FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Interactive Maps for Visualizing Optimal Route Planning Strategy

A case study of Tokyo Disneyland

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Elitza Vasileva
Matrikelnummer 01426939

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Projektass. Hsiang-Yun Wu, PhD

Wien, 26. Oktober 2018

Elitza Vasileva	Hsiang-Yun Wu

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Interactive Maps for Visualizing Optimal Route Planning Strategy

A case study of Tokyo Disneyland

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Elitza Vasileva
Registration Number 01426939

to the Faculty of Informatics

at the TU Wien

Advisor: Projektass. Hsiang-Yun Wu, PhD

Vienna, 26th October, 2018

Elitza Vasileva	Hsiang-Yun Wu

Erklärung zur Verfassung der Arbeit

Elitza Vasileva
Meißnergasse 22
1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 26. Oktober 2018

Elitza Vasileva

Danksagung

Zu aller erst will ich meiner Betreuerin Hsiang Yun Wu danken, die mich während der vollen Zeit meiner Bachelorarbeit unterstützt hat. Sie hat mir viele nützliche Tipps und Vorschläge nicht nur über die Implementierung und die verschiedene Algorithmen, sondern auch für das wissenschaftliches Schreiben gegeben. Ihre Unterstützung hat mir sehr geholfen, konzentriert zu bleiben und stets in die richtige Richtung zu arbeiten. Ihre Geduld und Ihr freundliches Verhalten haben dazu beigetragen eine angenehme Arbeitsatmosphäre zu bilden, welche essentiell für die Lösung komplizierter Probleme war.

Ich bedanke mich auch bei meinen Eltern, die mir durchgehend moralische Unterstützung gegeben haben und mich immer wieder motiviert haben meine Träume zu verwirklichen. Sie sind auch Diejenigen, die mir mein Auslandsstudium ermöglicht, und mich stets auf meinem Weg begleitet haben.

Weiters, will ich allen KollegInnen und FreundInnen aus der Universität danken. Wir waren ein Team und haben immer einander unterstützt. Sie haben mir auch in schwierigen Zeiten sehr geholfen und so einen großen Beitrag an meinem Erfolg geleistet haben.

Ich möchte einen großen Dank an Miroslav Kovatchev aussprechen, der mir bei der Findung essentieler Fehler im Programmcode geholfen hat.

Acknowledgements

First of all, I would like to thank my supervisor Hsiang Yun Wu, who has supported me the whole time throughout my work on the thesis. She gave me a lot of useful tips and suggestions not only for the implementation and the different algorithms but also for the written part. This helped me to be concentrated only on the most significant and to stick in the right direction. Her patience and friendly behavior also contributed to creating a pleasant working atmosphere, which was crucial in solving complicated problems.

I also want to thank my parents, who always gave me a moral support and pushed me to never give up in chasing my goals and dreams. They are also the people who enabled my studies in the first place and stood by me since the very first day at the university.

Furthermore, I would like to express my thanks to all my colleagues and friends from the university. We were one team with them and always supported each other. They helped me a lot in difficult times and this is one of the reasons why I went so far.

I would also like to express special thanks to Miroslav Kovatchev, who also gave me a hand several times by helping me to find essential mistakes in the source code.

Kurzfassung

Es gibt zahlreiche alltägliche Situationen wie Veranstaltungen, Konzerte, Sehenswürdigkeiten, Attraktionsparken usw., die die Besucher fordern, sich vor langen Schlangen einzureihen und Stunden mit warten zu verbringen. Ein Beispiel dafür sind die Disneyland Resorts. Sie sind alle sehr berühmt und ziehen jeden Tag eine große Anzahl an Touristen an. Aus diesem Grund, kann das Einreihen vor Attraktionen sehr zeitaufwändig sein - bis zu einige Stunden. Trotzdem, wird Disneyland von Millionen von Touristen jährlich besucht [sta]. Um so ein langes Warten zu vermeiden jedoch, müssen die Touristen einen Plan im Vorhinein machen - wann und in welcher Reihenfolge die gewünschte Attraktionen zu besuchen. Allerdings, es kann sehr viel Zeit im Anspruch nehmen so einen Plan zu erstellen. Es ist einerseits schwierig and andererseits kann sogar unangenehm sein, da viele Vorbedingungen miteinbezogen sein sollen. Im Hinblick darauf, der Zweck dieser Arbeit ist eine assistierende Applikation zu entwickeln. Ihre Ziele sind den Benutzern die Erstellung eines Plans für eine zukünftige Besuch in Tokyo Disneyland zu ermöglichen und erleichtern. Die Applikation besteht aus zwei Hauptkomponenten. Erstens, aus einem Optimisierungsalgorithmus, das einen Optimisierungsweg von den ausgewählten Attraktionen ausrechnet, sowie auch die Visualisierung dieser Wege, um die Attraktionen leichter zu finden. Dadurch wird die Zeit, die fürs Einreihen und vorlüfiges Planen verbraucht wird, gespart. So eine Methode wird es für die Touristen leichter machen, so viele Attraktionen wie möglich in einem einzigen Tag zu besuchen und auf diese Weise das Maximum von dem Besuch zu erreichen.

Abstract

There are many situations in our everyday life like events, concerts, landmarks, attraction parks, etc. that often require from visitors to line-up in front of long queues and thus spend hours in waiting. An example of that are the Disneyland amusement parks. They are all very popular and attract a significant number of people every day. For this reason, the lining-up in front of attractions may cost much time – even up to a couple of hours. Despite that, the Disneyland parks are visited by millions of people every year [sta]. So to avoid so much waiting they need to make a plan in advance – when and in which order to visit the wanted attractions. However, to make such a plan, it could be very time consuming, difficult and even unpleasant, because many prerequisites need to be considered in advance. Having the main problems and annoyances described, the goal of this thesis is to create an assisting application. Its purpose is to give the visitors the possibility to create their own plan for their visit to Tokyo Disneyland. It contains two main assisting components. Firstly, an optimization algorithm calculating an optimized route of the chosen attractions as well as a route visualization for an easy attraction finding. Both will reduce the time for lining-up and pre-planning. Such a technique will make it easier for visitors to see as many attractions as possible for a single day and thus, make the most of their visit.

Contents

CHAPTER 1

Introduction

1.1 Overview

This thesis is concentrated in observing and examining the problems regarding waiting times for different events, landmarks, facilities, etc. The introduction describes the existing problems in details and explain the motivation behind the searching for solutions of these problems. Furthermore, the different goals of the project is set and explained, containing the contributions of the work. In the end there is a section describing the whole structure of the thesis. This thesis would examine the problem by studying it in the case of the amusement park Tokyo Disneyland.

1.2 Background

Since the creation of amusement parks that serve as a type of entertainment, they gained much popularity, especially the Disneyland parks. They are delighted to have thousands of visitors every day. Despite the enormous size of the parks, the massive attendance of visitors can lead to unpleasant and long queues in front of the attractions. This will decrease the enjoyment of the visitors and restrict them in visiting only a few attractions for one day. This problem concerns the owners of the parks and drives them to look for solutions. One of the improvements, helping to reduce this problem is the FastPass tickets system. They allow tourists to visit attractions without having to wait for so long on the queues. This tickets, though, are limited and are available for only some of the attractions, which means that the problem with the line-up policy remains. This thesis will be about Tokyo Disneyland and the algorithms developed will be tested with its attractions and data.

1.3 Motivation and Goals

People who want to ride as many attractions as possible, need to make a plan in advance. They have to take into consideration in which order they have to visit the attractions so that they can reduce the waiting times. However, such a plan could be very complex since it requires many preconditions to be taken into account. One of them is the day of the visit and the weather – depending on that the amount of visitors varies. Another important issue is the hours for visiting each attraction. Depending on the time of the day, the size of the queues in front of each attraction is different. Since the visitors have no information about that, they will probably not be able to make a good plan, that is better than a random choice of attractions.

Therefore one of the goals of this project is to develop an optimization algorithm, calculating an optimal route for a given set of attractions. This algorithm will be based on data collected by Tokyo Disneyland. The data contains information about the waiting times for each attraction measured in the previous seven years – from November 2011 to February 2018. The waiting times are stored every 30 minutes. Using this data, the algorithm will try different permutations of the attractions, and this way calculate an optimal one. Due to the opening times of Tokyo Disneyland and the calculation time of the algorithm, the visitors should select not more than a certain number of attractions. An important aim is to develop such an optimization algorithm that the computation time is as minimal as possible and the total time for visiting the selected attractions is also minimal.

Despite knowing an optimal route for visiting the selected attractions, another issue is the enormous size of the park. A visitor should find the attractions as fast as possible without getting lost so that he/she can stick to the plan. However, this could be a very challenging task to achieve with only a static map. The user would need a more sophisticated visualization that can be interactive and change according to his/her preferences. The visualization should be a 2D map of the Tokyo Disneyland park displaying all the paths, buildings of attractions as well as the rest of the facilities. Nevertheless, there are some difficulties in creating a good visualization, since drawing the paths between different attractions at the same time could lead to intersections and overlaps of these paths. Consequently, the second leading goal of the project is to develop a user-friendly visualization minimizing the intersections of paths. Furthermore, an animation drawing the paths when selected should be implemented to support a better orientation and coordination of the visitors.

A user interface will support the user in interacting with the application. At the start of the application, the user will be able to select the date of visit as well as the set of the wanted attractions. In the visualization phase, the user will also have the possibility to interact with the application by selecting different attractions and thus display an animation of the path leading to the selected attractions.

Initially, the idea was to develop a stand-alone application. Although the screen of the application would have been bigger and with a higher resolution, the user could only use

it on a PC or a laptop. However, the application should be used in real time, which is why it was decided to develop it for smartphones in Android. This way, the application can support the visitors when they are at Tokyo Disneyland.

1.4 Contribution

The main contribution of this thesis can be split into two major parts:

- The improvement of an already existing optimization algorithm for a possibly optimal route finding, and
- A dynamic visualization of the founded optimal route in the style of a metro-map layout – reducing the number of crossings using a greedy algorithm

1.5 Structure of the Thesis

The structure of this thesis is as follows: Chapter 2 gives an overview of existing algorithms and related work on route optimization and visualization. Chapter 3 is the central part of the project and explains the methodology of the presented approach. The implementation including the system specifications, the platform and the external libraries that are used is described in Chapter 4. Chapter 5 represents the results and evaluation of the developed algorithms, as well as a description of the limitation and performance of the application. The final chapter - Chapter 6 - concludes the thesis and discusses possibilities for future work.

CHAPTER 2

Related Work

There are several different optimization algorithms, as well as some visualization methods to implement and develop. This chapter introduces some of the most well-known already existing algorithms regarding the topic and describes how they will be useful for this technique. It also gives an overview of how they could be adjusted and modified to solve the present problems. The chapter is separated into two major parts. The first part will present different studies and solutions about the line-up policy in attraction parks. The second part will focus on already existing algorithms and optimization approaches about visualizing routes that overlap and cross each other.

2.1 Algorithm Optimizing the Path Between Attractions

There have been several various works, focusing in the field of optimizing the route planning for amusement parks and especially for Tokyo Disneyland. One of the first works regarding this problem is presented by Shibuya et al. [SOO13]. Their goal is to minimize the total traveling time for visiting a maximum of 8 attractions. Their approach is based on the Travelling Salesman Problem (TSP) [SSS17], [Lap92]. TSP is an algorithm for finding the shortest route from a given list of cities and their pairwise distances by visiting each one of them only once and returning to the origin city. The pairwise distances calculation is done by using the Dijkstra Algorithm [Dij59], which finds the minimum path between any two points/cities. Their algorithm, however, is only limited to 8 attractions and cannot work in real time when having a bigger number of attractions. This is because the TSP is an NP-hard problem [Bru13]. With a growing number of cities, the time for calculation increases factorially.

A more sophisticated approach is presented by Ohwada et al. [OOK13]. Their algorithm is based on the algorithm of Shibuya et al. [SOO13] However, they use a branch and bound method to restrict the possible routes [MSJ16]. They add a further variable

identifying if there is a path between two attractions or not. Equation 2.1 shows the different components that are used to minimize the total time required.

$$\sum_{(i,j)\in I} (M_{ij} + W_{it} + P_i)X_{ij} \rightarrow min \tag{2.1}$$

I describes a set of attractions, while T stays for a set of time zones. The M_{ij} in Equation 2.1 represents the transit time from one attraction to the other, where $i \in I$ and $j \in I$. The waiting time of attraction $i \in I$ at time $t \in T$ is described by W_{it}. P_i represents the duration time of attraction $i \in I$. X_{ij} accepts only binary values - 0 and 1 and represents the branching of the presented algorithm. It defines if there is a path between any two attractions, where 1 stays for: "There is a path", and 0 means: "There is no path."

