UPPSALA
UNIVERSITET

IT16084
Examensarbete 30 hp
Oktober 2016

Tourist Scheduler Using Constraint Programming

Ilyass Garara

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Tourist Scheduler Using Constraint Programming

Ilyass Garara

Planning a trip in present time still requires a significant amount of time and effort. With the absence of a single service that manages the planning aspect of a trip, tourists have to explore multiple sources of data, both online and offline, in order to prepare their schedules. Therefore, the following master thesis project introduces a system that generates a schedule for tourists who would like to visit the city of Stockholm. The system communicates with a mobile application to receive user-defined data such as mobility, budget and points of interest in order to find the best schedule by combining the efficiency and flexibility of constraint programming with real-time data sources and a routing algorithm.

Handledare: Aneta Vulgarakis & Azadeh Bararsani
Ämnesgranskare: Edith Ngai
Examinator: Mats Daniels
IT16084
Tryckt av: Reprocentralen ITC

Acknowledgements

I would like to thank:

- My two supervisors at Ericsson Research, Azadeh Bararsani and Aneta Vulgarakis, for their help throughout the duration of the thesis
- The CityPulse team for providing help and support with their framework
- My reviewer Edith Ngai, for the advice she has shared about the thesis
- The Research department at Ericsson for their feedback about my work
- The Swedish Institute and Uppsala University for the opportunity to pursue my master's programme in Sweden

This master thesis has been partially supported by EU FP7 CityPulse Project under grant No.603095. `http://www.ictcitypulse.eu`

Glossary

- POI: Point Of Interest
- IoT: Internet of Things
- IoP: Internet of People
- TSP: Traveling Salesman Problem
- CPM: Critical Path Method
- ASP: Answer Set Programming
- GDI: Geospatial Data Infrastructure
- HTTP: HyperText Transfer Protocol
- OSM: OpenStreetMap
- API: Application Programming Interface
- REST: Representational State Transfer

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

With the ever-growing sector of tourism on a global scale, and the recent surge of Sweden as a popular tourist destination at the international level [1], there are numerous opportunities to offer services that improve the overall experience of visitors during their stay. In fact, tourism has come a long way since the emergence of online services, as different web and, later on, mobile applications took over the hegemony of printed guides, books, leaflets, and road maps.

Online services reduced the amount of efforts to plan for trips and for activities in a city compared to their predecessors, and yet their increasing use often implies that the traveler(s) have to browse several data sources (e.g. applications, websites, guides, etc.) in order to choose which places to visit, which events to attend, and which routes to take from one point to another. For instance, we often rely on search engines or websites such as TripAdvisor or Yelp in order to find interesting places and activities in a city, which, besides housing, is often the first step in planning a trip. Sometimes, incomplete or conflicting information from two sources forces us to consult the official website of a place (if a website exists), be it to get its opening times, price categories, or even contact information. When that first step is over, it might be necessary to plan for a way to move between these points of interest (POIs), whether that is due to a dense traffic or financial reasons. In that case, services such as Google Maps, OpenStreetMaps, and public transportation websites are usually the most reliable options.

Currently, it is not possible to use all of the available resources about a city in a single service, nor can any of them come up with complete plans that take all of the aforementioned necessities into account. As a result, even though the online services reduced the time spent flipping the pages of a printed book, they still did not manage to save more time from browsing different web pages and consulting several applications. In fact, an online service that plans all aspects of a trip appears as a natural evolution of the existing platforms and services, and the emergence of the Internet of Things (IoT) could be a major contributing factor to the spread of services with a similar mindset.

1.2 Objective & Scope

The main objective behind this thesis project is to explore the feasibility of a service that automatically generates complete schedules for a trip. It shall combine the existing sources of data related to events and POIs in a city, as well as all necessary routing or transportation resources. The user would thus select a list of POIs, give some information about the trip, and, based on these criteria, receive a schedule that includes all the chosen POIs. Therefore, the project introduces the following challenges:

- Identify the types of POIs and events to have in the city map
- Define the criteria that would help generate an efficient schedule
- Develop an algorithm that builds an optimal schedule based on the chosen criteria
- Use one or several components from the IoT project CityPulse
- Assess the performance and scalability of the algorithm and the system components

Chapter 2

Preliminaries

2.1 Existing Scheduling Problems

2.1.1 Hamiltonian Path/Circuit

As the red line in Figure 2.1 shows, a Hamiltonian path is a path that visits every single vertex in a graph exactly once [3]. A Hamiltonian circuit, on the other hand, is a Hamiltonian path where the initial and final vertices are connected to form a closed cycle, as illustrated by the dashed edge in Figure 2.1. Hamiltonian paths and circuits are very common in planning and scheduling problems where the places, activities or tasks can be expressed as vertices in a graph, and where each pair of vertices has an edge with a cost value.

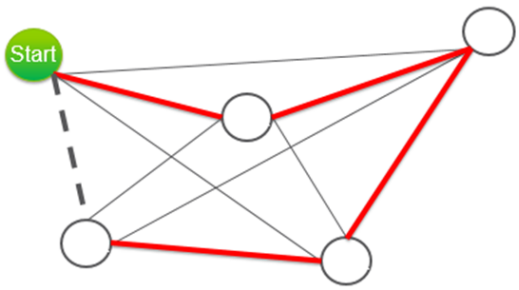

Figure 2.1: Example of the Traveling Salesman Problem

2.1.2 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a popular optimization problem where, starting from an initial city, a -sales- person has to visit all the cities included in the problem and come back to the initial city to finish the tour [4]. The main objective is to complete the tour with minimal distance, time, or financial cost.

The TSP can be expressed in a graph where vertices represent cities (or any other type of location), the weighted edges contain the distance between every pair of cities, and the result is a Hamiltonian circuit. Figure 2.1 is an example of a TSP where the circuit shows the solution. Symmetry also affects the performance of the TSP as asymmetric instances result in double the number of solutions than their symmetric counterparts. For instance, real-life routing TSPs imply that several streets are one-way streets or are restricted to specific types of vehicles (e.g. bikes, small cars etc.), which makes asymmetric TSPs more common in this area.

2.1.3 Activity Selection Problem

The Activity Selection Problem is another example of schedule optimization problems. Given a time frame and a set of activities with their respective start and end times, the objective is to execute the highest possible number of activities during the said time frame without overlapping [5]. The solution to the example in Figure 2.2 is the subset {a,d,g}, which is the biggest subset of activities that can be executed without time conflicts. This type of optimization can be very useful when the available time range cannot include all of the activities.

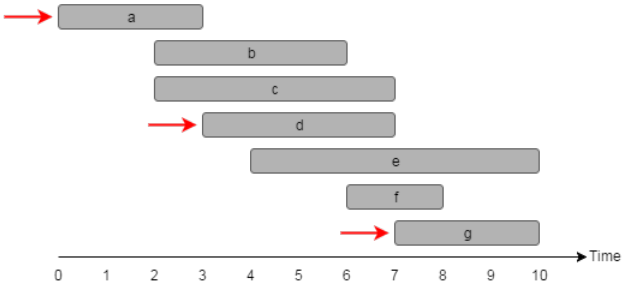

Figure 2.2: Example of the Activity Selection Problem

2.1.4 Critical Path Method

Critical Path Method (CPM) is an activity scheduling algorithm that is often used to manage project tasks. The project is thus divided into a list of tasks where each one has a duration and a list of prerequisite tasks. The goal behind the CPM is to use the precedence between different tasks to identify which ones are considered "critical" and whose delay would increase the total duration of the project as a result [6]. Figure 2.3 shows a network diagram that represents a list of project tasks that are connected based on task precedence. The path {A,C,F} in red has the longest duration and is therefore the critical path.

