unique queries (5625 for each time period). We used both scenarios for these queries: no delay and random delays scenarios. Furthermore, we observe how many of the infeasible trip plans are fixed with the methods, also comparing score differences.

Before we show the results, we should note that we used *Visit Validation Exception* (VV) and *Out of Time Exception* (OOT) for feasibility as in 4.8.1.

For no delay scenario, our Simulator was not running and all the vehicles arrived at stations at projected times. Out of 33750 queries, only 19 were

Table 5.1: No Delay Scenario

| | VV Ex | OOT Ex | Feasible | Avg Diff | Max Diff | Avg Score |
|-------------|-------|--------|----------|----------|----------|-----------|
| 10:00-14:00 | 0 | 0 | 5625 | - | - | 61.7946 |
| 10:00-16:00 | 0 | 0 | 5625 | - | - | 86.1884 |
| 10:00-18:00 | 0 | 0 | 5625 | - | - | 108.5822 |
| 12:00-16:00 | 4 | 0 | 5621 | 00:02:18 | 00:02:18 | 61.6610 |
| 12:00-18:00 | 14 | 0 | 5611 | 00:02:18 | 00:02:18 | 85.6246 |
| 12:00-20:00 | 1 | 0 | 5624 | 00:02:18 | 00:02:18 | 105.4511 |

VV Ex: Visit Validation Exception, OOT Ex: Out of Time Exception,

Feasible: number of queries that trip plan is feasible,

Avg Diff: Average time difference needed to make trip plan feasible,

Max Diff: Maximum time difference needed to make trip plan feasible,

For both of the time units, time format is hh:mm:ss,

Avg Score: Average total score of the trip plans (only feasible trip plans). We should note that each POI has a score between 0 - 10 and therefore a trip plan has a score of between 0 - $10n$, where n is number of POIs in the trip plan

Same abbreviations for Table 5.2

not feasible (Table 5.1). All of them happened with the same POI which is a traditional Turkish bakery closes at 16:00. Projected time for visiting the bakery was 30 minutes, our ILS implementation produced 19 unachievable plans arriving this location by 2 minutes and 18 seconds late. These infeasible trip plans were generated because ILS heuristic uses average travel times instead of a timetable.

In addition, we had no trip plans became infeasible because of the time limit condition. In other words, we did not get any OOT exception.

For random delay scenario, our Simulator was running and producing delays for each buses depending on the event probability of the lines that they were running on. In other words, delays were not related to each other. We should also mention that the Simulator was running between 10:30 until 12:30. Thus, the travel plans only happening around 12:30 are affected and had unexpected delays. The delays produced by Simulator do not affect the travel plans of the whole day. Moreover, the average delay for each public transportation unit were 23 minutes.

Table 5.2: Random Delay Scenario

| | VV Ex | OOT Ex | Feasible | Avg Diff | Max Diff | Avg Score |
|-------------|-------|--------|----------|----------|----------|-----------|
| 10:00-14:00 | 0 | 0 | 5625 | - | - | 61.7946 |
| 10:00-16:00 | 3 | 0 | 5622 | 00:02:42 | 00:03:13 | 86.1743 |
| 10:00-18:00 | 23 | 0 | 5602 | 00:05:43 | 00:18:03 | 108.5897 |
| 12:00-16:00 | 0 | 0 | 5625 | - | - | 61.6741 |
| 12:00-18:00 | 5 | 0 | 5620 | 00:02:06 | 00:02:06 | 85.6542 |
| 12:00-20:00 | 5 | 0 | 5620 | 00:02:06 | 00:02:06 | 105.4512 |

If we look at the results (Table 5.2), we can see that number of infeasible trips increased from 19 to 36 as we introduced delays. We did not have any problems with the trips starting from 10:00 in no delay scenario, but this time most of the infeasible trip plans were in these time periods. Another point is that trip plans became infeasible with greater time amounts. For example, one of the infeasible trips were off the projected time by 18 minutes and 3 seconds. Again, there were no OOT Exceptions.

One of our conclusions from this experiment is there was a correlation between the activation of delays in the transportation network and the increase in infeasible trip plans.