Ohwada et al. have developed three different algorithms each one focusing on a different aspect of finding an optimal route. The first algorithm is called an NT algorithm and calculates the optimal route without using the Fastpass system in Tokyo Disneyland. Fastpass gives the visitors an entry priority for riding attractions by paying additionally [GHMP13]. The second algorithm is more complicated and includes the usage of Fastpass Tickets for the attractions that use that system. The third algorithm takes into consideration aspects such as relaxation time, regarding the fact that the visitors should not visit two or more too thrilling and exciting attractions in a row.

The algorithm in this bachelor thesis will be based on the NT algorithm, which can be seen in Figure 2.1. The algorithm goes through all possible permutations of the given set of attractions if there is a path, finds the path with minimal total time and stores the order of attractions as a result.

2.2 Minimization of Path Crossings and Overlaps

Nowadays the visualization and representation of different paths and routes are essential for many navigational applications. However, visualizing paths in a way that is most clear and easy to follow, can be very challenging. Many factors influence the layout of the visualization. One of the main problems are the intersections and overlaps that may occur if several of the different paths that are displayed have common edges or nodes. The crossings and overlaps of lines should be reduced to a minimum. This problem is a very well-known problem when drawing a graph and has been extensively studied in the graph drawing literature.

One of the most common fields in which this problem occurs is when drawing a metro map layout [HMdN04]. The problem is defined as the *metro-line crossing minimization problem (MLCM)* [BNUW07], [BKPS08]. The metro map is represented as an undirected embedded graph $G = (V, E)$, where G represents the underlying network. V stays for the set of the different metro stations and E refers to the railway lines connecting these stations. Furthermore, there are paths of G which are referred to as lines and are

```
1.  PROCEDURE TRAVEL
2.      MIN_PATH ← ϕ;
3.      min_time ← ∞;
4.      INITIAL_LIST ← ATTRACTIONS - {goal};
5.      INITIAL_PATH ← <goal>;
6.      SEARCH(goal, INITIAL_LIST, 0, INITIAL_PATH);
7.  END

8.  PROCEDURE SEARCH(attraction, REMAIN_LIST, time, PATH)
9.      IF REMAIN_LIST is empty THEN
10.         arrival_t ← time + distance(attraction, goal);
11.         add goal to the end of PATH;
12.         IF arrival_t < MIN_TIME THEN
13.             MIN_PATH ← PATH;
14.             MIN_TIME ← arrival_t;
15.         END IF
16.     ELSE
17.         FOR each x in REMAIN_LIST
18.             arrival_t ← time + distance(attraction, x);
19.             t_getting_off ← arrival_t + wait(x,arrival_t) + t_enjoying(x);
20.             P_STACK ← PATH;
21.             add x to the end of P_STACK;
22.             L_STACK ← REMAIN_LIST-{x};
23.             SEARCH(x, L_STACK, t_getting_off, P_STACK);
24.         END FOR
25.     END IF
26. END
```

Figure 2.1: NT algorithm by Ohwada et al. [OOK13] computing an optimal path for a set of attractions using branch and bound.

represented as a set $\mathcal{L} = \{l_1, l_2, ..., l_k\}$. Each one of these lines l_i is represented by a sequence of edges $e_1 = (v_0, v_1), ..., e_d = (v_{d-1}, v_d)$ of G. The first node v_0 and the last node v_d represent the start and end stations of the metro line and are called terminals. Many different metro lines can visit the same edges and nodes. The MLMC problem aims to draw the different lines in a way that the number of intersections and overlaps among the layout is minimized.

Hong et al. [HMdN04] describe the metro map layout problem in their work and present a method to produce a good layout of the metro map automatically. They define a good layout of the metro map, which has specific criteria. In a good layout of a metro map, each line has to be drawn as straight as possible, and the paths should be drawn horizontally or vertically, but some of them could be at 45 degrees. Furthermore, each of the lines should have a unique color and there should not be any edge crossings

and overlapping of labels. They try five different layout methods by using a variety of combinations of spring algorithms [EL00]. Spring algorithms are tools that can be very helpful for visualizing undirected graphs. Applying them is useful to display symmetric properties of graphs.

Another solution to the problem is described in the paper by Martin Nöllenburg [Nöl10]. He develops an improved algorithm for the MLMC problem which has a running time of $O(|\mathcal{L}|^2 \cdot |V|)$, where $\mathcal{L}$ is the set of paths (or lines) that cover G and V is the set of nodes (metro stations).

A further interesting approach for automatic drawing of a metro map layout is described by Stott et al. [SRMOW11]. They use a multicriteria optimization in order to achieve that. They use a weighted sum of a fitness value for the layout of the map that depends on some different metrics. Stott et al. apply different clustering techniques to the map, such as a hill climbing optimizer, which helps to produce a good layout of the metro map.

CHAPTER 3

Methodology

This chapter presents the different methods and algorithms, which have been developed to solve the problem for Tokyo Disneyland. It involves the creation of a road network for the paths, different methods, data structures and approaches for the optimization algorithms and their visualization as well as a description of the user interface and the interaction with the application. The application name is "OptiRoute".

3.1 Preliminaries

To start with, several preliminaries and assumptions need to be taken into consideration in advance when developing the application. Their purpose is to reduce complexity and lay the foundations of the different algorithms.

These assumptions are:

1. The visitor's main goal is to visit as many attractions as possible for one day without counting time for relaxation as in the paper of Ohwada et al. [OOK13]. The idea is to make the maximum use of the stay in Tokyo Disneyland.
2. The visitor will always start and end his journey at the entrance of the park.
3. The visitor visits each attraction only once, without repeating attractions several times. For the route optimization algorithm, this means that each node should be visited exactly only once.
4. All the attractions in Tokyo Disneyland are open regardless of the date of visit specified by the user.
5. It will be taken for granted that the working time of the park is always from 8 a.m. to 10 p.m. and that all attractions are opened during this interval. This means that the visitors have a maximum of around 14 hours to spend in the park.

6. The average walking speed needed for the calculation of the time for transportation between attractions also has to be predefined and fixed. A good choice is 3km/h since it is preferable that the people are not too much in a hurry when walking from one to the other attraction.

7. The visitors will always start their tour through the attractions at the same time in the morning (for example when the attractions park opens or at a fixed time such as 8:15 o'clock).

8. The Fastpass Tickets system of Tokyo Disneyland, allowing the visitors to visit attractions with a minimum waiting, would not be taken into consideration for the development of the optimization algorithm.

9. Some of the attractions in Tokyo Disneyland have no specified duration time. For these attractions, an average duration time of 10 minutes is taken.

10. The visitor visits each attraction only once, without repeating attractions several times. For the optimization algorithm of the path order, this means that each node should be visited only once.

3.2 Overview of the Present Approach

Figure 3.1 gives an overview of the workflow of OptiRoute and all major and essential steps that occur while interacting with it. Initially, there are two views requiring input information from the user, which is needed for the subsequent calculations. These two views can be seen in Figure 3.1 in the field called "User Input Parameters". The user needs to select the date of visit as well as the desired attractions to visit. The desired attractions can be selected in any order since this does not influence the final result. This information is then used for calculating the optimization algorithm responsible for finding an optimal route for visiting the selected attractions. The resulted order of the attractions is then passed on to the next algorithm which purpose is to calculate the offset of paths and reduce overlaps. Finally, the computed information is used to give a visual representation of the calculated data in the previous steps.

3.3 Building A Road Network and Pairwise Shortest Paths

This section is fundamental for both: the optimization of the routes as well as the generation of the road network and its visualization. Generation of road network means the creation of a single connected graph on the map containing all the coordinates of the paths between attractions, as well as the storage of various map data, needed for further calculations.

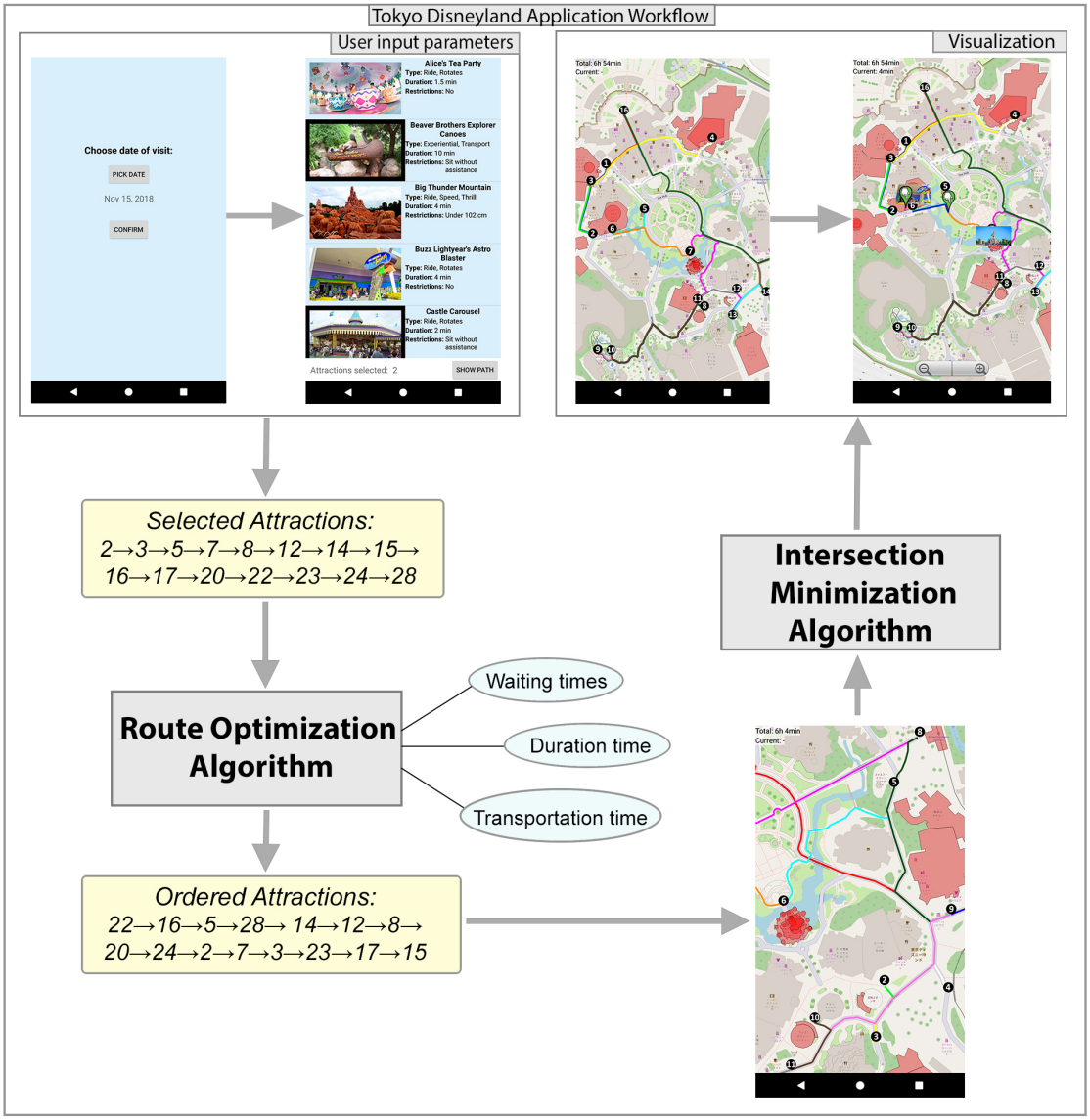


Figure 3.1: The whole workflow of the Tokyo Disneyland Application.

3.3.1 Routes as a Single Connected Graph

The routes connecting the different attractions need to be represented as a single connected graph in order to be able to calculate the pairwise shortest paths with the Dijkstra Algorithm afterward. The data obtained from OpenStreetMap contains only single nodes or a group of nodes representing coordinates in the territory of Tokyo Disneyland. Each coordinate has a tag consisting of key and value that clarifies what part of the map it represents. This could be a building, a footway, a way for pedestrians, a bridge, a garden, etc. Some of the different keys and the values they can have are represented in Table 3.1. For the creation of a single connected graph of the paths only coordinates with the key = "*highway*" and value = "*footway*" or "*pedestrian*" should be extracted. As already mentioned the nodes only contain a group of coordinates, which means that the start and the end coordinate of different groups might be identical and need to be taken only once.

Key	Value
amenity	"restaurant" ,"theater", "cafe", "ice cream", "fast food", "toilets", "parking"
shop	"gift", "toys", "jewelry", "photo", "perfumery", "clothes"
highway	"pedestrian", "footway", "service"
barrier	"gate", "fence", "retaining wall", "hedge", "wall"
railway	"switch", "narrow gauge", "monorail", "disused"
building	"yes", "roof", "retail"
landuse	"forest", "construction"
natural	"tree", "water"
leisure	"garden"

Table 3.1: The different keys and values that each coordinate (node) has in the .osm file of Tokyo Disneyland

For the representation as a graph, the data structure adjacency list was used. In this case, the adjacency list to adjacency matrix is preferred, because in an adjacency matrix, regardless of the number of edges, for each pair it stores if there is an edge or not. The complexity of an adjacency matrix is $\mathcal{O}(n^2)$ and of an adjacency list is $\mathcal{O}(n+m)$, where n stands for the number of nodes and m for the number of edges. Since most of the nodes (coordinates) are connected to only two edges and the graph is sparse, using an adjacency list is the better choice [gra].