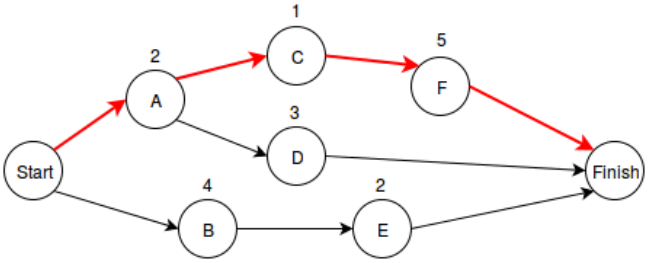

Figure 2.3: Example of a Critical Path

2.2 Existing Technologies

2.2.1 Constraint Programming

Constraint Programming is a type of Declarative Programming that solves complex problems thanks to the relationships between different variables, and a set of search heuristics. These relationships, expressed as constraints, control the values of variables and, later, allows the program to derive one or many solutions for the problem instance. The most common approach to that is the refinement model, usually through constraint propagation, which attributes an initial finite domain to each variable, and then restricts and reduces the domain using the constraints in the program [7]. Constraint Programming is usually carried using libraries for Imperative Programming languages.

Gecode

Gecode stands for **Ge**neric **Co**nstraint **D**evelopment **E**nvironment. Gecode's manual defines it as "an open, free, portable, accessible, and efficient environment for developing constraint-based systems and applications" [8]. It provides a constraint solver that performs propagation on a model, then proceeds with the search by branching over the search space to find one or many solutions. Gecode, as well as all of its programs, are implemented in C++, and it provides several implemented global constraints, some of which are excellent to solve scheduling problems. It is rapidly growing into a platform of choice for constraint programming, both in academia and industry, due in large part to its efficient memory management, time performance, and scalability [9].

2.2.2 Answer Set Programming

Answer Set Programming (ASP) is a type of Logic Programming, where logic is used as the basis to solve complex problems. ASP programs consist of a set of rules (e.g. facts, predicates, constraints) from which answer sets are derived. It is also a subtype of Declarative Programming because these rules describe the result(s) that the program should accomplish, rather than the steps it should follow in order to do so (i.e. Imperative Programming) [10].

Clingo

Clingo is one of the most popular ASP tools. It is an integrated system that combines the ASP solver Clasp, and GrinGo. GrinGo is a grounder that translates the user's rules into a logic program. Clingo pipes the resulting program that is generated by GrinGo, which Clasp uses to compute the answer sets based on conflict-driven answer set solving [11] [12]. Clingo, Clasp, and GrinGo are implemented in C++, but their programs use Lparse syntax.

ClingCon

ClingCon is an ASP solver with integrated features from Constraint Programming. Similarly to Clingo, ClingCon is a combination of tools, namely Clingo and Gecode. It basically incorporates

constraint propagation into ASP solver Clingo in order to improve performance related to conflict analysis [13]. However, the latest version of ClingCon (2.0.3) was released in 2012 and the tool is no longer updated or maintained.

2.3 CityPulse Project

CityPulse is a open-source project that relies on real-time IoT stream processing and knowledge-based computing to affect decision making in smart city applications. It is supervised by the European Commission and has a consortium comprised of 9 different universities and companies [14].

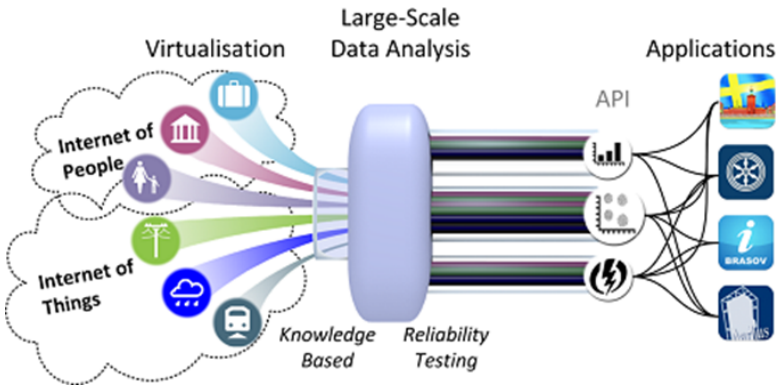

Figure 2.4: CityPulse architecture overview

As Figure 2.4 illustrates, CityPulse is a very interesting project in the sense that it combines many data sources that differ in origin (e.g. sensors, social media) and type (e.g. traffic, pollution, density), so as to provide smart city applications with real-time data and, in turn, find optimal solutions that comply with the user's requirements and preferences. This is done using modules such as User-Centric Decision Support, which uses Answer Set Programming to provide optimal solutions to complex requests such as routing, and adjusts these solutions to real-time events if necessary [15]. The Tourist Scheduler use case will implement a component with a role similar to the Decision Support component in order to find the best schedules based on the user's preferences.

Chapter 3

Requirements & Design

3.1 Use Case Description

Based on the concept that was elicited in Section 1.2, the system shall consist of an application that interacts with the user, in addition to a backend that handles data sources and the schedule requests. The application thus shows an interactive map that allows the user to search and browse for POIs, and provides information pages about every existing POI. In addition, it contains forms that collect more information about the trip and the traveler, which the user can send to request an appropriate trip schedule. Then, the backend responds with a complete path or circuit around the city after gathering all necessary data about the POI's schedules and the possible routes to take. The use case diagram in Figure 3.1 summarizes the principal actions that the user should be able to perform.

3.2 Functional Requirements

Table 3.1 and Table 3.2 list all the functional requirements for the Android application and the backend, respectively.

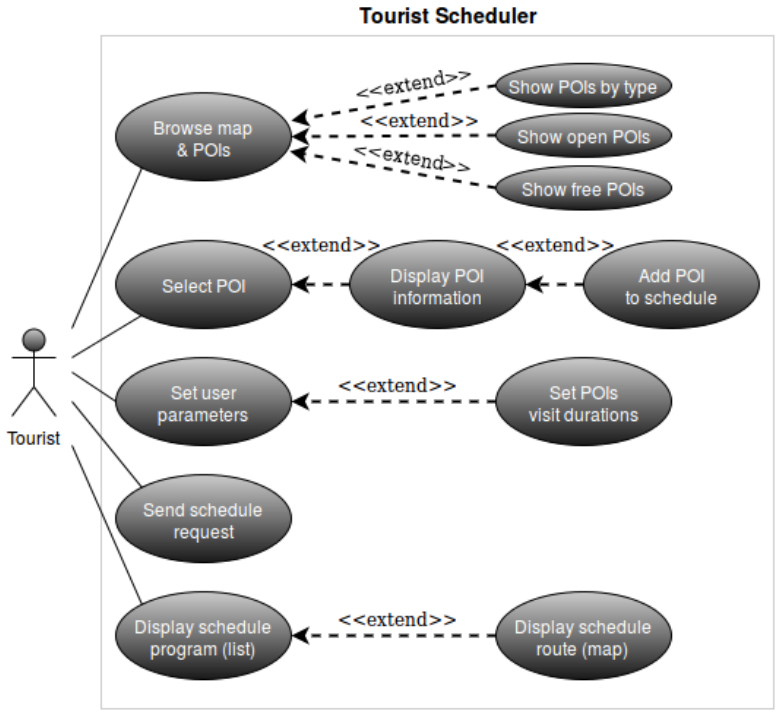

Figure 3.1: Use case diagram

No.	Function	Description
1	Browse Map	navigate in a map and see available POIs
2	Search POI	display a POI in the map based on searched keyword(s)
3	Filter POIs	show or hide a POI category (i.e. by type, fee, opening)
4	Explore POI Profile	display information about a POI (see Section 3.5)
5	Add POI to Request	add selected POI to the schedule request
6	Add Trip Settings	enter data about the trip (e.g. budget, mobility, time)
7	Edit Visit Durations	change the visit duration of every selected POI
8	Send Trip Request	submit the POI list and user settings to the server
9	Explore Schedule	display the resulting schedule per segment
10	Display Route	show the resulting schedule's routes in the map