Another conclusion is that there were 389612 travels for 33731 feasible trip plans (no delay scenario). Out of 389612, 346160 were walking and remaining 43452 were traveling by public transportation. If we take the average, there were 11.55 travels per trip plan, making 9.55 POI recommended on average. In 10.26 of the travels were walking, whereas only 1.29 were using public transportation. For the 43452 public transportation travels, 92761 bus trips are used, making 1.13 transfers per public transportation travel. For random delay scenario, the number of walking travels and public transportation travels were almost same but the number of transfers decreased to 1.10 from 1.13.

5.3.1 Handling Infeasible Trip Plans Results

In Section 3.2.4, we explained two simple methods to cope with infeasible trip plans. For the infeasible trip plans in the previous section, we used

Table 5.3: Repairing Infeasible Trip Plans, No Delay Scenario

| | UnF | DP Pass | IATT Pass | UnF Avg | DP Avg | IATT Avg |
|-------------|-----|---------|-----------|---------|---------|----------|
| 10:00-14:00 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10:00-16:00 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10:00-18:00 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12:00-16:00 | 4 | 4 | 4 | 79.85 | 77.475 | 71.300 |
| 12:00-18:00 | 14 | 14 | 14 | 96.00 | 94.800 | 95.043 |
| 12:00-20:00 | 1 | 1 | 1 | 107.30 | 116.300 | 115.000 |

UnF: Number of infeasible trip plans,

DP Pass: Number of feasible trip plans repaired with Disabling POIs (DP) Method,

IATT Pass: Number of feasible trip plans repaired with Increased Average Travel Times (IATT) Method,

UnF Avg: Average Score of infeasible trip plans,

DP Avg: Average Score of trip plans repaired with DP,

IATT Avg: Average Score of trip plans repaired with IATT,

Same abbreviations for Table 5.4

repairing methods and again measured how many times the recommender repaired the trip plans. In no delay scenario (Table 5.3), all of the repaired trip plans were feasible trip plans for the both methods. There was a score difference of 0.978% between the total score of the infeasible trip plans and the plans repaired by Disabling POIs Method (DP). The score difference between the total score of the infeasible trip plans and the plans repaired by Increased Average Travel Times Method (IATT) were 2.26%. DP Method seemed to be the winner in this scenario.

In random delays scenario (Table 5.4), 3 of the repaired trip plans were still infeasible for DP method. As a result, DP had score difference of 13%. IATT Method had score difference of 11.07%. IATT seemed to be the winner in this scenario by the score difference and number of feasible trip plans. For both of the methods, score difference increased with introduced delays.

Table 5.4: Repairing Infeasible Trip Plans, Random Delay Scenario

| | UnF | DP Pass | IATT Pass | UnF Avg | DP Avg | IATT Avg |
|-------------|-----|---------|-----------|---------|---------|----------|
| 10:00-14:00 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10:00-16:00 | 3 | 3 | 3 | 112.766 | 92.300 | 101.266 |
| 10:00-18:00 | 23 | 20 | 23 | 106.695 | 87.487 | 92.647 |
| 12:00-16:00 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12:00-18:00 | 5 | 5 | 5 | 81.560 | 82.840 | 79.1000 |
| 12:00-20:00 | 5 | 5 | 5 | 105.600 | 108.120 | 97.080 |

Conclusion and Future Work

The purpose of this work was to solve Tourist Trip Plan Desing Problem (TTDP) while providing travel plans between visits using real-time public transportation data. In order to solve TTDP, we used a heuristic called ILS. To provide travel plans, we implemented a route planner using an optimization technique called Transfer Patterns Algorithm.

At the beginning, we assumed that transportation units send real-time GPS data. Therefore, the initial requirement for the route planner was to consider the latest locations of the vehicles. A requirement for the recommender was to be able to evaluate trip plan queries in real-time.

At the end, our requirements were fulfilled and we managed to build a system that recommends trip plans considering real-time transportation data. Our tool was able to satisfy the initial requirements. Therefore, we can say that it is possible to build a system as we stated, using ILS heuristic with Transfer Patterns Algorithm. For the experiments, we used real bus transportation data of the city of Izmir. In addition, we created an input scenario with 75 points of interest for the recommender.

We modeled TTDP as Orienteering Problem with Time Windows (OPTW). Our recommender uses personalized data from the user and generates the point of interest scores for OPTW solution. For transportation between these points, we used location-to-location queries evaluated by our route planning algorithm with transfer patterns.