3.3.2 Calculation and Storage of Paths

The next important issue is the computation of the minimum pairwise distances between any pair of two attractions. The very first step is to store the coordinates of each attraction, which were taken manually from the official site of OpenStreetMap [ope]. Except for the coordinates of attractions, there is one further point that has to be stored – the coordinates of the entrance of Tokyo Disneyland. The entrance is needed for both:

the route optimization algorithm and the visualization, since all visitors start and end their trip there.

Unfortunately, the exact entry of each attraction cannot be obtained from the map. That is why approximate coordinates of the paths that are most close to the attraction itself were taken from the map. These coordinates exactly, however, may not have been stored in the adjacency list, because they could differ since they are taken manually. For this reason, it is essential to determine, which are the coordinates stored in the adjacency list that are closest to the coordinates from the paths that represent the attractions. The calculation is done using the shortest distance between the coordinate representing an attraction and all coordinates from the adjacency list. It is important to mention that all path coordinates lay on corners – this means only on edges different from 180°. However, some special cases again need to be processed manually. These are the cases when the attraction is positioned between two points. This means that for these attractions an additional point should be stored manually.

After having determined the path positions of the attractions on the map, the next step is to calculate the minimum pairwise distances. This is done using the Dijkstra Algorithm. The algorithm is executed for every single attraction, including the entrance, and it calculates its smallest distances to all other attractions. As a result, in the end, the minimum path between any two attractions is stored twice: for both directions. This makes further calculations easier since there is no need to reverse the direction of the coordinates each time the attractions are visited in the opposite order.

3.3.3 Storage of Pairwise Distances

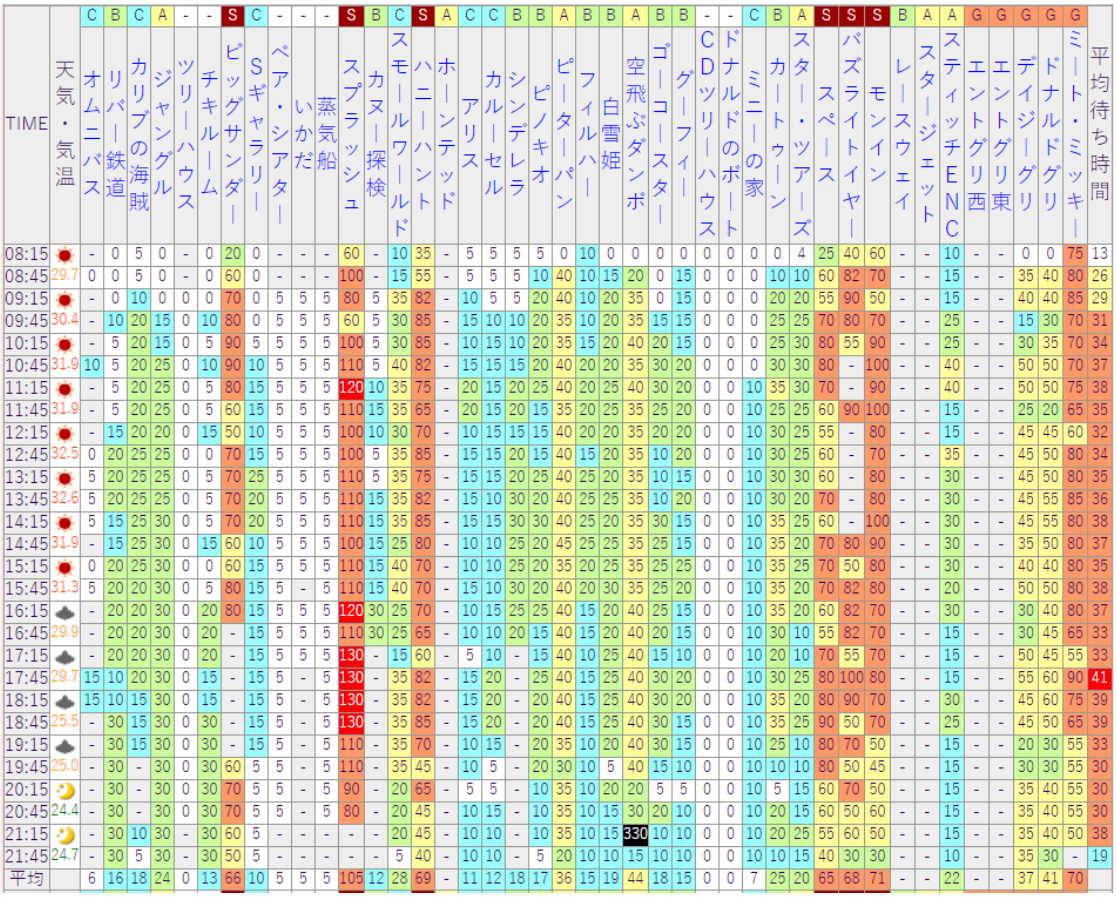
Finally, the pairwise distances between attractions have to be computed. Using the stored minimum paths, each coordinate is being traversed and the distance between two consecutive coordinates calculated and summed up.

3.4 Route Optimization

This optimization algorithm is one of the two main contributions of this bachelor thesis. Its main purpose is to calculate the most efficient route between the number of selected attractions, reducing the total time needed for visiting them. The result is a specific order in which the given attractions have to be visited in order to save time. Crucial information for calculating this algorithm are the waiting times, the transportation times and the duration of the attractions.

3.4.1 Preliminary Calculations Using External Data

Waiting Times
The waiting times are obtained from a website, storing data about the number of visitors on each day of the year [wai]. The data contains the following information - for each day of the previous years and on every thirty minutes the average waiting time for each



TIME	天気・気温	オムニバス	リバー鉄道	カリブの海賊	ジャングル	ツリーハウス	チキルーム	ビッグサンダー	Sギャラリー	ペア・シアター	いかだ	蒸気船	スプラッシュ	カヌー探検	スモールワールド	ハニーハント	ホーンテッド	アリス	カルーセル	シンデレラ	ピノキオ	ピーターパン	フィルハー	白雪姫	空飛ぶダンボ	ゴーコースター	グーフィー	CDツリーハウス	ドナルドのボート	ミニーの家	カートゥーン	スター・ツアーズ	スペース	バズライトイヤー	モンインン	レースウェイ	スタージェット	スティッチENC	エントグリ西	エントグリ東	デイジーグリ	ドナルドグリ	ミート・ミッキー	平均待ち時間
		C	B	C	A	-	-	S	C	-	-	-	S	B	C	S	A	C	C	B	B	A	B	B	A	B	B	-	-	C	B	A	S	S	S	B	A	A	G	G	G	G	G	
08:15	☀	-	0	5	0	-	0	20	0	-	-	-	60	-	10	35	-	5	5	5	5	0	10	0	0	0	0	0	0	0	0	4	25	40	60	-	-	10	-	-	0	0	75	13
08:45	29.7	0	0	5	0	-	0	60	0	-	-	-	100	-	15	55	-	5	5	5	10	40	10	15	20	0	15	0	0	0	10	10	60	82	70	-	-	15	-	-	35	40	80	26
09:15	☀	-	0	10	0	0	0	70	0	5	5	5	80	5	35	82	-	10	5	5	20	40	10	20	35	0	15	0	0	0	20	20	55	90	50	-	-	15	-	-	40	40	85	29
09:45	30.4	-	10	20	15	0	10	80	0	5	5	5	60	5	30	85	-	15	10	10	20	35	10	20	35	15	15	0	0	0	25	25	70	80	70	-	-	25	-	-	15	30	70	31
10:15	☀	-	5	20	15	0	5	90	5	5	5	5	100	5	30	85	-	10	15	10	20	35	15	20	40	20	15	0	0	0	25	30	80	55	90	-	-	25	-	-	30	35	70	34
10:45	31.9	10	5	20	25	0	10	90	10	5	5	5	110	5	40	82	-	15	15	15	20	40	20	20	35	30	20	0	0	0	30	30	80	-	100	-	-	40	-	-	50	50	70	37
11:15	☀	-	5	20	25	0	5	80	15	5	5	5	120	10	35	75	-	20	15	20	25	40	20	25	40	30	20	0	0	10	35	30	70	-	90	-	-	40	-	-	50	50	75	38
11:45	31.9	-	5	20	25	0	5	60	15	5	5	5	110	15	35	65	-	20	15	20	15	35	20	25	35	25	20	0	0	10	25	25	60	90	100	-	-	15	-	-	25	20	65	35
12:15	☀	-	15	20	20	0	15	50	10	5	5	5	100	10	30	70	-	10	15	15	15	40	20	20	35	20	20	0	0	10	30	25	55	-	80	-	-	15	-	-	45	45	60	32
12:45	32.5	0	20	25	25	0	0	70	15	5	5	5	100	5	35	85	-	15	15	20	15	40	15	20	35	10	20	0	0	10	30	25	60	-	70	-	-	35	-	-	45	50	80	34
13:15	☀	5	20	25	25	0	5	70	25	5	5	5	110	5	35	75	-	15	15	20	25	40	25	20	35	10	15	0	0	10	30	30	60	-	80	-	-	30	-	-	45	50	80	35
13:45	32.6	5	20	25	25	0	5	70	20	5	5	5	110	15	35	82	-	15	10	30	20	40	25	25	35	10	20	0	0	10	30	20	70	-	80	-	-	30	-	-	45	55	85	36
14:15	☀	5	15	25	30	0	5	70	20	5	5	5	110	15	35	85	-	15	15	30	30	40	25	20	35	30	15	0	0	10	35	25	60	-	100	-	-	30	-	-	45	55	80	38
14:45	31.9	-	15	25	30	0	15	60	10	5	5	5	100	15	25	80	-	10	10	25	20	45	25	25	35	25	15	0	0	10	35	20	70	80	90	-	-	30	-	-	35	50	80	37
15:15	☀	0	20	25	30	0	0	60	15	5	5	5	110	15	40	70	-	10	10	25	20	35	20	25	35	25	25	0	0	10	35	25	70	50	80	-	-	30	-	-	40	40	80	35
15:45	31.3	5	20	20	30	0	5	80	15	5	-	5	110	15	40	70	-	15	10	30	20	40	20	30	35	25	20	0	0	10	35	20	70	82	80	-	-	20	-	-	50	50	80	38
16:15	☁	-	20	20	30	0	20	80	15	5	5	5	120	30	25	70	-	10	15	25	25	40	15	20	40	25	15	0	0	10	35	20	60	82	70	-	-	30	-	-	30	40	80	37
16:45	29.9	-	20	20	30	0	20	-	15	5	5	5	110	30	25	65	-	10	10	20	15	40	15	20	40	20	15	0	0	10	30	10	55	82	70	-	-	15	-	-	30	45	65	33
17:15	☁	-	20	20	30	0	20	-	15	5	5	5	130	-	15	60	-	5	10	-	15	40	10	25	40	15	10	0	0	10	20	10	70	55	70	-	-	15	-	-	50	45	55	33
17:45	29.7	15	10	20	30	0	15	-	15	5	-	5	130	-	35	82	-	15	20	-	25	40	15	25	40	30	20	0	0	10	30	25	80	100	80	-	-	15	-	-	55	60	90	41
18:15	☁	15	10	15	30	0	15	-	15	5	-	5	130	-	35	82	-	15	20	-	20	40	15	25	40	30	20	0	0	10	35	20	80	90	70	-	-	30	-	-	45	60	75	39
18:45	25.5	-	30	15	30	0	30	-	15	5	-	5	130	-	35	85	-	15	20	-	20	40	15	25	40	30	15	0	0	10	35	25	90	50	70	-	-	25	-	-	45	50	65	39
19:15	☁	-	30	15	30	0	30	-	15	5	-	5	110	-	35	70	-	10	15	-	20	35	10	20	40	30	15	0	0	10	25	10	80	70	50	-	-	15	-	-	20	30	55	33
19:45	25.0	-	30	-	30	0	30	60	5	5	-	5	110	-	35	45	-	10	5	-	20	30	10	5	40	15	10	0	0	10	10	10	80	50	45	-	-	15	-	-	30	30	55	30
20:15	🌙	-	30	-	30	0	30	70	5	5	-	5	90	-	20	65	-	5	5	-	10	35	10	20	20	5	5	0	0	10	5	15	60	70	50	-	-	15	-	-	35	40	55	30
20:45	24.4	-	30	-	30	0	30	70	5	5	-	5	80	-	20	45	-	10	15	-	10	35	10	15	30	20	10	0	0	10	20	15	60	50	60	-	-	15	-	-	35	40	55	30
21:15	🌙	-	30	10	30	-	30	60	5	-	-	-	-	-	20	45	-	10	10	-	10	35	10	15	330	10	10	0	0	10	20	25	55	60	50	-	-	15	-	-	35	40	50	38
21:45	24.7	-	30	5	30	-	30	50	5	-	-	-	-	-	5	40	-	10	10	-	5	20	10	10	15	10	10	0	0	10	10	15	40	30	30	-	-	10	-	-	35	30	-	19
平均		6	16	18	24	0	13	66	10	5	5	5	105	12	28	69	-	11	12	18	17	36	15	19	44	18	15	0	0	7	25	20	65	68	71	-	-	22	-	-	37	41	70	

Figure 3.2: The structure of the waiting times data.

attraction is stored. This information could be used to predict the future waiting times for a specific attraction on a specific day. This is done by taking the average waiting times from the previous years. It is possible that on a specific year and day one of the attractions was closed. However, since it was taken for granted that all attractions are opened on the date specified by the user, the average waiting times are simply calculated for the rest of the years. The structure of the data can be seen in Figure 3.2. The representation is very illegible and hard to read, which makes the OptiRoute application a faster and better solution for calculating the waiting times.