Table 3.1: Functional requirements for the application

No.	Function	Component
1	Start/Stop Server	Request Handler (RH)
2	Forward HTTP Request	Request Handler
3	Read HTTP Request Parameters	Request Handler
4	Load/Update POI Database	Request Handler & GDI
5	Load API events	Request Handler & Event Handler
6	Build Request	Request Handler
7	Get Candidate Routes	Request Handler & GDI
8	Get Schedule	Request Handler & Constraint Solver
9	Build Route	RH & Public Transport Travel Planner
10	Send HTTP Response	Request Handler
11	Get Events	Event Handler
12	Connect to GDI Database	GDI

Table 3.2: Functional requirements for the application

3.3 Non-functional Requirements

The following non-functional requirements were taken into account during this project:

- *Usability*: A pillar of the interactive application, usability gives a much better experience for the user and simplifies the task of entering and submitting the correct information that matters to the system in the correct format.
- *Response Time*: In addition to the benefits a good time performance presents in enhancing the overall user experience, it is primordial for the system to send back results in a reasonable amount of time.
- *Scalability*: Due to the various durations of trips, users can select widely different numbers of POIs. As a result, the backend of the system must keep a relatively stable response time with regards to the number of selected POIs.
- *Accuracy*: If the system is to be of any use in the real world, the schedules that are sent to the user as solutions to his/her queries must be accurate. In fact, the schedules should not only contain the right POIs and the correct routes, it should ideally provide the best possible alternative.
- *Robustness*: Because user input could be erroneous, the application must make sure that a user request has the correct parameters and that their values are valid. Therefore, the

application -and the system as a whole- must handle and respond to erroneous parameters appropriately in order to avoid unexpected failures.

3.4 System Architecture

In addition to the Android application, the Tourist Scheduler system is composed of several modules at the level of the backend, as shown in Figure 3.2. Each component is listed and its role described in this section.

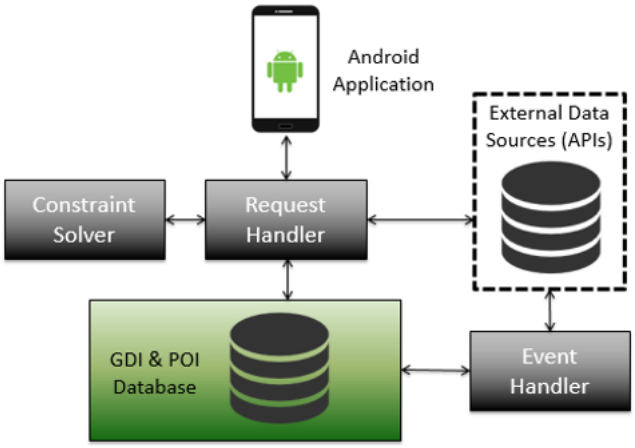

Figure 3.2: System architecture overview

3.4.1 Request Handler

The main component of the backend is the Request Handler. It receives requests from the application and manages the whole process behind generating and sending back responses. The Request Handler responds to two types of requests from the Android application; it sends the list of POIs that should appear in the map when the application starts (see Figure 3.4), and processes schedule requests (see Figure 3.3). In order to achieve these two objectives, it relies on several other components, which will be introduced in the remaining parts of this section.

3.4.2 Constraint Solver

The Constraint Solver represents the core logic of the backend. Its role is to make decisions related to scheduling and travel planning based on the parameters that were specified by the

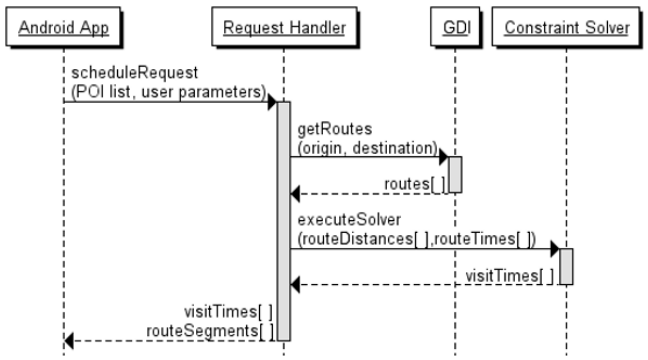

Figure 3.3: Sequence diagram of a schedule request scenario

user in a schedule request. Therefore, the Constraint Solver has to consider the respective opening and closing times of all selected POIs, the time and distance between every pair of POIs, in addition to the travel period of the tourist. Generating the best schedule means that the result should cover all of the selected POIs while minimizing the time and/or distance costs. Therefore, the following constraints need to be satisfied:

- **No time conflicts**: the program must assign distinct visit times to the POIs, and consider the visit duration of each POI and the time cost to travel between two POIs.
- **Hamiltonian path**: the resulting schedule must visit each POI once as in a Hamiltonian path, which means that it must determine the subsequent POI to be visited for each node.
- **Starting location**: since the starting location is expressed as a POI in the model, the program must force it to be the first POI to be visited.
- **Minimum cost**: it is primordial for the model to find the result with minimal cost. Therefore, each POI must compute its respective cost based on its successor in the path.
- **Travel period**: the resulting schedule must comply with the travel period that is set by the user. For instance, if the user selects a one-day trip from 9AM to 8PM, the journey from the starting location must start after 9AM, whereas the visit of the last POI must end at 8PM at the latest.

It should be noted that the time unit used in this model should be minutes. The reason is that minutes are the smallest significant unit for time variables and constants such as the opening and closing times of the PoIs, as well as the starting and ending times of a visit. Therefore, all time variables in this model are integer variables in minutes.

3.4.3 Geospatial Data Infrastructure

The Geospatial Data Infrastructure (GDI) is part of the CityPulse framework [14]. It consists of a geospatial database of the city of Stockholm, which contains data about its landmarks and different types of paths and routes. In this project, the GDI is used as a source of data where the Request Handler finds the shortest route between two POIs, which allows it to store the route as a candidate on one hand, and use the route's time and distance to build the data matrix that is sent to the Constraint Solver on the other. Additionally, the GDI is extended with an extra database that stores POI and event data (see Figure 3.6).

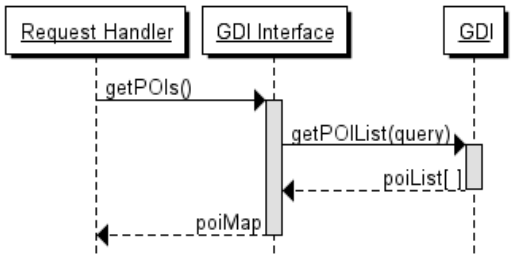

Figure 3.4: Sequence diagram of importing the POI list

3.4.4 Event Handler

The Event Handler is in charge of keeping track of new events that are scheduled to take place in the city. It communicates with several external data sources and updates the list of upcoming events whenever it finds new entries. The Event Handler runs as part of the Request Handler when it starts and, subsequently, is called periodically to keep the list up-to-date. Figure 3.5 summarizes the steps taken by the Event Handler.

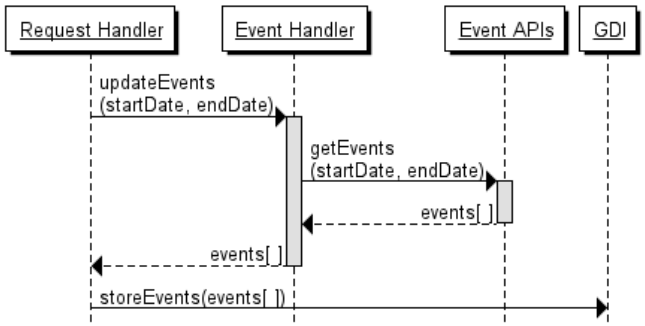

Figure 3.5: Sequence diagram of the Event Handler

3.5 Data Model

Before going further with data organization and management in the system, it is primordial to know which information about the user and the eventual trip would be of help to generate an efficient schedule. The following list shows the criteria that were adopted in this use case, and explains the impact that each one has on the scheduling process:

- *POI List*: The system needs to know the user's chosen POIs, which implies that each POI must come with information about location, opening times, and entry fee.
- *Visit Duration*: For each selected POI, the system needs to know its visit duration. Therefore, the application should provide a recommended visit duration for each POI, and users should have the freedom to specify their own.
- *Budget*: Budget can be compared to the total cost of accessing the selected POIs, and can help the application prevent the schedule requests with insufficient budget from being sent to the backend and increase its load.
- *Trip Start & End Dates*: Helps the system consider the opening times of the POIs based on the day of the week that matches the period of the trip.
- *Mobility*: Allows the system to use the appropriate components when computing the best routes depending on whether the visitor has a car or is using public transportation.
- *Starting Location*: Enables the system to give an accurate schedule that starts from the actual starting point, rather than one of the selected POIs or a default location.