The tourist trip plan recommendations can be computed in real-time and each one takes 10 milliseconds in average. Both route planning and

tourist trip planning problems are very hard to solve in real-time, separately. We focused on solving these problems together which made the problem even more difficult. Therefore we solved the problem using optimization techniques. Since the techniques used might yield few non-optimal solutions in the presence of delays, we wanted to check how many of the results were correct in our experiments. We examined timings for the visitations at the points of interest and checked if the plans were feasible. For the few solutions that do not satisfy problem constraints, we provided simple solutions and investigated the results again.

For producing real-time bus data, we implemented a Simulator. Our Simulator considers event delays and probabilities for accident, traffic jam, and construction work cases. Moreover, the Simulator produces realistic GPS data. For communications, we used eJabberd Server and built an Android application as a client.

For the future work, there are some improvements can be made in different parts of the system. An addition to route planning part might be implementing the full version of Transfer Patterns Algorithm. Right now we use a simple version of it, without computing hub stations and storing transfer patterns as directed acyclic graphs. With these improvements, transfer patterns precomputation can be done faster and resulting patterns can be stored using less space on the hard drive.

At the beginning of the project, we wanted to use real-time bus GPS data for our real-time route planning algorithm. At that time, there was no infrastructure for real-time data because of the system changes happening at ESHOT, so we could not request it. Instead, we used our own simulator imitating bus behavior by producing GPS data also producing delays with different event probabilities. If real-time data of this kind is provided, our system can be integrated into it and the recommender would be closer to real-life scenario of developing countries.

For the recommender part, more options can be provided to users so that the system produces more personalized recommendations. Another contribution for this part might be recommending multiple day trip plans (TOPTW) instead of single day trip plans (OPTW). This can be done by few additions on our OPTW solution.

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum

Unterschrift

Bibliography

- [1] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *Proceedings of the 18th Annual European Conference on Algorithms: Part I*, ESA'10, pages 290–301, Berlin, Heidelberg, 2010. Springer-Verlag.
- [2] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. *Route Planning in Transportation Networks*, pages 19–80. Springer International Publishing, Cham, Switzerland, 2016.
- [3] Hannah Bast, Matthias Hertel, and Sabine Storandt. Scalable transfer patterns. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 15–29. Society for Industrial and Applied Mathematics, 2016.
- [4] Hannah Bast, Jonas Sternisko, and Sabine Storandt. Delay-robustness of transfer patterns in public transportation route planning. In *ATMOS-13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems-2013*, volume 33, pages 42–54. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.
- [5] Reinhard Bauer and Daniel Delling. Sharc: Fast and robust unidirectional routing. *J. Exp. Algorithmics*, 14:4:2.4–4:2.29, January 2010.

- [6] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez. Recommender systems survey. *Know.-Based Syst.*, 46:109–132, July 2013.
- [7] Derek Bridge, Mehmet H. Göker, Lorraine McGinty, and Barry Smyth. Case-based recommender systems. *Knowl. Eng. Rev.*, 20(3):315–320, September 2005.
- [8] Gerth Stølting Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries. *Electronic Notes in Theoretical Computer Science*, 92:3 – 15, 2004.
- [9] Robin Burke. *Hybrid Web Recommender Systems*, pages 377–408. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [10] I-Ming Chao, Bruce L. Golden, and Edward A. Wasil. The team orienteering problem. *European Journal of Operational Research*, 88(3):464–474, 1996.
- [11] Daniel Delling, Andrew Goldberg, and Renato Werneck. Graph partitioning with natural cuts. Technical report, December 2010.
- [12] Daniel Delling, Thomas Pajor, and Dorothea Wagner. *Engineering Time-Expanded Graphs for Faster Timetable Information*, pages 182–206. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
- [13] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.
- [14] General Directorate of ESHOT. <http://www.eshot.gov.tr> (visited on 20-04-2017).
- [15] Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. *Multi-criteria Shortest Paths in Time-Dependent Train Networks*, book-Title="Experimental Algorithms: 7th International Workshop, WEA 2008 Provincetown, MA, USA, May 30-June 1, 2008 Proceedings, pages 347–361. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
- [16] David Eppstein and Michael T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. In *Proceedings of the 16th ACM*

- SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '08, pages 16:1–16:10, New York, NY, USA, 2008. ACM.
- [17] A. Felfernig and R. Burke. Constraint-based recommender systems: Technologies and research issues. In *Proceedings of the 10th International Conference on Electronic Commerce*, ICEC '08, pages 3:1–3:10, New York, NY, USA, 2008. ACM.
 - [18] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.
 - [19] Ander Garcia, Maria Teresa Linaza, Olatz Arbelaitz, and Pieter Vansteenwegen. *Intelligent Routing System for a Personalised Electronic Tourist Guide*, pages 185–197. Springer Vienna, Vienna, 2009.
 - [20] Damianos Gavalas, Charalampos Konstantopoulos, Konstantinos Mastakas, and Grammati Pantziou. A survey on algorithmic approaches for solving tourist trip design problems. *Journal of Heuristics*, 20(3):291–328, June 2014.
 - [21] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, August 2012.
 - [22] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, December 1992.
 - [23] Bruce L. Golden, Larry Levy, and Rakesh Vohra. The orienteering problem. *Naval Research Logistics (NRL)*, 34(3):307–318, 1987.
 - [24] Nathaniel Good, J. Ben Schafer, Joseph A. Konstan, Al Borchers, Badrul Sarwar, Jon Herlocker, and John Riedl. Combining collaborative filtering with personal agents for better recommendations. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*, AAAI '99/IAAI

- '99, pages 439–446, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.
- [25] Hideki Hashimoto, Mutsunori Yagiura, and Toshihide Ibaraki. An iterated local search algorithm for the time-dependent vehicle routing problem with time windows. *Discrete Optimization*, 5(2):434 – 456, 2008.
 - [26] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. *Fast point-to-point shortest path computations with arc-flags*, pages 41–72. American Mathematical Society, 2009.
 - [27] Marisa G. Kantor and Moshe B. Rosenwein. The orienteering problem with time windows. *The Journal of the Operational Research Society*, 43(6):629–635, 1992.
 - [28] C P Keller and M F Goodchild. The multiobjective vending problem: a generalization of the travelling salesman problem. *Environment and Planning B: Planning and Design*, 15(4):447–460, 1988.
 - [29] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.
 - [30] Pasquale Lops, Marco de Gemmis, and Giovanni Semeraro. *Content-based Recommender Systems: State of the Art and Trends*, pages 73–105. Springer US, Boston, MA, 2011.
 - [31] Rolf H. Möhring. *Verteilte Verbindungssuche im öffentlichen Personenverkehr Graphentheoretische Modelle und Algorithmen*, pages 192–220. Vieweg Teubner Verlag, Wiesbaden, 1999.
 - [32] Matthias Müller-Hannemann and Mathias Schnee. Finding all attractive train connections by multi-criteria pareto search. In *Proceedings of the 4th International Dagstuhl, ATMOS Conference on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems*, ATMOS'04, pages 246–263, Berlin, Heidelberg, 2007. Springer-Verlag.

- [33] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *J. Exp. Algorithmics*, 12:2.4:1–2.4:39, June 2008.
- [34] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D Zaroliagis. Experimental comparison of shortest path approaches for timetable information. In *ALENEX/ANALC*, pages 88–99. Citeseer, 2004.
- [35] Giovanni Righini and Matteo Salani. Decremental state space relaxation strategies and initialization heuristics for solving the orienteering problem with time windows with dynamic programming. *Comput. Oper. Res.*, 36(4):1191–1203, 2009.
- [36] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra’s algorithm on-line: An empirical case study from public railroad transport. *J. Exp. Algorithmics*, 5, 2000.
- [37] General Transit Feed Specification Reference. <https://developers.google.com/transit/gtfs/reference> (visited on 20-04-2017).
- [38] T. Tsiligrirides. Heuristic methods applied to orienteering. *Journal of the Operational Research Society*, 35(9):797–809, 1984.
- [39] Pieter Vansteenwegen. Planning in tourism and public transportation. *4OR*, 7(3):293, 2008.
- [40] Pieter Vansteenwegen, Wouter Souffriau, Greet Vanden Berghe, and Dirk Van Oudheusden. Iterated local search for the team orienteering problem with time windows. *Comput. Oper. Res.*, 36(12):3281–3290, December 2009.
- [41] Pieter Vansteenwegen and Dirk Van Oudheusden. The mobile tourist guide: An or opportunity. *OR Insight*, 20(3):21–27, 2007.
- [42] Cédric Verbeeck, Pieter Vansteenwegen, and El-Houssaine Aghezzaf. The time-dependent orienteering problem with time windows: a fast ant colony system. *Annals of Operations Research*, pages 1–25, 2017.
- [43] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.