Transportation Times

The transportation time from one attraction to the other depends on two things – the average walking speed of the visitor and the distance between the attractions. The average walking speed of the user is as already settled in the assumptions - 3km/h. The distance between the attractions can be obtained from the data that was calculated and stored as described in the section about the pairwise distances. Using this information

the transportation time is calculated in minutes. The time is always rounded into the positive direction, to be sure that the user will get to the desired attraction on time.

Duration of the Attractions
Each attraction at Tokyo Disneyland has a specific duration time. The information about that can be obtained directly from the official site of Tokyo Disneyland [off]. However, there are several attractions that do not have a duration time, due to the fact that the user can spend there as much time as he/she wants. For such attractions, an average duration time of 10 minutes was taken as already described in the preliminary section. This is because it is assumed that the visitor would like to visit as many attractions as possible and not waste much time in only a single attraction. So since most of the attractions have a duration of 5 to 15 minutes, 10 minutes is a good choice.

These three things determine how long it would take the user to visit the selected attractions and are the values needed for Equation 2.1. To represent the walking paths of Tokyo Disneyland as a graph, the attractions will be represented as nodes and the routes will be represented as edges. The weight of each edge would be the summation of the three described values. This problem seems similar to the known TSP as already described in the 2 chapter, but it is more complicated. The duration of the attractions and the transportation time are constant values, which regardless of the order of attractions, never change. The waiting times, however, vary not only on different days but every thirty minutes. This means, the weight of a single edge changes every time the order is changed, even when the order of only two attractions is being swapped. The TSP has a constant weight of the edges. Since the TSP is an NP-hard problem, this problem is also an NP-hard problem. To find the most optimal route, all different permutations of the chosen attractions need to be traversed. This is the brute-force search approach, where every possible solution is tested. For a small number of attractions the algorithm will still run fast, but with a growing number, the time needed for calculation grows factorially. This means that for a bigger number of attractions the brute-force search may last up to days, months, years, etc. Therefore a trade-off between the computation time and the accuracy of the final result should be one of the aims of this optimization algorithm. To achieve that, a couple of constraints need to be added. Their purpose is to exclude some of the possible permutations and thus reduce the computation time. This algorithm will be described in the following sections.

3.4.2 Constraints Regarding the Reachable Attractions

As was mentioned in Chapter 2, Ohwada et al. [OOK13] constraint their optimization algorithm using a branch and bound method. They limit the possible paths, by restricting which pairwise attractions are reachable from the others and which are not. In their paper, however, there is not a detailed description of which attractions they assume as not reachable. Consequently, it was decided that it is most logically to set those paths as not reachable that are longer than a certain distance. This way, the optimization algorithm will have to calculate fewer permutations and work faster. It was assumed

that this would, nevertheless, still lead to a satisfying result, since the visitor would not have to walk for long between attractions and save time from that.

After a couple of tests with different attractions and different constraining distances, this method proved to be useless. Approximately half of the combinations of different attractions turned to have no result or to need too much time for computation. The reason lies in the fact that, if among the chosen attractions there is a single attraction that is too far away from all the others, it would never be reachable and the algorithm will not return a result. On the other hand, if most of the chosen attractions are too close to each other, the provided constraint would not be useful in this case, and the algorithm will still traverse all permutations.

Based on that findings a new approach has to be developed, that overwhelms the weaknesses and drawbacks of the previously described one. Instead of using a constant value for the distance, it was decided that the best procedure is the algorithm to work dynamically. This means that all calculations will be done in real-time and will depend on the input data. This way, calculating which attractions are reachable and which not will depend on the user's choice of attractions. However, before describing the approach in details, two essential prerequisites need to be set. First of all, the optimization algorithm without these constraints had to be tested, to figure out what is the maximum number of attractions that are still computed in a reasonable time. A reasonable time is assumed to be less than 10 seconds since the purpose of this technique is that it will be computed in real time, so the user should not have to wait long for computation. The results showed that the maximum possible attractions within this time are 9, assuming that for all the rest algorithms and calculations a time of 3-8 seconds also will be needed. Table 5.1 represents the computational time needed for the different number of attractions. A more detailed description of that follows in the 5 chapter. As next, it had to be determined what is the maximum number of attractions that the user can choose to visit for one day. It was discovered that with already 15 attractions on very busy days the total time for visiting all of may exceeds the opening times of the park. With the assumption that the new algorithm will reduce that time, 15 attractions are chosen as an upper bound. For an experiment about that refer to Table 5.3.

3.4.3 Dynamic Calculation of Pairwise Reachable Attractions

Knowing the number of attractions for which the optimization has to be executed, as next, it had to be determined which attractions should be set as not reachable. Due to the fact that for up to 9 attractions the brute-force search is used and all permutations are calculated, representing that as a graph, would mean that all attractions will be connected to all others without containing loops. This results in an undirected complete graph $G = (V, E)$, where V refers to the vertices (attractions) of the graph and E represents the set of edges (paths) between these vertices. In the following explanations to a vertex would be referred with both words - vertex and attraction, when talking about the graph and to an edge with the words edge and path. Since this is a complete graph, it will be denoted by K_n, where n stays for the number of vertices. Such a graph has $n \cdot (n-1)$

directed edges and $\frac{n \cdot (n-1)}{2}$ undirected edges. For $n = 9$ this would mean $9 \cdot 8 = 72$ directed edges or 36 undirected edges in total. To achieve the same computational time for a greater number of attractions, some of the edges need to be removed and thus reduce the number of permutations. In these 36 edges, the two edges connecting the entrance with the starting and ending attraction are not included. Each attraction is reachable from the entrance. The entrance is also reachable from each attraction since it serves as a starting and ending point. Table 3.2 gives an overview of the edges that need to be removed according to the number of attractions.

Attractions	Directed Edges	Undirected Edges	Edges to Remove
n = 9	$9 \cdot 8 = 72$	36	0
n = 10	$10 \cdot 9 = 90$	45	45 - 36 = **9**
n = 11	$11 \cdot 10 = 110$	55	55 - 36 = **19**
n = 12	$12 \cdot 11 = 132$	66	66 - 36 = **30**
n = 13	$13 \cdot 12 = 156$	78	78 - 36 = **42**
n = 14	$14 \cdot 13 = 182$	91	91 - **38** = **53**
n = 15	$15 \cdot 14 = 210$	105	105 - **40** = **65**

Table 3.2: Edges to be removed according to the number of attractions. The numbers in bold indicate the number of undirected edges that have to be removed. The numbers marked in red are special cases for $|V| = 14$ and $|V| = 15$ attractions where adding 2 more edges for $|V| = 14$ and 4 for $|V| = 15$ does not make a significant change in the computation times.

The algorithm for removing edges works the following way – all the edges should be sorted in descending order according to their distance. The algorithm first removes the edges (paths) with the longest distance, until all necessary edges are removed, so that in the end there are only 36 undirected edges left. However, this approach could lead to several problems, such as removing all the paths that connect a particular attraction. Moreover, the fact that each attraction should be visited only once makes the algorithm even more complicated. Therefore, there is another important condition, which should resemble these problems. It is that for $|V| = 10, 11, 12, 13$ attractions (without the entrance) each vertex should have a $deg(v_i) = 2$, which means it should contain at least 2 edges incident to it. This condition makes sure that if any attractions are isolated and are too far away from all the others, they still would be reachable by being able to get in and get out of them [con]. In addition, the fact that the entrance has a path to every attraction also helps in some cases. As an illustration, there is the case when the graph contains more than one cycle consisting of only $|V| = 3$ vertices. Two of the vertices have a $deg(v) = 2$ and are connected to each other and to the third one. Such a case can be seen on Figure 3.3.

The worst case that can occur with 13 attractions is represented in Figure 3.4. There are $|V| = 5$ vertices that have a $deg(v) = 2$ and are all connected to the same 2 other vertices. The previously described conditions cannot solve this graph since there is no

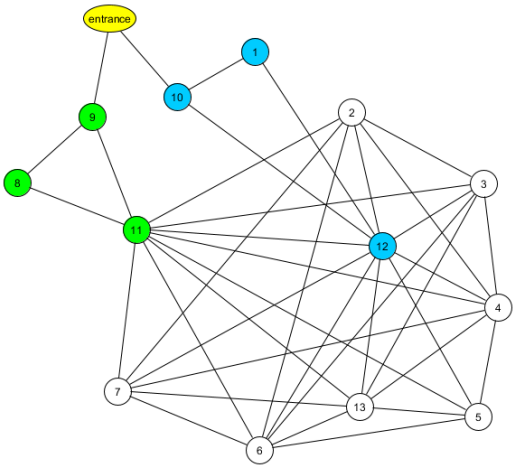


Figure 3.3: A graph with two cycles consisting of 3 nodes. In each cycle two of the nodes have only two edges and are connected to each other and to the third node. Using the entrance it is possible to traverse all nodes visiting each one of them only once.

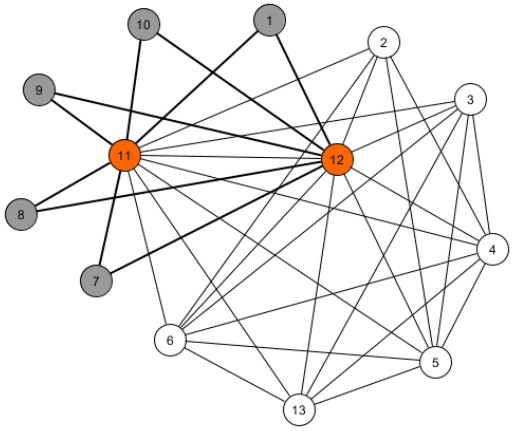


Figure 3.4: A graph containing the worst case for $|V| = 13$ attractions. No path exists that can traverse all nodes without visiting at least one of the nodes twice.

possibility to traverse all the attractions without visiting some of them twice. However, practically this case should never happen with the graph structure of Tokyo Disneyland. The attractions in Tokyo Disneyland are separated in several different groups according to their main topic and characteristics as it can be seen in Figure 3.5. The attractions in a single group lie dense to each other, while the different groups have a longer distance in-between. The suggested worst case contains five attractions (with numbers - 7, 8, 9, 10, 1) that only have two edges and all of them are connected to the same two other attractions (11 and 12). With a total number of $|V| = 13$ attractions even if seven attractions of the same group are taken and the rest 5 are from different groups, when removing edges it would be implausible to have all of them connected with the same two other attractions. Testing with 50,000 randomly chosen sets of 10 to 13 attractions was carried out. The results of the tests showed that for all of the sets the algorithm found a path, which means it did not crash. This algorithm for removing the edges is called the *Edge Removal Algorithm* and the pseudocode for it can be seen on Algorithm 3.1. Nevertheless, there is another algorithm called the *Edge Replacement Algorithm* that is responsible for resolving the problems caused by some of the existing cycles. If there is even a single case that can lead to an unsolvable graph, it will adjust the graph so that it is solvable. The following paragraphs should describe that additional algorithm in details.