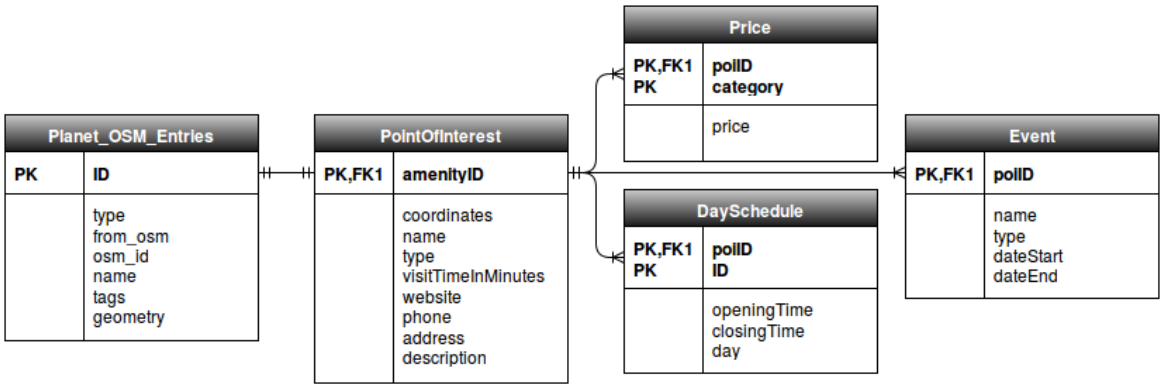

Figure 3.6: Overview of the GDI's POI Database

Android application receives most of its data from the Request Handler. It consists mainly of information that is used to display them on the map and build their profile pages. In addition, it receives data to display a schedule result. This communication is done through several objects that are illustrated in Figure 3.7.

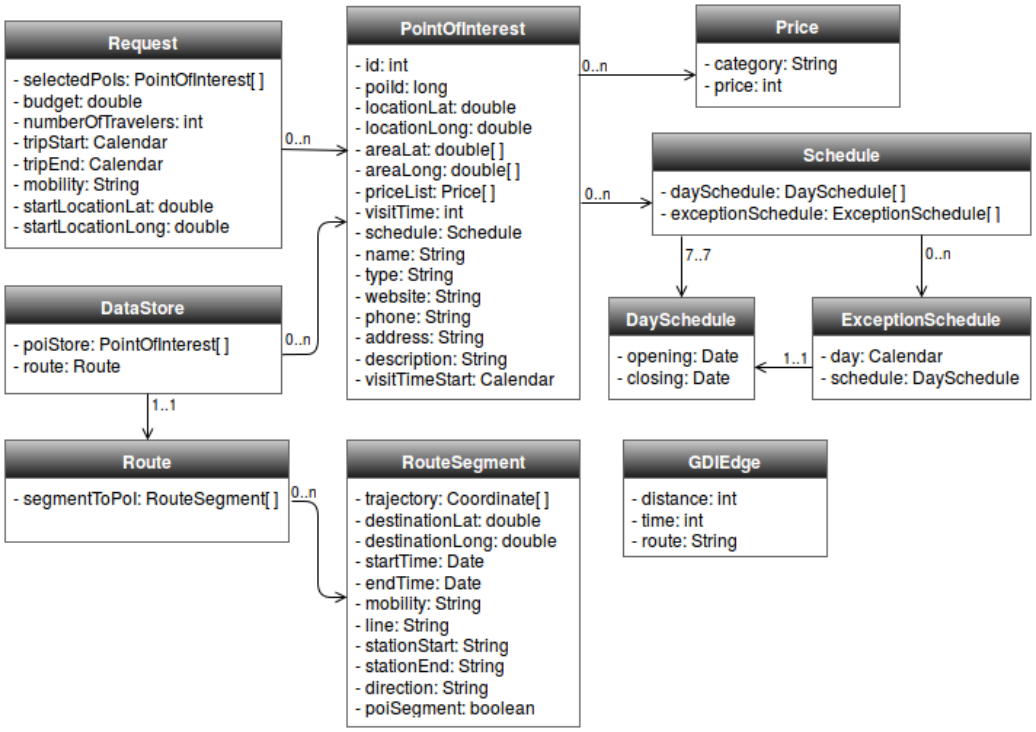

Figure 3.7: Class diagram of the package Objects

As for the Request Handler, it loads the list of available POIs from the GDI when the server starts. The POI database that complements the GDI is shown in Figure 3.6. It also communicates with an external API to find public transport options between two POIs, and whose significant data is listed in Table 3.3.

Variable	Type	Description
originCoordLat	String	The latitude of the starting position of the route
originCoordLong	String	The longitude of the starting position of the route
destCoordLat	String	The latitude of the destination of the route
destCoordLong	String	The longitude of the destination of the route
date	String	Date of the journey, format: yyyy-MM-dd
time	String	Time of the journey day, 24-hour format: HH:mm
lang	String	The language of the JSON answer, set to English
numF	Integer	Number of returned results, set to one result
originWalk	Integers	The starting position of the route
destWalk	Integers	The starting position of the route

Table 3.3: Request Data for the public transport travel planner

Chapter 4

Implementation

4.1 Android Application

Table A.1 in Appendix A summarizes the technical specifications of the application. The user interface is designed and implemented following Google's Material Design guidelines [16]. The main activity contains the map where the user can select POIs to explore their information or add them to the trip list. It uses the Google Maps Android API. When the user registers on their website, the API key is generated and can be copied into the Android application. As for its content, since the Tourist Scheduler system stores its own POI information in the GDI, the activity containing the map manages POI markers and perimeters manually. The main activity also provides a search, which re-centers the map to the position of the POI that was queried. Despite handling case sensitive entries, this functionality is still basic as it only returns results if the full POI name is entered. Finally, the activity contains a navigation drawer that lists different filters to display or hide POIs based on different categories.

Other activities include the Parameters activity, which allows the users to fill in some information such as the budget, the number of travelers, the travel period, the mobility, and the starting location. The ConfirmPOIs activity, on the other hand, lists the selected POIs for review, and the user can cancel some POIs or change the recommended duration of the visit. When the schedule is sent back from the backend, the results are displayed in Directions activity. As its name suggests, it shows the results as a list of directions to move step-by-step

from one POI to another. The same result can also be shown as a route in the map.

Data Management

Most of the data is managed using the class AppData. It is a static class that initializes the list of POIs and stores them in HashMap, which uses the Objects package shown in Figure 3.7. In addition, it stores the created markers and perimeter polygons associated with POIs. Most importantly, it handles the unique instance of class Request, which, as the name suggests, gathers all input that is sent to the backend as request information.

Asynchronous Tasks

In order to communicate with the Request Handler, the Android application uses AsyncTask extensions. These classes allow the application to send asynchronous requests to the Request Handler without affecting the main thread. In fact, since the main thread is responsible for its user interface management (among other functionalities), sending a request would cause the application interface to freeze until a response is received and processed. However, the use of AsyncTask, coupled with interface classes/functions, prevents the freeze from occurring while also ensuring that functions in need of the response as input will wait until it is available. In the Tourist Scheduler application, AsyncTask is used to retrieve the POI database, which is then stored as a Map of PointOfInterest object instances in AppData, and to read the schedule that corresponds to a request.