As Table 3.2 represents, with a growing number of attractions, the number of edges

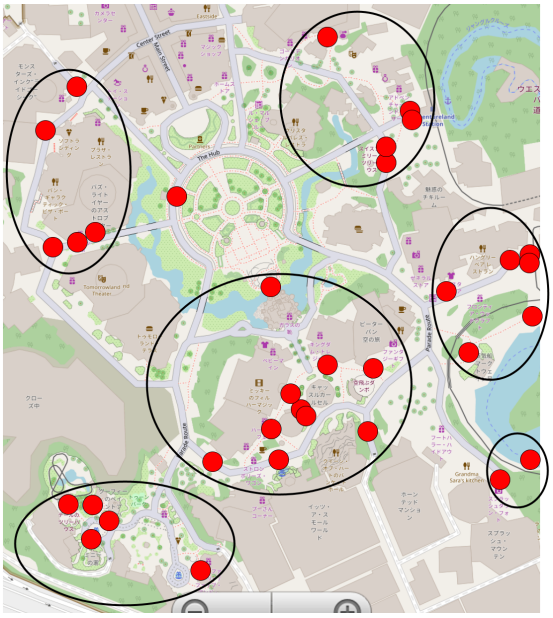


Figure 3.5: The red circles indicate positions on the paths that are closest to the respective attraction. The black circles show how are these attractions grouped according to their characteristics and types, but also the distance between them.

that need to be removed also grows and thus there are less possible permutations to be traversed in total. For $|V| = 14$ and $|V| = 15$ attractions (without entrance) the condition of having at least a $deg(v_i) = 2$ is no longer eligible. This would restrict the permutations too much, because in the worst case there could be more than five attractions with only two edges. Furthermore, having only two edges for so many nodes could lead to a lot of unsolvable graphs. For this reason, the condition for $|V| = 14$ and $|V| = 15$ attractions is adjusted and each vertex should have at least a $deg(v_i) = 3$. However, since almost two-thirds of the available edges have to be removed, the possible permutations decrease drastically, and the algorithm works even faster than with fewer attractions. To increase accuracy and to sustain similar to the previous times needed for computation it was tested that for $|V| = 14$ attractions except the $|E| = 36$ edges, two additional edges could be left without removing them. This means, in the end $|E| = 38$ edges can be used for the permutations of $|V| = 14$ attractions. The path is still calculated in less than 10 seconds, but the chance of getting a better result in the end is increased. The same applies for $|V| = 15$ attractions, however, in this case even $|E| = 40$ edges can be left and the time is still within 10 seconds. These two special cases are marked with red numbers in the table. However, these two special cases are only tested and work with Tokyo Disneyland. Therefore, for other amusement parks, there is no certainty that the same number of edges would lead to the expected results. This depends mainly on the positions of the attractions and the structure of the graph, so for another park, these values have to be tested and if needed adjusted.

Algorithm 3.1: Edge Removal Algorithm

Input : An undirected complete Graph $G = (V, E)$, where V is the set of vertices(attractions) and E is the set of edges(paths) that connect the vertices. Furthermore, l is a list containing all edges from E in descending order.

Output : A modified undirected Graph $G = (V_1, E_1)$, where $V_1 = V$ and $|E_1| = finalEdges$ and the list l with the remaining edges

```
1  minDegree ← 2;
2  finalEdges ← 36;
3  if |V| = 14 then
4  |  minDegree ← 3;
5  |  finalEdges ← 38;
6  end
7  if |V| = 15 then
8  |  minDegree ← 3;
9  |  finalEdges ← 40;
10 end
11 while l.size() > finalEdges do
12 |  for all edges v_i → v_j, i ≠ j in l do
13 |  |  if deg(v_i) > minDegree and deg(v_j) > minDegree then
14 |  |  |  l.remove({v_i, v_j});
15 |  |  |  l.remove({v_j, v_i});
16 |  |  else
17 |  |  |  Continue with the next edge in l;
18 |  |  end
19 |  end
20 end
```

This *Edge Removal Algorithm* is used to restrict the total possible permutations for a higher number of attractions (more than 9 and less than 16). It defines which two attractions are reachable and which not. This algorithm is used as a branch and bound in the NT algorithm, described by Ohwada et al. [OOK13] and that can be seen in Figure 2.1. The NT algorithm together with the *Edge Removal Algorithm* was tested with around 10,000 randomly chosen combinations of 14 and 15 attractions, which took a couple of hours to compute. About 90% of the tested combinations returned a result. The rest 10% had no solution. The reason for that lies in the fact that even with these conditions for the branch and bound, some special cases need to be taken into consideration and be further resolved. Therefore, there is another algorithm called *Edge Replacement Algorithm* as already mentioned, which is responsible for that. It checks first if the graph is a single connected component and if a path to each vertex exists. In case the graph is separated into two or more components, these components should be connected to each other by adding a further edge so that in the end there is a single

connected component. However, the testing showed that this case could happen very rarely and there is another reason for not finding a solution.

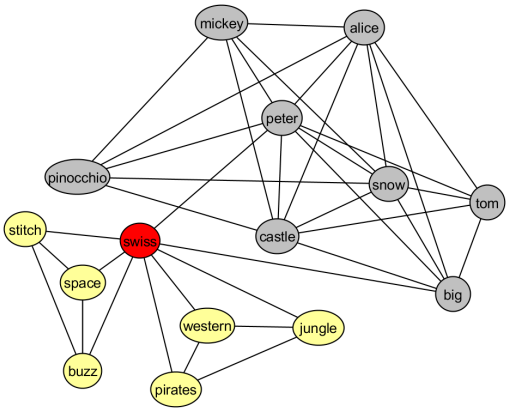
In some of the cases, there were cycles in the graph from which it was not possible to get out without visiting at least one of the nodes twice. One such case can be seen on Figure 3.6a. There are two of the cycles, whose vertices are marked in yellow. Each one of these vertices has the minimum number of edges - $deg(v) = 3$. To get in or out of both of the cycles and visit the other vertices the attraction "Swiss", which is marked in red, must be visited. Nonetheless, this is not possible without traversing "Swiss" twice. This would break the rule that was set in the beginning - not visiting the same attraction twice.

The idea to solve this problem is to replace some of the edges with others. What can be established from the graph is that the vertices "Swiss" and "Peter" contain the most edges. Both of them have a $deg(v) = 8$. However, "Swiss" is adjacent to more vertices with $deg(v) = 3$ edges than "Peter", where exactly the problem lies. Other problematic cases that had no solution were examined too, and it was figured out that in all of them the attraction with the highest degree is connected to the problematic cycles. So the algorithm chooses the attraction with the most paths that are connected to most other attractions with only three edges. So the edge connecting that attraction to the attractions with three paths should be removed, and another one should be added. It has been concluded, however, that if only one edge is replaced, some of the graphs would still have no solution. This is why it was found out that replacing two of the edges is enough to find a solution. Hence, from the attractions adjacent to the attraction with the highest degree, the algorithm searches for two attractions having paths with a maximum distance. Each one of the two attractions should be "disconnected" from the attraction with the highest degree and find two other attractions to connect to. The other two attractions, however, must not be connected to the first two and also not be part of the two cycles. Moreover, both of them should be different from each other and have a shorter distance compared to the other edges in the graph. Figure 3.6b represents a possible replacement of two of the edges connected to "Swiss". The blue ones are the old edges and the green ones are the new edges. This replacement of the edges transforms the graph in a way that all attractions can be visited only once in a single route returning to the original point: the entrance. One of the possible routes of the new graph can be seen in Figure 3.6c.

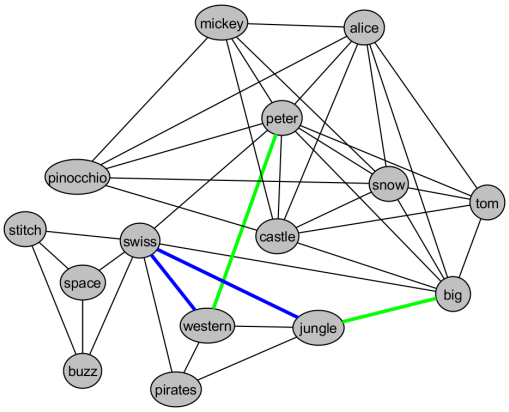
The cycles in the graph should not be removed since they are crucial for the permutations and finding an optimal route. This is why the algorithm should replace some of the edges in such a way that the problematic cycles no longer exist. The presented *Edge Replacement Algorithm* is called in a loop, so that it would be executed as many times as needed to get a solution in case after the first execution it is still not possible to find a route. The algorithm proved to be very efficient and always finds a route despite the problematic cases. It usually uses a single loop to find a solution and in extremely rare cases: two loops.

The NT algorithm together with the *Edge Removal Algorithm* and the *Edge Replacement Algorithm* is called the *Route Optimization Algorithm*. The *Route Optimization Algorithm*

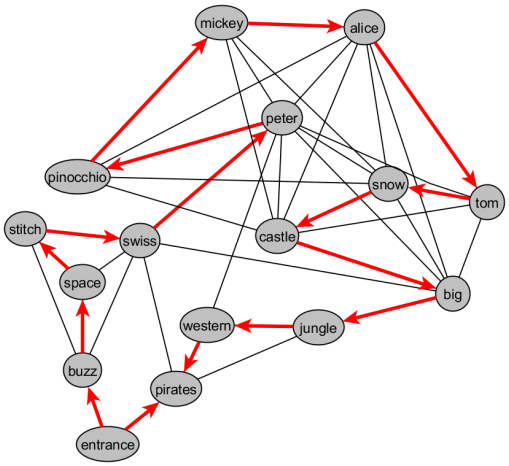
was tested with more than 150,000 different combinations of attractions with sets of 15 attractions. That test aimed to verify that no matter the set of attractions, the algorithm will always find a solution. The testing took around a day and a half to complete and the algorithm found a solution for all the tested combinations. Around 600 times in total out of 10,000 the *Edge Replacement Algorithm* was used to replace the edges and resolve the cycle problems in the graphs. The pseudocode of the *Edge Replacement Algorithm* can be seen on 3.2.



(a) Two cycles consisting of 3 nodes and all connected to the same node.



(b) Replacing two of the problematic edges with new edges.



(c) A possible path after replacing the edges.

Figure 3.6: Rearranging edges to resolve the cycle and make it possible to traverse all attractions by visiting each one of them only once.

Algorithm 3.2: Edge Replacement Algorithm

Input : An undirected Graph $G = (V, E)$, where V is the set of vertices(attractions) and E is the set of edges(paths) that connect the vertices. l is a list, containing all edges from E

Output : A modified undirected Graph $G = (V_1, E_1)$, where $V_1 = V$ and $E_1 = newEdges$

```
changeTwoEdges ← 2;
while checkConnectivity() of G with BFS is false do
    l.remove({v_i, v_j} ∈ E with the longest distance);
    l.add({v_i, v_j} ∉ E that connects vertices from two of the components and has
     shortest distance);
end
v_maxDeg ← a vertex with max degree that is adjacent to the most vertices that
 have degree 3;
l_deg3 ← stores all vertices with degree 3 that are adjacent to the vertex with max
 degree;
while changeTwoEdges > 0 do
    for v_i from l_deg3 do
        if edge {v_maxDeg, v_i} is longest then
            v_toChange ← v_i ;
        else
            continue with next v_i;
        end
    end
    for every vertex v_j not adjacent to v_toChange do
        if v_j ≠ v_maxDeg and v_j ≠ v_i from l_deg3 then
            if deg(v_j) is smallest and v_j ≠ to previous v_new then
                v_new ← v_j
            end
        else
            continue with next v_j;
        end
    end
    l.remove({v_maxDeg, v_toChange} ∈ E);
    l.add({v_toChange, v_new} ∉ E);
    changeTwoEdges − −;
end
```

3.5 Visualization and Route Intersection Minimization

This section describes the second contribution of this thesis. It involves the visualization of the results obtained from the calculation of the route optimization algorithm. The whole route should be displayed in a way that is most convenient for the visitors and to support their faster coordination. This includes offsetting paths and avoiding overlaps of the different paths for a better visual representation.

3.5.1 Paths Offset and Intersection Minimization

The main part of the visualization is the paths themselves. Initially, all paths were drawn on the map taking the original coordinates from the minimum pairwise paths as described in one of the previous sections. However, this leads to a very illegible map, because some parts of the path are traversed more than once for the different attractions. This is the case especially for a bigger number of attractions. Such overlaps can lead to confusions, since the user may not know how many and which paths exactly go through a particular part of the path. Figure 3.7 shows how does the paths' visualization look like without offsetting the overlapping parts and without minimizing intersections. To overcome this problem, the paths need to be offset on the places where more than one path passes through. Furthermore, to preserve the map clear and easily readable, the paths should be offset in such a way that the intersections and overlaps with the other paths are held as minimal as possible.

Paths Offset
Before describing the *Route Intersection Minimization Algorithm*, it is important to explain the algorithm responsible for the paths' offset. An offset or parallel line means a displacement of every point/coordinate of the original line by a specific amount. Each path consists of several different coordinates that are connected to each other and thus form multiple lines. This is called a polyline. Each coordinate of the polyline consists of the values latitude and longitude, determining the position of the coordinate on the Earth. What has to be achieved, is to be able to offset these polylines with a certain distance to the left or the right side of their original position. The well-known vectors from the Mathematics and the different operations with them can be helpful for accomplishing this task. Every two consecutive coordinates from the polyline from a straight line and can be represented as a vector. The direction in which these coordinates are visited will determine the direction of the vector. After building a vector from the two coordinates, the normal vector could be easily calculated. The normal vector is the vector perpendicular to the original vector, and if its sign is changed, the normal vector will point in the opposite direction. However, after calculating the normal vector, it has a certain length. Since all computations are done with real coordinates, the length of the vector should be adjusted so that it fits the scale of the map. To test different options and find the most appropriate length, the vector should have in the beginning information only about its direction. To achieve that, the vector should be normalized. Such a vector has a length of 1.

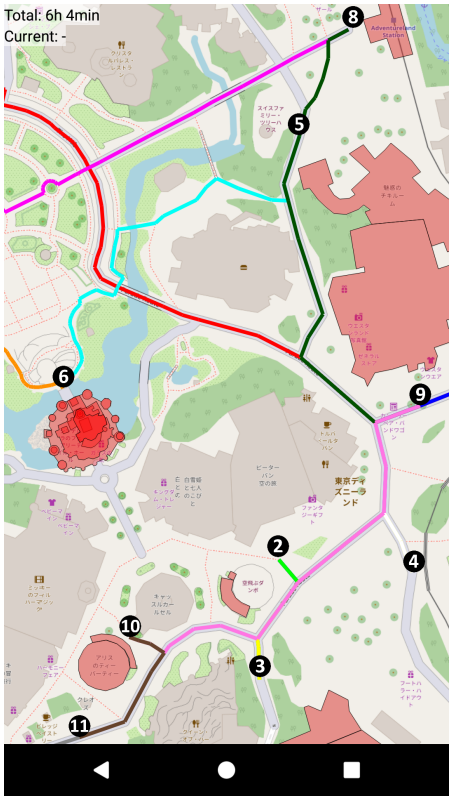


Figure 3.7: Visualization of paths without offset and intersection minimization.

As next it had to be determined which value to choose so that when multiplied with the normalized normal vector, the offset would be small, but still visible. Working with coordinates, however, makes things more difficult, because instead of having integers, in this case, the rational numbers come in hand. Not all rational numbers can be accurately represented with programming languages, which is why some numerical mistakes may occur when doing calculations with them. Due to the level of complexity when working with coordinates to find an appropriate value, various options had to be tested. The testing showed that to multiply the perpendicular vector with the value 0.00001 and to add the result to the coordinates of the original vector brings the best results. An example offset can be seen in Figure 3.8, where the red path is the initially drawn and the dark green path is the offset path.

As already mentioned the polyline consists of several coordinates and each coordinate is positioned on a corner. This means - on places where the polyline changes its direction. The algorithm goes through all the coordinates of the polyline and takes always a pair of consecutive coordinates. It calculates the displacement (offset) for that pair according to the normal vector. In most cases, both start and end coordinate from each line need to be offset. The end coordinate is a start coordinate of the next line, which means that each coordinate will be offset twice, but in different directions depending on the

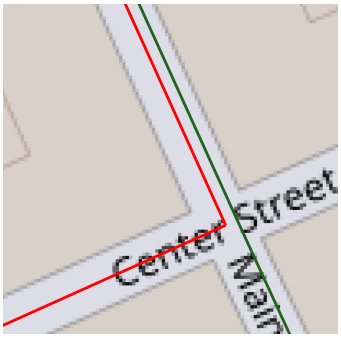
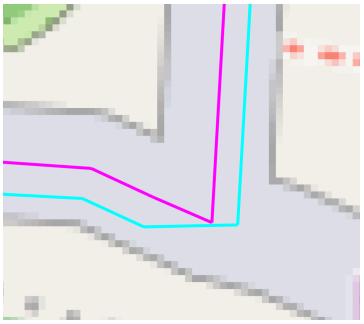
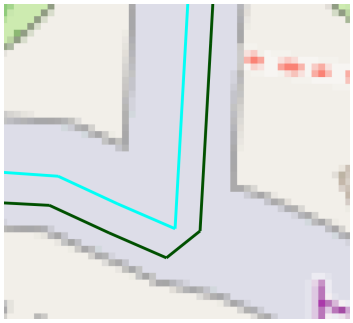


Figure 3.8: Example showing the offset of paths that go through the same part of the road. The red path is the original one, and the dark green path is the offset path.

normal vector. However, sometimes only the start coordinate of the line is being offset. Which coordinates should be offset depends on the angle that is formed up between two consecutive lines. This angle determines the magnitude of change in the particular direction. Both start and end coordinates from a line need to be offset if the inner angle between the current and the consecutive line is smaller than a particular value. The reason is that, depending on the angle, each of the two lines will have a normal vector in different directions and to have the corner coordinate displaced correctly, it should be offset twice in the directions of both normal vectors. The chosen value is set to be 170°, because two consecutive lines with angles between 171°and 180°look almost as a straight line.



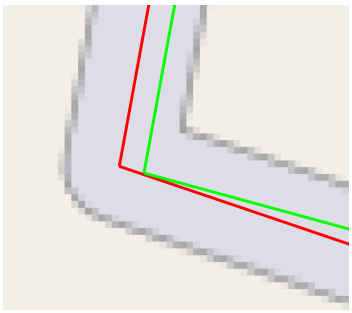
(a) Wrong offset - offsetting and drawing only the start coordinate of each line. Magenta - original polyline, cyan - offset polyline.



(b) Correct offset - offsetting and drawing both start and end coordinates of each line. Cyan - original polyline, dark green - offset polyline.

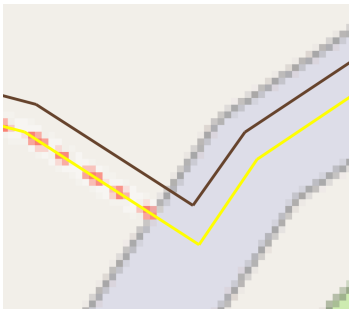
Figure 3.9: Offseting coordinates at the outer side of the angle between the original lines.

Furthermore, two cases need to be examined depending on in which direction the line is offset. The first case happens when the new lines must be offset to the outer side of the original lines - to the side with the outer angle. A simple way to create an accurate



(a) Wrong offset - offsetting and drawing only the start coordinate of each line. Red - original polyline, green - offset polyline.

(b) Wrong offset - offsetting and drawing both start and end coordinates of each line. Yellow - original polyline, orange - offset polyline.



(c) Correct offset - offsetting both start and end coordinates and drawing only their intersection point. Yellow - original polyline, brown - offset polyline.

Figure 3.10: Offseting coordinates at the inner side of the angle between the original lines.

representation is to just offset the end coordinate of the first line and the start coordinate of the second line. This way, in the result there would be a further line looking like as if the corner is offset and cut so that two distant corners show up. If only the start coordinate is offset, the line would look incorrect. Figure 3.9 represents the case for outer offsets. The second case is a bit more complicated since it requires further calculations. This is the case when the angle between two lines is less than 170°and the new polyline should be positioned in the inner side of that angle. If both end and start point of the first and second line are offset, the result would look like on Figure 3.10b. However, what can also be observed is that the new lines intersect each other at a specific point. So, in this case, this intersection point should be calculated, and only it will be drawn on the map. Figure 3.10 represents the offset at the inner part of the line and the different possibilities depending on which coordinates are being offset.

Intersection Minimization Algorithm

After defining the way in which the paths are being offset, the next algorithm, which is crucial for representing the paths visually, should be described. This is again an optimization algorithm. It aims to reduce the overlaps and intersections between the different paths. The following algorithm is a greedy algorithm. Its purpose is to try to find a globally optimal solution by finding an optimal solution only locally at each step.

Each path between two attractions has a different color so that it is easier to identify it. Initially, each path between two attractions is drawn at the original coordinates - in the middle of the road - in case there is no other path already going through that path fully or partially. If there is already another path, the coordinates that would overlap with

it should be offset. For this part of the polyline, the algorithm calculates two possible offsets – one on the left side and one on the right side. It checks then which side returned fewer intersections of the current polyline with the previously drawn one. It draws it with offset at the side with fewer intersections. Already drawn paths, however, should not be changed after drawn once, no matter if there is another more optimal global solution when changing their offset. Only the paths that are about to be drawn should be offset so that they minimize the overlaps. This is why the algorithm is greedy.

The algorithm stores for each line (edge) consisting of two consecutive coordinates how many paths are already going through it and how are they positioned. It does not matter how many paths go through a particular part of the map. If a new path should also go through that part, the algorithm always first checks the two possible offsets - on the left and the right side of all already existing paths at the current position. It always chooses to offset the new polyline to the side with fewer intersections. The value for the offset depends on how many paths are already offset in the specific direction at the current path segment. A path segment is considered to be a single line from the polyline. The first polyline that is positioned in the middle is without offset. If the next one is offset to the right the value for the offset is 0.00001, and if the following polyline should also be offset to the right at this part of the path the value would be double – 0.00002. So the value always grows according to the number of polylines offset in the particular direction. The algorithm responsible for offsetting paths and minimizing the intersections is called the *Intersection Minimization Algorithm* (Algorithm 3.3).

However, a great deal of attention must be paid when dealing with the vectors' direction. The direction in which the path is visited is important and plays a significant role when storing the order in which the paths are offset. The map of the application is never rotated, and thus its position stays the same. When looking at it, the entrance is always located above all other attractions. Bearing that in mind, the right side of the map will always correspond to the right side of the visitor if he/she goes from the bottom to the top. When the user goes in the opposite direction, however, the right side of the map would mean the left side of the visitor. This is very important for the calculations. The offset for each line segment is stored twice in both directions but with swapped values. For example, if a line traversed from bottom to top has two lines offset to the left and 1 to the right, the offsets will be stored once like this and once in the opposite direction – from top to bottom with 1 line on the left and two lines on the right. Figure 3.11 gives different examples of visualization using the *Intersection Minimization Algorithm*.

3.5.2 Visual Components Assisting the Users

Despite the visualization of paths, there are some further components that assist the user in finding attractions easier and faster. As first, to emphasize which attractions have been chosen by the user and to be able to distinguish them better from the others, the buildings of all chosen attractions are being colored differently. Furthermore, next to each building of an attraction a circle with a number is displayed, showing the order in which each attraction has to be visited. On the left top corner of the screen of the map,

(a) (b) (c) (d)

Figure 3.11: Different visualizations of paths between attractions using the algorithms for paths' offset and reduction of overlaps.

Algorithm 3.3: Intersection Minimization Algorithm

Input : A polyline p defined by a list of vertices l_{poly} to be offset to the left or right side. The number of lines l_{left}, offset on the left side and l_{right}, offset on the right side of the current edge $\{v_i, v_{i+1}\}$.

Output : The needed offset in the specific direction

```
1  currentOffset ← 0.00001;
2  offsetDirection ← null;
3  leftIntersections ← checkLeftIntersections() - calculates all intersections of p
     on the left side ;
4  rightIntersections ← checkRightIntersections() - calculates all intersections of p
     on the right side ;
5  if leftIntersection <= rightIntersections then
6  |  offsetDirection ← left;
7  else
8  |  offsetDirection ← right;
9  end
10 for All {v_i, v_{i+1}} in l_poly do
11 |  if offsetDirection == left then
12 |  |  currentOffset ← currentOffset · l_left;
13 |  else if offsetDirection == right then
14 |  |  currentOffset ← currentOffset · l_right;
15 |  end
16 |  Offset {v_i, v_{i+1}} with the currentOffset in the offsetDirection;
17 end
```

there is a representation of the time needed to visit all chosen attractions marked with "Total: ". Underneath there is also a field with the text "Current: ". Its purpose will be explained in the next section since it is part of the user interaction.

3.6 User Interface and Interaction

The interaction with the application is an essential part and will be described in details in this section. As it can be seen in Figure 3.1, representing the workflow of OptiRoute, the calculations require some input information. When starting the application, the user first needs to pick a date on which he/she wants to visit Tokyo Disneyland and then click "Confirm". The next required information is the attractions that the user wants to visit. He/She can choose up to 15 attractions, and the selected attractions are highlighted with a black frame around the image. Furthermore, on the bottom of the screen, the user sees the number of chosen attractions. When he/she is ready with the selection, he/she can proceed to the next screen clicking on the button "Show Path". After clicking that the *Route Optimization Algorithm* starts its calculations and the results are used for the *Intersection Minimization Algorithm.* As next, the computations of both algorithms are

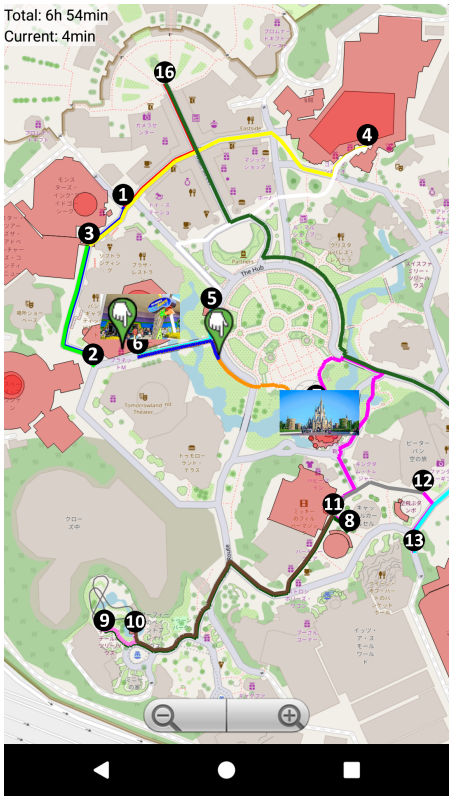


Figure 3.12: The animation between two attractions and the photos of the two attractions, as well as the landmarks.

represented visually on the map. The route always starts and ends at the entrance and all the paths have different colors.