Recycler Views

The application uses RecyclerView extensions in several parts of the system. For instance, it is used in ConfirmPOI activity to display the list of selected POIs so that the user can change their respective visit durations from a list box. It is also used in the POI info activity in order to show the list of days where the POI has a special opening schedule. However, the interesting example is in DirectionsActivity, where results are displayed as a list of places to visit separated route directions. In this case, a RecycleView checks whether a segment is a POI or a route, and then uses the appropriate user interface fragment to inflate the list.

4.2 Request Handler

Being the central part of the backend, the Request Handler consists of a simple Java server built with the `com.sun.net.httpserver` package. The server processes incoming HTTP GET requests using classes that extend the HttpHandler class, which will be covered in upcoming sections. Before the server starts, the Request Handler imports the POI entries from the GDI's POI database. It uses the GDI's Java interface and places the POIs in a static hash map that is available to all modules. The key of the hash map consists of the POI's equivalent OSM ID.

A few requirements are needed for the Request Handler to function properly. First of all, despite the portability of Java programs, the Request Handler must be hosted on a Linux distribution (see Section 5 for more about the testing laptop). This is due to the Request Handler's reliance on the Constraint Solver, as the program does not behave correctly under Windows. The second condition is to establish an SSH tunnel to communicate with the GDI, since it is not managed by the Request Handler. The SSH tunnel is therefore established manually on the terminal using the command below and upon entering the correct password. The IP address and port number refer to the GDI hosted on CityPulse's development server:

```
ssh -N -L5438:localhost:5432 tunnel@131.227.92.55 -p8020
```

4.2.1 POI List Handler

When the user launches the Android application, the main activity with the map is the first page displayed on the screen. However, the map as it is imported from the Google Maps Android API does not contain any markers or POIs. As mentioned in previous sections, the POIs are managed separately at the backend using the GDI POI database.

As a result, the Android application immediately requests the backend to send the most recent POI list. This handler is thus in charge of responding to that request, as it takes the POI map that was created at the start of the server with the GDI Java interface, and uses Gson to send back the response in JSON format.

4.2.2 Schedule Handler

This is the handler that is responsible for processing and responding to schedule requests. As Figure 3.3 depicts, its work can be divided into three major steps. First of all, it processes the parameters of the requests. The handler thus reads the parameters sent with the GET request into a hash map, which is then used to configure the elements of the Request static object.

The second step consists of preparing the data needed by the Constraint Solver. Besides some initial data that is taken from the Request object (i.e. POI number, tour time start and end), the Constraint Solver needs routing data between every pair of selected POIs. In other words, the handler needs to build a -virtual- matrix that contains time and distance information, which is done using the GDI interface (see Section 4.4). Before that, it is primordial to determine how to treat the starting place of the tour because it affects the structure of the said matrix.

$$\begin{bmatrix} 0 & 3 & 5 \\ 3 & 0 & 1 \\ 5 & 1 & 0 \end{bmatrix}$$

Matrix 1: Discarding the starting point

$$\begin{bmatrix} 0 & 2 & 6 & 1 \\ 0 & 0 & 3 & 5 \\ 0 & 3 & 0 & 1 \\ 0 & 5 & 1 & 0 \end{bmatrix}$$

Matrix 2: Including the starting point

The first option is to discard it from the matrix as illustrated in Matrix 1. This will keep the size of the matrix to nxn, where n is the number of selected POIs. It will also need a number $n^2 - n$ of calls to retrieve all the routes from the GDI, since there is no need to send calls from a POI to itself. The last step would be to fetch one more route from the starting point to the first POI to be visited, raising the total number of calls to $n^2 - n + 1$.

The second alternative, on the other hand, includes the starting point in the matrix as shown again in Matrix 2. This option implies that more route calls are needed from the starting point to all selected POIs in order to have the data ready before one of these POIs is selected as the first to visit. However, assuming the return to the starting point after the end of the tour is not a necessity, the first column of the matrix needs no processing, which takes the number of calls

to $(n+1)^2 - (2n+1)$, or n^2. While this would result in some overhead, the second alternative is much more accurate because it finds the best routes while taking the starting point into consideration. The first option might also generate a good schedule, but it is as likely to start the schedule with the POI that is the farthest from the starting point.

When the routing matrix is built, each route that was returned has its waypoints stored in a hash map for later use. When the Constraint Solver sends back its results, the handler builds the route that will be sent in the HTTP response. It generates the result using RouteSegment instances, where each segment represents a step in the trip; a visited POI, or a route between two POIs. A segment gathers location data from the Request, the visit starting time from the Constraint Solver, and the route's waypoints from the aforementioned hash map. The segment is then added to the Route object, which is finally sent back as a JSON response.

4.3 Constraint Solver

The Constraint Solver module consists of a C++ program that is implemented using the *Gecode* library. Due to the similarity of both problems, the implementation of the Tourist Scheduler model is partly inspired by the solution for the Traveling Salesman Problem provided by the *Gecode* website [17]. More specifically, it uses a very similar approach to the Circuit constraint.

4.3.1 Derived Constants

- n represents the number of POIs, including the starting point
- *dayStart* represents the time of the day where visits can start
- *dayEnd* represents the time of the day where visits have to end
- *openingTimes* lists the opening time of all POIs
- *closingTimes* lists the closing time of all POIs
- *visitDurations* lists the selected visit duration for all POIs

- *timeCost* matrix lists the travel time cost between every two POIs

4.3.2 Decision Variables

- *visitTimes* is an integer array variable that represents the starting time of n POIs. Its domain considers both the valid visit period (i.e. *dayStart*, *dayEnd*), and the schedule of the POI (i.e. *openingTimes*, *closingTimes*). Hence, **for a POI** i:

  $visitTimes_i \in \{\max(dayStart, openingTime_i), \min(dayEnd, closingTime_i - visitDurations_i)\}$.

- *successors* is an integer array variable of size n where each element (i.e. POI) takes the index of the POI to be visited next. This decision variable helps determine the sequence of the POIs that will be visited. Therefore, **for all** i in *successors*:

  $successors_i \in \{0, n-1\}$.

- *totalTimeCost* represents the total time it takes to visit all POIs and move between every two POIs. Its initial domain cannot exceed (in reality, reach) the time to visit all POIs and the total time to move between all possible combinations of POIs. Hence:

  $totalTimeCost \in \{0, \sum visitDurations + \sum timeCost\}$.

4.3.3 Redundant Decision Variables

- *succCost* is an array of size n where each element represents the cost to move from its respective POI to the next POI (i.e. $successors_i$). Therefore, **for each** i in *succCost*:

  $succCost_i \in \{0, \max(totalTimeCost)\}$.

- $p0$ represents the first POI to be visited, which is locked to the starting position of the schedule. Therefore:

  $p0 \in \{0, n-1\}$.

- *timeCostArgs* maps *timeCost* in the constraint program.

4.3.4 Problem Constraints

- CIRCUIT *constraint*: This constraint ensures that each POI element in the array decision variable*successors* will be assigned an index representing the next POI to be visited and create a Hamiltonian circuit with an optimal time cost using $timeCostArgs$. It also computes the individual time cost to the next POI in $succCost$, and the total time cost in the decision variable $totalTimeCost$.

- *No time conflict*: This constraint ensures that the program not only gives distinct visit times to the POIs, but also considers the visit duration of each POI and the time cost between two POIs. Hence, **for every two POI indexes** i **and** j:

$$visitTimes_i + visitDurations_i + timeCost_{i,j} < visitTimes_j \vee$$

$$visitTimes_j + visitDurations_j + timeCost_{j,i} < visitTimes_i$$

4.3.5 Channeling Constraints

- ELEMENT *constraint*: using ELEMENT on $p0$ and $successors$ channels the variable $p0$ to the element in $successors$ with value 0. This is the constraint that forces the schedule to start with the first POI, which also happens to be the starting position.