The circles with numbers represent the order in which the attractions need to be visited. These circles are clickable. When the user clicks on one of these, the path from the previous to this attraction is showed with an animation. The animation is a marker moving on the road between the two attractions. Because the coordinates taken from the OSM are only positioned on corners, the distances between two consecutive coordinates vary a lot. Running a linear animation will lead to a very irregular movement of the marker so that it will move faster on lines with a longer distance than on short. This is because the animation tries to run the distance between attractions always for the same amount of time. To overwhelm that problem further coordinates had to be added between every two coordinates so that the distance between any two consecutive coordinates is exactly 1 meter. This was achieved by interpolating each line with the needed amount of steps so that each new line gets a distance of 1 meter. This resulted in animation with a constant and smooth movement of the path between any two attractions.

Moreover, photos above the building of both attractions pop out and if the user clicks

on the photo, it gets bigger, so that it is easier distinguishable. In addition, different landmarks like coffees, restaurants, shops that are close to the path between the two attractions, are marked (Figure 3.12). If the user clicks on them a name and a photo of the current landmark is being displayed. As already mentioned in the last subsection on the top left corner there is further information about the time needed to visit the whole route. Not only the total time is displayed, but also the current, marked with "Current". If the user clicks on a circle with a number, the minutes needed for the transportation from the last to the selected attraction are being displayed after "Current:". All these possible interactions with the application give the user more freedom and assist him/her in the coordination and easier finding of paths.

Here is a list of the possible interactions a user can do:

Interactions for selecting input parameters

- Define the date on which he/she wants to visit Tokyo Disneyland.
- Selected up to fifteen attractions to visit.

Interactions during the visualization

- Move the map and zoom in and out with the two buttons at the bottom corner of the screen.
- Click on the numbers in the black circles on the map and this way display the animated route, leading to the selected attractions, as well as images of the attractions and positions of landmarks near to the animated route.
- Click on the images of attractions to scale them up and see them more clear and click twice to scale them down.
- Click on the landmark positions to display an image and the name of the current landmark.

CHAPTER 4

Implementation

The current prototype of the framework is implemented on Android using Java. It is developed tested and executed on a PC, which features a Nvidia GTX 1050 TI GPU, an Intel Core i7-7700HQ CPU 2.80GHz, 16 GB RAM and a 256GB SSD. Android was chosen as a platform since the idea is to use the technique on the mobile device when the user is in Tokyo Disneyland. In order to implement the application, a lot of external libraries as well as various data are necessary.

The waiting times used for the optimization algorithm were stored in JSON files. In order to process, read and write such type of files and work with them in Java the external library JSON.simple is integrated into the project. Open Street Map (OSM) is used for the map representation, because in comparison to Google Maps, the different components of the map like buildings, green areas, roads, etc. are distinguished easier. The coordinates and map information necessary for the computations are obtained from the OSM database. The map data for the area of Tokyo Disneyland is stored in a file with the extension .osm. So that it is possible to read and use the information from a .osm-file, an additional library is used. The library has a simple, but sufficient for the purpose functionality and is called BasicOSMParser. It allows the extraction of the information from a .osm-file to several .csv-files containing different information about the coordinates on the map.

For the visual representation, two main libraries come in help. The Osmdroid library, which is an android library that provides different tools and views that make the interaction with OpenStreetMap-Data possible. All visualizations of paths and the outline of the buildings, as well as marking points on the map, are possible with this library. For some of the more complex representations, the additional library Osmbonuspack is used.

To deal with the computations regarding the coordinates, two external libraries are also integrated into the project. The first one is geodesy. It is used to calculate the real distance between two coordinates on Earth since it is crucial to have an accurate pairwise

distances computation between the attractions. For the interpolation of lines, used for the animation of paths, the library GeographicLib is imported and used in the project. It can find each coordinate lying between two coordinates only by specifying the distance from the coordinates. The distance is represented as a fraction, which means a portion of the whole line. For example, $\frac{1}{2}$ or $\frac{1}{3}$ of the line.

4.1 Installation and Running of OptiRoute

The OptiRoute application uses the open-source build automation system Gradle, which automatically creates all the dependencies with external libraries and tools. All the listed libraries in the previous section should be imported in the "build.gradle". All external files and data are incorporated into the project and are attached to the application. In case they have to be changed or extended, they are positioned in the "raw" folder which can be found in the resources of the android project. These are the two main aspects that are important to be able to install the application. To run it, simply the "app"-folder must be executed and the desired mobile device selected, which can be an emulator or a real smartphone. The source code of the project can be found on GitHub [Vas].

CHAPTER 5

Results and Discussion

This chapter presents the results of this thesis and demonstrates if they cover the expectations and goals set at the beginning of the project. The chapter is split into three parts. While the focus of the first part lies on evaluating the *Route Optimization Algorithm*, the second part examines the *Intersection Minimization Algorithm*. The third part is about the limitations of the developed technique, as well as a discussion on its performance.

5.1 Route Optimization Algorithm

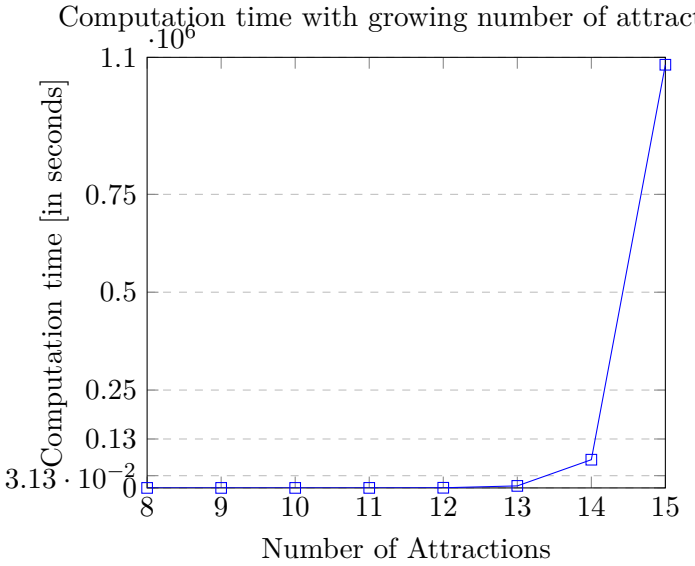
Having all set tasks finished, the application for Tokyo Disneyland is capable of optimizing the route of a set of selected attractions, as well as visualizing this route most conveniently. This section aims to examine the effectiveness of the *Route Optimization Algorithm* by testing its running time and the quality of the results it returns. Since the presented problem is an NP-hard problem, with a growing number of attractions, the time needed for computation grows rapidly. To be exact, the algorithm runs in a factorial time when solving the problem with a brute-force search. As was already mentioned, it was determined that for up to 9 attractions the algorithm can calculate the most optimal solution using a brute-force search. However, of fundamental interest is to test the results returned for 10 to 15 attractions, where several constraints were implemented to reduce the time for computation. Table 5.1 represents the calculation time of the NT - algorithm for a different number of attractions using the brute-force search. The time was determined by executing tests with the different numbers of attractions, but it was tested with maximal 14 attractions. Dependency and a relation between the number of attractions and the computation time were discovered during the tests. This relation is described by Equation 5.1.

$$T_n = n \cdot T_{n-1} \tag{5.1}$$

Number of Attractions	Computation time
8	≈ 0 sec.
9	≈ 0 sec.
10	≈ 3 sec.
11	≈ 33 sec.
12	≈ 6,6 min.
13	≈ 1,43 h.
14	≈ 20,02 h.
15	≈ 12,51 d.

Table 5.1: Computation time using a brute-force search to find the most optimal route order for a fixed number of attractions.

T_n represents the computation time for a set of n attractions and T_{n-1} is the computation time of n-1 attractions. Based on that equation it was concluded that for 15 attractions the computation time would take around 12,5 days to find the most optimal solution using a brute-force search. However, the goal of the application is to be used in real-time.



In order to analyze the effectiveness of the current approach, the result of the algorithm is compared to the result of a randomly chosen order of a set of attractions. To be more concrete a set of attractions are tested with the *Route Optimization Algorithm* and the resulted overall time for visiting these attractions is stored. For the same set of attractions, a hundred of possible permutations are taken randomly and the overall time for each permutation is stored. As next, the minimum, the average and the maximum time of these hundred results are computed.

The test is executed with the maximum possible attractions a user can choose – 15 since this is the most complicated case and most edges are being removed. The algorithm is tested for ten different sets, each containing 15 attractions, which are chosen randomly. Furthermore, the test is conducted for two different dates – a date on which the traffic of people is average and a date on which usually the park is very crowded. The idea is to examine the behavior of the algorithm under different circumstances. Table 5.2 and Table 5.3 represent the results of the tests. The column called "Route Optimization Algorithm" represents the time needed to visit all attractions by using the *Route Optimization Algorithm* already described in this thesis. The three other columns (named "Random Order") represent the results from the hundred different permutations of the same set of attractions by computing the overall time for each one of them. The first column stays for the minimum overall time from the sets, the middle shows the average times, and the third one represents the maximum overall time from the sets. The most crucial value of the three values is the average time. Comparing the overall time of the *Route Optimization Algorithm* and the average time from the random order of attractions, it can be observed that the algorithm has a significantly better time. In most cases, the user can save between 100 and 150 minutes if he/she uses this technique instead of choosing him-/herself how to visit the attractions.

Interestingly, the algorithm also performs better than the minimum time among those 100 tests. When the permutations of the same sets of attractions are increased to 1000 or 10 000, the average time stays almost the same with a small amplitude of change: +- 5 min. The minimum time, however, may get less with an increased number of tests, since the probability of getting the optimal solution among the tested permutations of attractions increases. Nevertheless, the average time is most important for evaluating the algorithm. Moreover, the *Route Optimization Algorithm* makes it possible to be able to visit 15 attractions. There are days with huge crowds so that the queues in front of attractions are just unimaginably long. On such days without a strict plan, it would be impossible to visit all 15 attractions. Table 5.3 can confirm this statement. The maximum time for visiting attractions is around 14 hours, which makes 840 minutes. The date 15.08.2018 is a date with many visitors in Tokyo Disneyland. For this reason, the overall times for all sets of attractions is much longer than the times on 30.04.2018. The table shows that there are two sets with a maximum overall time more than 840 minutes. This means the user will not be able to visit all 15 attractions on that date. However, having the same set of attractions, but calculating their order with the optimization algorithm, the times are reduced to under 840 min. So the algorithm also makes it possible to visit more attractions, especially on very busy days.

Despite evaluating the *Route Optimization Algorithm* with a randomly chosen different routes of a set of attractions, it is of great importance to compare it with the most optimal solution. However, as it can be seen in Table 5.1, the computation time for 15 attractions is expected to last around 12 days and a half for a single set. Therefore, to be able to make a conclusion about the effectiveness of the algorithm, it needs to be tested with at least five different sets, which would take in total more than two months to

Set Number	Route Optimization Algorithm	Random Order (min)	Random Order (average)	Random Order (max)
1	**383**	442	**513**	597
2	**347**	414	**472**	545
3	**368**	408	**491**	550
4	**428**	488	**537**	611
5	**424**	471	**554**	632
6	**376**	438	**502**	609
7	**363**	404	**472**	575
8	**453**	484	**537**	599
9	**499**	541	**602**	671
10	**381**	423	**481**	563

Table 5.2: Comparison of the overall time [in minutes] for visiting 10 different sets of attractions on 30.04.2018, calculated with the *Route Optimization Algorithm* and with 100 different random orders of each set.