4.3.6 Branching Heuristics & Exploration Order

This model uses the value selection heuristic INT_VAL_MIN for the variable $totalTimeCost$. Since this choice guarantees that the first result will have the lowest cost, it implies that the search tree is explored using Depth-First Search (DFS) rather than Branch-And-Bound (BAB). As for $visitTimes$, INT_VAL_MIN is used as value selection heuristic as well, since it is preferable that visits start as soon as possible anyway, whereas the choice for the variable selection heuristic is discussed in Section 5.3.

4.4 Geospatial Data Infrastructure

The GDI is the only component that is provided by CityPulse and used in the Tourist Scheduler project. It consists of a PostgreSQL database that is enhanced with the PostGIS extension, which gives the possibility to manipulate geographic data about coordinates, paths and building shapes [15]. The GDI currently uses a modified version of Dijkstra's algorithm when looking for routes, but there are plans to alter the approach by using the A* (i.e. A-star) algorithm instead. This change should improve the accuracy of the routes and deliver a better time performance.

The Tourist Scheduler is limited to the city of Stockholm, and its road map and paths data is imported from OpenStreetMap to the GDI. In addition, another database is added at the level of the GDI in order to store custom information about the POIs of the Android application. As Figure 3.6 shows, data includes the POIs respective descriptions, contact information, schedules, and entrance fees.

4.5 Public Transportation Travel Planner

The Public Transportation Travel Planner is a simple Java class that imports routing data from an external RESTful API. It works on a per-request basis by returning one itinerary using public transport for a single request. The API relied on in this use case is Trafiklab's ResRobot, which is the only available API that provides public transport routing services from all companies in Sweden. The website `ResRobot.se` is a known Swedish website that provides public transport routing services thanks to its Travel Planner ("Reseplanerare"), and the aforementioned Trafiklab API offers the exact same service. The Java module sends an HTTP request and receives a JSON response with the result, which it processes and stores in the Route object. The typical structure of an API response can be found in [18], and the data that is extracted from the response is illustrated in Figure 4.1.

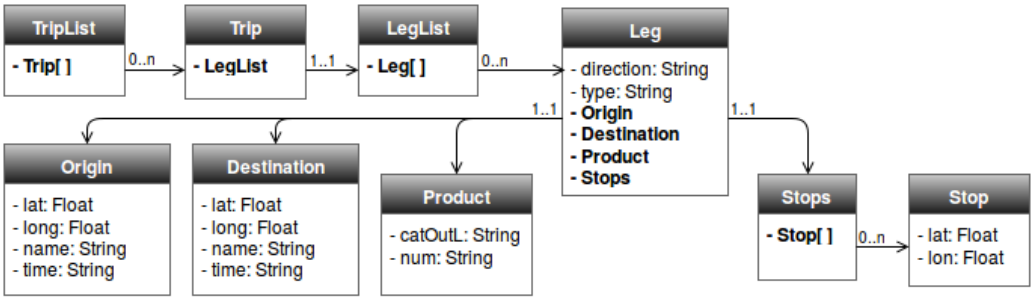

Figure 4.1: Significant data in a ResRobot API response

4.6 Event Handler

The Event Handler is a periodic module that is called by the Request Handler to fetch for new events in Stockholm. This is achieved by querying external APIs that share events in culture and sports among others. When new events are found, they are added as new entries in the GDI's POI database. Currently, as there is no available API that provides such a service, the Request Handler uses a class that reads simulated sports events from a JSON file. This module is executed once a day (i.e. every 24 hours) and should update the GDI's POI database with the new events.

Chapter 5

Testing & Evaluation

Several experiments were carried on the components of the system. Besides the overall performance of the system, there is a specific focus on the time performance of the Constraint Solver and CityPulse's GDI database and interface. All the experiments that are reported here were conducted under Linux Ubuntu 14.04LTS (32bit), Intel(R) Core(TM)2 Duo CPU T6670 @ 2.20GHz x 2 with 2GB RAM.

5.1 Overall Performance

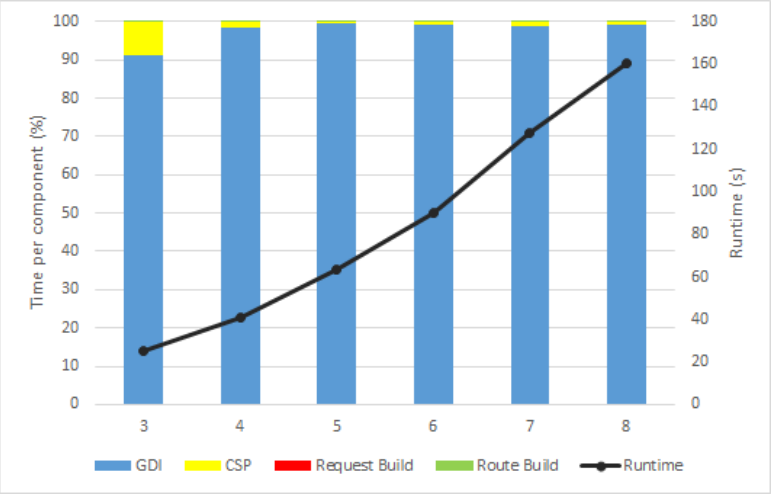

Figure 5.1: System overall performance

Figure 5.1 shows the overall performance of the backend based on the number of selected POIs. It is a black box test that computes the total time it takes a request that is sent from the Android application to be processed by the Request Handler, and for its subsequent response to be sent back. The test shows that the time required to generate the schedule is a little slow and suffers from a lack of scalability as it rapidly grows when more POIs are selected. If we compare this performance to the time it takes for tourists to plan trips with the services mentioned in Section 1, we can see a drastic improvement since it might reduce hours of searching and planning to a couple of minutes for 7 POIs. Nevertheless, the total waiting time remains quite high compared to publicly available applications and might have negative effects on its usability.

In order to know more about the possible bottleneck(s) at the level of the backend, more tests were conducted in order to extract the performance of each component separately. The bars thus display the percentage of the overall time that each component takes. As we can see, the GDI is by far the most costly component, taking between 90% and 99% of the total time, while the Constraint Solver takes the larger part of the remaining percentage. As a result, more tests were conducted on these two important backend components.

5.2 GDI

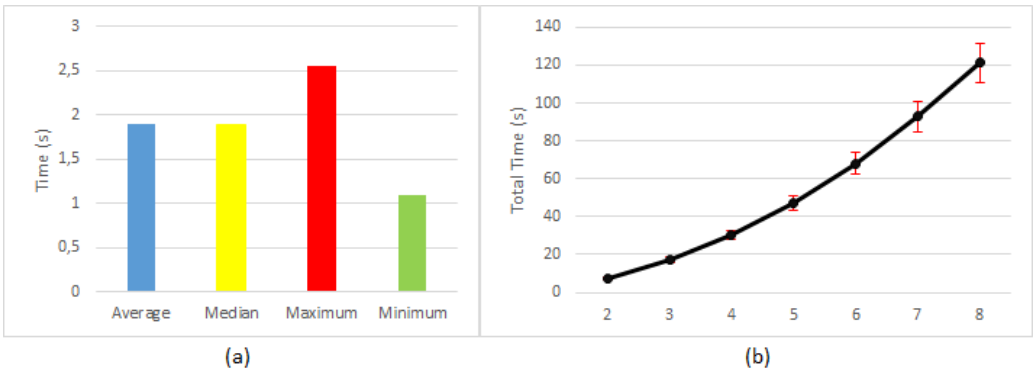

Figure 5.2: GDI time performance

Another perspective of the performance of the GDI is available at Figure 5.2. Graph (a) shows that a single GDI request takes around 1.9 seconds on average, but could also go up to

2.5 seconds on occasions. As a result, with the standard deviation being close to 0.16 seconds, Graph (b) shows that a schedule request containing 8 selected POIs needs 120 seconds for the GDI on average with a variation reaching ±10 seconds. Here, it becomes clear that the current approach of gathering routing data before selecting the best schedule using the Constraint Solver is not efficient time-wise. A possible reason to this latency could be that the PostGIS extension and the Dijkstra algorithm cost some overhead time to the GDI when sending back routing data with each query result [19]. A non-mutually exclusive explanation could be that it is the only component that does not communicate with the backend locally since it is hosted by the CityPulse development server, which is physically located in Surrey, in Ireland.

Regardless of the reason, the system can provide optimal results if this latency is reduced. While it could not be tested, it is still possible that the component yields a better performance if deployed locally with the Request Handler. Otherwise, there should be more efforts to explore the possibility of tailoring the routing queries and procedures to the Tourist Scheduler system. Another option would be to store or cache the routing results, which would avoid sending the same GDI query for every request.

5.3 Constraint Solver

Table 5.1 shows the program's performance for different numbers n of POIs. Contrary to what was witnessed with the GDI, the performance of the Constraint Solver seems extremely satisfactory and scalable, with a single execution taking at most 2 milliseconds and 12 failures. However, the program suddenly fails with 20 POIs without witnessing any increase in the execution time or failures beforehand. This failure probably implies that an appropriate schedule could not be found, so further investigation was conducted with the tests displayed in Table 5.2. In these tests, all POIs were fixed to a single visit duration, which is 60, 30, and 10 minutes, respectively. These tests prove that the Constraint Solver does not have a limit of POIs it can handle, but rather that the time range could not fit all of the selected POIs with their respective visit times and travel times.

n	Time (s)	Failures
2	0,000	0
3	0,000	4
4	0,000	6
5	0,001	6
6	0,001	6
7	0,001	6
8	0,001	6
9	0,001	5
10	0,001	12
11	0,002	12
12	0,002	12
13	0,002	12
14	0,002	12
15	0,002	12
16	0,002	12
17	0,002	12
18	0,002	12
19	0,002	12
20	Timeout	-

Table 5.1: Results for the scheduler constraint program

60-minute visits		
n	**Time (s)**	**Failures**
2	0,000	0
3	0,000	4
4	0,000	6
5	0.001	6
7	0.001	6
8	0.001	6
9	0.001	12
10	0.001	12
11	Timeout	-

30-minute visits		
n	**Time (s)**	**Failures**
2	0,000	0
3	0,000	4
4	0,000	6
7	0,000	6
8	0.001	6
9	0.001	12
15	0.001	12
17	0.002	12
18	Timeout	-

10-minute visits		
n	**Time (s)**	**Failures**
5	0.001	6
10	0.001	12
15	0.001	12
20	0.002	12
25	0.002	12
30	0.003	12
35	0.005	12
40	0.008	12
41	Timeout	-

Table 5.2: Results for the Constraint Solver with fixed visit durations

As for the choice of heuristics, Table 5.3 shows that variable selection heuristic `INT_VAR_MIN_MIN` yields a slightly better performance, handling up to 10 POIs with 60-minute visit duration, and 40 POIs with 10-minute visit duration. However, the difference remains too small to jump to conclusions as further tests should be conducted.

Heuristics	Max n	Failures
INT_VAR_MIN_MIN	10; 40	12
INT_VAR_MAX_MIN	09; 37	12
INT_VAR_ACTIVITY_MIN	09; 37	12
INT_VAR_ACTIVITY_MAX	09; 37	12
INT_VAR_SIZE_MIN	08; 35	12

Table 5.3: Maximum POIs handled per variable selection heuristic

5.4 Comparison to Existing Solutions

5.4.1 Choice of Technology

As described in the previous chapters, one of the most crucial components of the Tourist Scheduler is the Constraint Solver, which uses the Gecode library in a C++ program. The reason behind choosing constraint programming rather than answer set programming is due to the flexibility that constraints programs offer to express constraints and, if necessary, to design custom propagators and branchers. Another key difference lies in answer set programs' drastic drop in performance with a high number of rules. For example, the CityPulse framework contains the Decision Support component, which helps select the best route based on real-time data such as traffic and pollution indexes [15]. Decision Support relies on answer set programming using Clingo, and even though it generates efficient results, it takes roughly one second to respond. The Constraint Solver, as Section 5.3 shows, proved to be much faster.

5.4.2 Comparison to Algorithms

The model of the Constraint Solver has several similarities with some of the algorithms described in Section 2.1. In fact, a number of comparisons can be drawn with the Traveling Salesman Problem, most notably the fact that both require a unique visit to each vertex in

their graph representations. One small difference is that, while the solution of a TSP is a Hamiltonian cycle, the schedule generated by the Constraint Solver is a Hamiltonian path, since Section 4.2 already mentions that the time cost matrix nullifies the cost to travel back to the starting location from the last POI in the schedule (i.e. the first column in the matrix). The aim of this modification is to minimize the delay caused by the GDI since a Hamiltonian cycle would increase the number of GDI route requests from n^2 to $n^2 + n$. The main difference between the constraint model and the TSP, though, is the necessity to respect each POI's opening and closing times. In fact, the objective of the TSP is to find the journey with minimal cost assuming that its vertices can be visited anytime, whereas the Tourist Scheduler must also determine the visit time of each POI while abiding by their respective schedules.

Fitting all POI visits in the schedule is one aspect of scheduling that is not managed flexibly in the Tourist Scheduler, which is due principally to the selected trip day(s) and time(s). In fact, the tests conducted in Section 5.3 show how the current model fails to deliver any result when the number of selected POIs is too high and their schedules and durations cannot be sorted. Consequently, it would be very convenient to engage into a trade-off based on the Activity Selection Problem. In that scenario, the model could find the largest subset of POIs that can fit the total time range of the trip before attempting to find a suitable schedule. The solution may not include all the POIs that the user selected, but the assumption here is that the user would rather know that the request is unsatisfiable and receive a partial schedule than not receiving any schedule at all. As it pertains to its implementation in the Constraint Solver, this improvement can be achieved with the UNARY resource constraint although its current model in *Gecode* is not final [8].

Chapter 6

Conclusion

This thesis project was a solid effort to automate trip planning and provide a reliable service that gathers all the information needed by tourists to prepare for their trips to Stockholm. In fact, it succeeds in overcoming one of the major challenges for tourists, which is to search for information in many sources, by making everything available in one Android application.

The project also gives a good preview of what can be expected from future applications in the same sector such as Google Trips, which currently manages transportation and reservations but lacks an automated planning feature as of this writing [20]. From a business standpoint, the Tourist Scheduler could work as a standalone application with several opportunities to include partnerships. For instance, POI information could be synchronized with other services such as Yelp and TripAdvisor, just like they could be managed by their actual owners/institutions. Public transport and taxi companies could also be a potential target when it comes to routing.

The current system performance is already a significant improvement over the current procedure where tourists need to search in several sources, which could take up to several days. Most notably, the Constraint Solver delivers great results with excellent scalability and time performance. Nevertheless, the system could benefit from additional efforts to reduce the overhead caused by the GDI, an objective which can potentially be achieved with the use of A* instead of the Dijkstra routing algorithm. In addition, the system must reduce its dependence on external RESTful APIs to ensure that events and public transport are always up-to-date.

Chapter 7

Future Work

While already potentially useful with its current format, the Tourist Scheduler can be still be enhanced with additional features that would improve its efficiency and diversify its services. This section lists some of the improvements that were considered during this project but were either out of its scope, or could not be included due to the limited time.

Traffic monitoring

So far, the routing options the GDI provides do not consider the traffic situation, since the assumption is that the application can be used to plan trips ahead of time. However, with the availability of traffic data in Stockholm, it is possible to use a genetic or clustering algorithm that analyzes and monitors traffic per area and time of the day, which, in turn, would allow the algorithm to deduce or estimate the traffic situation and use it as a cost multiplicator when seaching for routes.