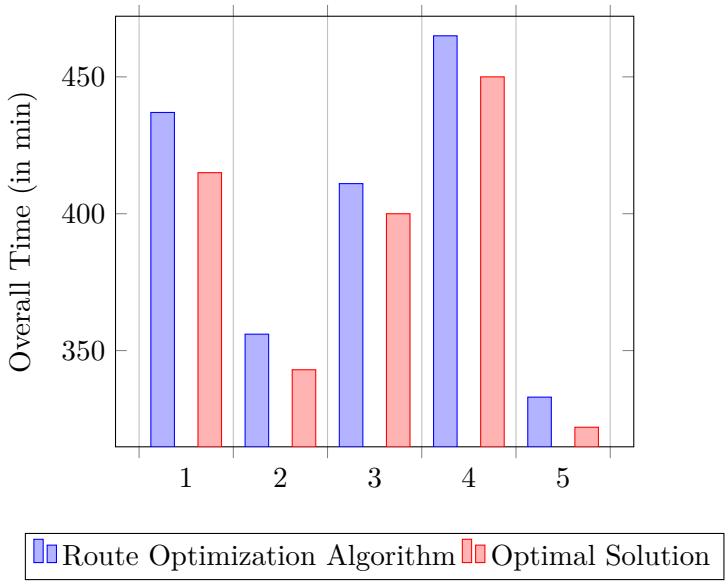
Set Number	Route Optimization Algorithm	Random order (min)	Random order (average)	Random order (max)
1	**580**	679	**751**	820
2	**530**	592	**673**	740
3	**565**	610	**689**	769
4	**608**	632	**729**	799
5	**652**	700	**795**	853
6	**568**	633	**712**	791
7	**503**	550	**623**	687
8	**656**	676	**750**	812
9	**744**	791	**863**	921
10	**535**	603	**674**	764

Table 5.3: Comparison of the overall time [in minutes] for visiting 10 different sets of attractions on 15.08.2018, calculated with the *Route Optimization Algorithm* and with 100 different random orders of each set.

compute. A more reasonable experiment could be conducted with sets of 14 attractions since the calculation time is much shorter, but the complexity is still high. The results of that experiment are displayed on Table 5.4. The difference between the result of the *Route Optimization Algorithm* and the optimal solution varies between 11 and 22 minutes for the tested five sets. Having these results, it can be concluded, that the *Route Optimization Algorithm* appears to be quite effective. It must be stated that 22 minutes is not much time, bearing in mind the fact that the algorithm runs for only a couple of seconds in comparison with 20 hours for the most optimal solution.

Set Number	Route Optimization Algorithm	Optimal Solution	Difference
1	437	415	22
2	356	343	13
3	411	400	11
4	465	450	15
5	333	322	11

Table 5.4: Comparison of the overall time [in minutes] for visiting 5 different sets of 14 attractions calculated with the *Route Optimization Algorithm* and the optimal solution.



The results from the experiments demonstrate that the *Route Optimization Algorithm* is rather effective in finding a solution pretty close to the most optimal one. Moreover, it performs much better than randomly selected attractions by being able to save the visitors up to 2 hours and a half. However, the findings for the comparison of the *Route Optimization Algorithm* and the most optimal solution are based on a limited number of tested sets of attractions. This is because the computation time for finding the most optimal solution takes too much time. The results from such analyses should, therefore, be treated with considerable caution.

5.2 Intersections Minimization Algorithm

Having evaluated the *Route Optimization Algorithm*, the purpose of this section is to evaluate the *Intersection Minimization Algorithm* of the different paths between attractions. On Figure 3.7 the paths are displayed without displacement and without minimizing the overlaps and intersections. The different examples on Figure 3.8 on the

other hand shows the paths with an offset and with reduced intersections. The aim here is to compare both types of visualization and determine if and to what extent the users find the offset visualization better and easier.

The evaluation is conducted in a form of an interview. Six different participants at different ages and with a different level of background knowledge in the topic are interviewed. They are first asked to choose from the list of attractions 15 of them that they would like to visit, as well as the day of the visit. After that, the *Route Optimization Algorithm* calculates the most optimal route of the set of attractions. The result is visualized first without using the offset and intersection minimization algorithm. Afterward, the same set of attractions is visualized using that algorithm. Both visualizations are showed to each participant, and a couple of questions are asked about them.

The questions are the following:

1. What would you say about both visualizations?
2. Which visualization do you prefer and find easier to follow and why?
3. Can you think of a more sophisticated and intuitive way to visualize the routes? If yes, what is it?
4. Do you think the animation, as well as the images of the attractions and landmarks, are helpful when clicking on a specific attraction number?
5. Would you use the OptiRoute Application if you are about to visit Tokyo Disneyland or you prefer to make the plan and route yourself?

The results of the interviews of all participants are quite similar for most of the questions. However, there are minor differences. The first question gave the participants the opportunity to express their opinion about both visualizations as a whole. They all found the second visualization more understandable. However, some of them said that both of them are good and helpful since the user can see the whole route in both visualizations. The second question suggested a more concrete answer expecting the participants to choose the visualization they would prefer to use. All of them were categorical that the visualization using the *Intersection Minimization Algorithm* is much better and easier to follow since all routes are visible and have fewer intersections and overlaps. Half of the participants could not suggest a more intuitive way to visualize the routes. The rest gave some ideas about displaying the route only between any two attractions when the user clicks on the desired attraction. Another suggestion was to add arrows to the routes so that even when zoomed in the user knows the walking direction. Furthermore, one of the participants suggested that it would be more intuitive if the map is being rotated so that the visitor can easily orientate.

All participants find the animation helpful since it highlights the current route. They also find the images of attractions very useful, since the user knows this way how the

attraction should look like in reality and find it easily. Also, the landmarks are regarded as useful by the participants, because they believe they will lead them the way and even give them suggestions for a restaurant or a shop they would like to visit after the tour. Finally, all participants were asked if they would use the application if they were about to visit Tokyo Disneyland. They all prefer using the OptiRoute application instead of creating the plan themselves. They believe this would save them much time and will guide them through the park, as well as it will give them a preliminary overview of the whole route and the total time needed for visiting.

5.3 Limitations and Performance

One of the main limitations of the current project is the fact that there is a limited amount of attractions that the user can choose. For example, on not so busy days the queues and waiting times are not so long so that it is possible to visit more than 15 attractions, which the application does not allow. Furthermore, the user is limited in his/her choice. This is because no time for resting is considered in the application. Also, the average walking speed of the visitors is set to be constant, which in reality varies depending on the visitor and his physical state.

The performance of the computation in the application depends mainly on the storage of the external data that is needed for the algorithms. Initially, for every single loop or calculation, the data was read from the external time. This cost, however, much time so that only up to 6 attractions could be calculated with all permutations in less than 10 seconds. Consequently, all the external data was stored in different data structures in the beginning. This increased the performance drastically and made it possible to traverse all permutations for up to 9 attractions.

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

In this bachelor thesis, different algorithms for solving the problem with the waiting times in amusement parks were introduced. In the first part of the thesis, several algorithms optimizing the route plan strategy were presented. Together they form the so-called *Route Optimization Algorithm*. The algorithm takes a set of selected attractions and the date of visit as input and transforms the selected attractions into a route with a specific order. The resulted route is calculated in real-time, taking not more than 5 seconds, no matter the number of attractions and optimizes the total time for visiting the selected attractions. Therefore, the project fulfilled all the goals set in the beginning, regarding the route optimization algorithm and even performs better than expected. The trade-off between the computational time and the resulted route is as good as possible regarding the goals set at the start of the project.

The second central part of the thesis is the visual representation of the routes. This visual representation has to happen in real-time and display the routes, depending on the user's choice and the results from the *Route Optimization Algorithm*. The routes should be displayed in a simple, but appropriate and easily readable way. Consequently, it was decided that the most convenient representation, in this case, would be similar to the metro-map layout style, which contains paths with a particular offset and with a reduced amount of intersections between them. The presented *Intersection Minimization Algorithm*, responsible for the offsets and the intersections minimization using a greedy algorithm, brought to good visual results, which also fulfilled the goals set in the second contribution of the thesis. The path animation and the attractions' images and the landmarks are just additional extensions that support and assist the users in finding their way easier and faster.

Both algorithms together are well compatible, by providing the most help and assistance to the visitors. Together they are offering the users not only a route plan strategy but

also a dynamical visual representation. This way, the future visitors can significantly reduce the time needed not only for pre-planning but also for the duration of the visit itself, by being able to enjoy such amusement parks and what they have to offer to the maximum.

6.2 Future Work

The current project can be improved and developed and serve as a basis for future works. Some of the improvements that could be integrated are to overwhelm the limitations and remove some of the preliminaries and assumptions set in the beginning. These assumptions make it clear what information and preconditions build the basis of the technique. They make things simpler, because otherwise, the complexity would go beyond the scope of this thesis, which is already complex enough.

However, removing them will increase the flexibility and the features of the application. For example, additional time could be considered for having rest or a lunch, such as setting a fixed time the visitor would like to rest. Furthermore, instead of having a constant walking speed for everyone, a varying walking speed can be implemented by selecting at the beginning of the application the type of visitor (a pregnant woman, a person walking very fast, a slow walking person, etc.). Despite these improvements, also a starting time for visiting the attractions can be considered so that the visitors can have more flexibility in choosing at what time of the day they would like to visit Tokyo Disneyland. Also, some of the suggestions from the participants in the qualitative evaluation can be used as an idea for further development of the techniques.

The basic approach used in OptiRoute can be integrated into many other similar issues, where the problem with the queues and waiting times occurs. Consequently, OptiRoute could be improved in many ways in future works and thus help users avoid waiting in long queues and save much time.

List of Figures

List of Tables

List of Algorithms

Bibliography

[BKPS08] Michael A. Bekos, Michael Kaufmann, Katerina Potika, and Antonios Symvonis. Line crossing minimization on metro maps. In Seok-Hee Hong, Takao Nishizeki, and Wu Quan, editors, *Graph Drawing*, pages 231–242, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[BNUW07] Marc Benkert, Martin Nöllenburg, Takeaki Uno, and Alexander Wolff. Minimizing intra-edge crossings in wiring diagrams and public transportation maps. In Michael Kaufmann and Dorothea Wagner, editors, *Graph Drawing*, pages 270–281, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Bru13] Corinne Brucato. The traveling salesman problem. June 2013.

[con] The traveling salesman problem with integer programming and gurobi. `http://examples.gurobi.com/traveling-salesman-problem/`.

[Dij59] E. W. Dijkstra. A note on two problems in connecion with graphs. *Numer. Math.*, 1(1):269–271, December 1959.

[EL00] Peter Eades and Xuemin Lin. Spring algorithms and symmetry. *Theoretical Computer Science*, 240(2):379 – 405, 2000.

[GHMP13] M. Blazey G. Hernandez-Maskivker, G. Ryan and M. Pamies. Fast lines at theme parks. *International Journal of Psychological and Behavioral Sciences*, 7(6), 2013.

[gra] Graph and its representations - geeksforgeeks. `https://www.geeksforgeeks.org/graph-and-its-representations/`. Accessed: 2012-11-13.

[HMdN04] Seok-Hee Hong, Damian Merrick, and Hugo A. D. do Nascimento. The metro map layout problem. In *InVis.au*, 2004.

[Lap92] Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231 – 247, 1992.

[MSJ16] Mirta Mataija, Mirjana Rakamaric Segic, and Franciska Jozic. Solving the travelling salesman problem using the branch and bound method. *Zbornik Veleucilista u Rijeci*, 2016.

[Nöl10] Martin Nöllenburg. An improved algorithm for the metro-line crossing minimization problem. In David Eppstein and Emden R. Gansner, editors, *Graph Drawing*, pages 381–392, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[off] Tokyo disney resort official site. `https://www.tokyodisneyresort.jp/en/index.html`.

[OOK13] H. Ohwada, M. Okada, and K. Kanamori. Flexible route planning for amusement parks navigation. In *2013 IEEE 12th International Conference on Cognitive Informatics and Cognitive Computing*, pages 421–427, July 2013.

[ope] Open street map, map of tokyo disneyland. `https://www.openstreetmap.org/#map=18/35.63294/139.88052`.

[SOO13] Takahiro Shibuya, Masato Okada, and Hayato Ohwada. A practical route search system for amusement parks navigation. *Systemics, Cybernetics and Informatics*, 2013.

[SRMOW11] J. Stott, P. Rodgers, J. C. Martinez-Ovando, and S. G. Walker. Automatic metro map layout using multicriteria optimization. *IEEE Transactions on Visualization and Computer Graphics*, 17(1):101–114, Jan 2011.

[SSS17] N. R. Syambas, S. Salsabila, and G. M. Suranegara. Fast heuristic algorithm for travelling salesman problem. In *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*, pages 1–5, Oct 2017.

[sta] Visitor statistics in tokyo disneyland. `https://www.statista.com/statistics/236159/attendance-at-the-tokyo-disneyland-theme-park/`.

[Vas] Elitza Vasileva. Github repository tokyodisneyland. `https://github.com/ladyeli/TokyoDisneyland`. Accessed: 2018-05-15.

[wai] Waiting times in tokyo disney land and disney sea. `http://www15.plala.or.jp/gcap/disney/`.