Parking options

Another feature of similar impact would be the inclusion of parking recommendations. In fact, the application currently satisfies user requests by showing the routes from a POI to another, but it does not give any indication about where the user can park the car, hence

implicitly assuming that it can it can be parked right next to the POI and impacting the budget estimation as well. Therefore, it is necessary to recommend parking lots to the user based on budget and proximity, which can be achieved by including all needed information in the GDI, and implementing additional constraints at the level of the Constraint Solver.

POI capacity & guided tours

Even though this system's primary objective consists of simplifying the planning aspect of a trip, it also has the potential of improving the trip experience as well. For instance, similarly to the aforementioned traffic monitoring feature, the users could benefit from keeping track of the capacity of each POI if the system could recommend different visit times depending on how many previous users were advised a certain time. For example, if too many users were advised to visit a certain museum at 3PM and its capacity is small, the application should avoid recommending that time slot for future trip requests and suggest visit times with lower affluence. Similarly, since some POIs schedule regular guided tours for free, it would be interesting for the users to visit these POIs during the guided tours. The challenge in this case would be to make these two features coexist since privileging the guided tour time would result in higher attendance at that specific time.

Tourist & transport cards

Renowned tourist destinations often offer special packages for tourists in order to take full advantage of the city. For example, there are tourist pass cards that can be purchased once to access a certain number of museums. Similarly, there are cards and offers related to public transport to reduce the hassle of purchasing tickets for every small journey. These offers are very important to the budget management side of the Tourist Scheduler, and their inclusion in the system should be one of the top priorities.

System Improvements

Aside from new features, the current system also needs many improvements to increase its performance. For example, the application can better manage the life cycle of its activities, and it can benefit from caching some of its data such as POI images and information. In addition, the map can be visually enhanced by displaying routes in a segment-based approach, which would allow the user to better follow the schedule and the paths. Other options such as storing schedule requests and their results should also be added for later access by the user.

As for the backend, the priority has to be about improving the overall structure of its components and the performance. As Figure 5.1 suggests, improving the processing time of a trip request implies investigating the GDI to reduce the time it takes to respond to queries. Besides performance, the system could certainly benefit from additional data sources for POIs and events. Indeed, events can be retrieved from more RESTful APIs, but external sources are not optimal long-term solutions as there is no guarantee about their availability in the future. The other solution could be to use the CityPulse Social Data Reader [use cited works to describe component]. Last but not least, because it was out of the scope of this project, a better implementation of the server should be considered, with alternatives like NGINX offering several advantages, including its asynchronous architecture and low resource consumption [21].

Appendix A

Installation Instructions

A.1 System requirements

Request Handler

- Linux operating system (tested with Ubuntu 14.04 LTS)
- Java JRE 1.8.0_91 or higher
- Gecode version 4.4 or higher

Android Application

Minimum Version	Android 5.0 Lollipop
Minimum SDK Version	API level 21
Target Version	Android 6.0 Marshmallow
Target SDK Version	API level 23
Google Play Services	Version 9.0.2

Table A.1: Technical specifications for the Android application

Table A.1 summarizes all the technical requirements for the Android Application. The user also has the option to enable location services in order to use the current position as the starting point of a schedule.

A.2 Running the System

Request Handler

1. Establish an SSH tunnel from the terminal. The password can be found in the GDI Java interface, which is available in the CityPulse Development SVN repository:
   `ssh -N -L5438:localhost:5432 tunnel@131.227.92.55 -p8020`
2. Make sure that port 8001 is available wherever the backend is to be hosted
3. Run the file Request-Handler.jar

Android Application

The Android application only requires its APK to be installed on the phone. Before running the application, the user must make sure that the phone has access to the internet.

Bibliography

[1] Tourism Industry in Sweden: Sector Overview. *BusinessSweden*. Available at `http://www.business-sweden.se/globalassets/invest-new/reports-and-documents/tourism-sector-overview.pdf`. Accessed 2016-08-13

[2] Open Stockholm Portal. *CityofStockholm*. Available at `http://dataportalen.stockholm.se/dataportalen`. Accessed 2016-08-13

[3] Rahman M.S., Kaykobad M., Firoz J.S.. New sufficient conditions for Hamiltonian paths. *DepartmentofCSE, BUET*. Available at `ieeexplore.ieee.org/iel7/6504850/6509698/06509716.pdf`. Accessed 2016-09-15

[4] J.L. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSAJ.Comput.*, vol. 4, pp. 387411, 1992.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest. Introduction to Algorithms. *Cambridge, MA : TheMITPress*, 3rd ed., 2009.

[6] Baker S.L.. Critical Path Method (CPM). *UniversityofSouthCarolina*. Available at `http://hspm.sph.sc.edu/Courses/J716/CPM/CPM.html`. Accessed 2016-09-15

[7] Borning A., Freeman-Benson B., Wilson M.. Constraint Hierarchies. *LispandSymbolicComputation*, Vol.5 No.3. September 1992. pages 223-270.

[8] Schulte C., Tack G., Lagerkvist M.Z.. Modeling and Programming with Gecode. 2015. pages 1-3.

[9] Flener P., Carlsson M., Schulte C., "Constraint Programming in Sweden", *IEEEIntelligentSystems*, Vol.24, No.2, pages 87-89, March/April 2009.

[10] Gebser M., Schaub T., Thiele S.. GrinGo: A New Grounder for Answer Set Programming. *UniversityofPotsdam*. Available at http://www.cs.uni-potsdam.de/wv/pdfformat/gescth07a.pdf. Accessed 2016-09-15

[11] Gebser M., et al.. A User's Guide to gringo, clasp, clingo, and iclingo. *UniversityofPotsdam*. Available at https://www.cs.utexas.edu/users/vl/teaching/lbai/clingo_guide.pdf. Accessed 2016-09-15

[12] Gebser M., et al.. The Conflict-Driven Answer Set Solver clasp: Progress Report. *UniversityofPotsdam*. Available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.157.9903&rep=rep1&type=pdf. Accessed 2016-09-15

[13] Gebser M., Ostrowski M., Schaub T.. Constraint Answer Set Solving. *UniversityofPotsdam*. Available at http://www.cs.uni-potsdam.de/wv/pdfformat/geossc09a.pdf. Accessed 2016-09-15

[14] CityPulse: Real-Time IoT Stream Processing and Large-scale Data Analytics for Smart City Applications. *CityPulse*. Available at http://www.ict-citypulse.eu/. Accessed 2016-09-15

[15] Mileo A. et al.. User-Centric Decision Support in Dynamic Environments. *CityPulse*. Available at http://cordis.europa.eu/docs/projects/cnect/5/609035/080/deliverables/001-609035CITYPULSED52renditionDownload.pdf. Accessed 2016-10-01

[16] Introduction - Material Design Guidelines. *Google*. Available at https://material.google.com/. Accessed 2016-08-13

[17] Gecode TSP class reference. *Gecode*. Available at http://www.gecode.org/doc-latest/reference/classTSP.html. Accessed 2016-08-13

[18] ResRobot Reseplanerare - Sk Resa. *Trafiklab*. Available at https://www.trafiklab.se/api/resrobot-reseplanerare/resrobot-reseplanerare-sok-resa. Accessed 2016-10-01

[19] Chapter 4: Using PostGIS. *PostGIS*. Available at http://postgis.net/docs/manual-1.3/ch04.html. Accessed 2016-10-01

[20] Official Google Blogs: See more, plan less - try Google Trips. *Google*. Available at `https://googleblog.blogspot.se/2016/09/see-more-plan-less-try-google-trips.html`. Accessed 2016-10-10

[21] NGINX Wiki Documentation. *NGINX*. Available at `https://www.nginx.com/resources/wiki/`. Accessed 2016-10-10