Georg-August-Universität
Göttingen
Institute for Computer Science

ISSN 1612-6793

Master Thesis

in the study program "Applied Computer Science"

Breadth First Search-Based Path Finding for Flight Connections

Florian Henke

at the Institute for

Applied Computer Science
University of Göttingen

May 5, 2018

Georg-August-Universität Göttingen
Institute for Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel.	+49 (551) 39-172000
Fax	+49 (551) 39-14403
Email	office@cs.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

Göttingen, May 5, 2018

Master Thesis

Breadth First Search-Based Path Finding for Flight Connections

Florian Henke

May 5, 2018

Mentors:

Prof. Dr. Wolfgang May
Prof. Dr. Stephan Waack

Abstract

The intention of this master thesis is to develop a path finding algorithm to find a connection between any two locations worldwide including driving, public means of transportation and airplanes. Therefore, APIs of *Skyscaner*, *eStreaming* and *Google Maps* for querying travel data are introduced and a graph structure for storing cached flight connections and airports is developed. Also a method to obtain the most important airports worldwide for inter-continental flights and to approximate flight prices is developed. Then, two path finding algorithms that can be combined to increase their performance are developed. One algorithm is called *Hotspot Search* and the other one is called *Modified Recursive and Parallel Breadth First Search (MBFS)*. The *Hotspot Search* uses only a subset of all available airports to find a connection between two airports, thus it performs very fast by accepting not to find the best connection. Its results are used as an upper bound for the *MBFS* algorithm to improve its performance. The purpose of the *MBFS* algorithm is to find the best connections between any two places worldwide. To evaluate the algorithms an application which implements these algorithms is developed. Both algorithms individually as well as the combination of them are evaluated on their performance and the quality of their results. Also different settings of parameters of the *MBFS* algorithm are evaluated on their performance and on the result quality to find the best setting for the algorithm.

Contents

1. Introduction

For people who do not live in a town with an airport, the first question when it comes to flight booking is how to find the best origin airport. In case the destination place does not have an airport either the next question that arises is which is the best destination airport. Göttingen, a city in the center of Germany with a population of 119,000 inhabitants is a very good example for that issue. Göttingen itself has no airport. But in a 200 km beeline distance around Göttingen there are nine civil airports located (Table 1). The closest airport is Cassel, but from this airport departs only approximately one flight a day. The beeline distance from Göttingen to Paderborn is twice the beeline distance between Göttingen and Cassel, however the time needed to drive from Göttingen to Paderborn equals the time needed to get to the airport of Cassel. Both airports are difficult to reach by public means of transport. The third closest airport is Erfurt which is badly reachable by car and by public transport from Göttingen. But on the fourth rank is Hanover. Its airport provides a lot of connections to European destinations and is excellently reachable from Göttingen by car and public transport. Leipzig, Dortmund, Münster and Bremen are not that close but these airports provide a lot of discount connections. Already on rank nine occurs the airport of Frankfurt which is the biggest airport in Germany and provides connections to places worldwide. Whenever several origin airports, several destination airports and maybe also different departure dates come into question a huge number of possible combinations have to be checked.

Another problem is that usually only those connections are offered in which all sub connections are served by the same airline or at least alliance. That leads to the problem that a connection between two small places can often not be found by flight search engines since no airline or alliance serves the whole connection. It also happens, that the only airline or alliance that serves the whole connection is really expensive or time-consuming. Nevertheless, there may exist good connections by booking different airlines. One example is the connection between Arusha, a small town located in Tanzania (Africa) and Balmaceda, a small town in the mountains of Chile (South America). By searching a connection between these places no results show up on traditional flight search engines like *Skyscanner* [Skyc] although there exist connections as shown in Figure 1.

The purpose of this thesis is to develop a path finding algorithm that addresses these issues. Therefore the basic requirements for the algorithm are:

- Find the best fitting origin and destination airports for the given origin and destination places and find a connection to these airports.
- Find a flight connection between these airports in case a connection exists.

Before this algorithm can be developed the underling data structure has to be defined. Therefore a multiply weighted graph structure that contains all airports as nodes and flights between them as edges will be introduced in this thesis. The multiple weights are required because flights have two properties that have to be considered for path finding, the price and the duration. Later on a method to combine both of these properties by introducing virtual costs will be described.

Intentionally it could be an idea to use a plain *Breadth First Search* to find a path on this graph between the origin and destination airport. The problem is that a lot of airports provide a lot of flights every day. The airport of Frankfurt for example provides in average 650 outbound connections a day. Since the number of outbound connections of the airports describe the branching factor of the graph a plain *Breadth First Search* would result in a huge number of paths after few steps. Additional constraints like the fact that the departure time has to be after the previous arrival time are also required but not considered by using a traditional *Breadth First Search*. Therefore, an algorithm named *Modified Recursive and Parallel Breadth First Search (MBFS)* which is based on the traditional *Breadth First Search* but proceeds recursively and in parallel, follows additional constraints and terminates paths that are not promising, is developed in this thesis.

To increase the runtime of this algorithm, an algorithm that uses only a small subset of all airports to find a connection between two given places is additionally developed. This algorithm is named *Hotspot Search*, its purpose is to set an upper bound for the duration and price of a connection for the *MBFS* algorithm.

To evaluate these algorithms an application named *Trip Planner* that implements the algorithms is developed. To obtain data sets on which the algorithms can be tested, several APIs are used. To find the origin and destination airports and to get the connection to and from them via car or public transport, *Google Maps* APIs are used. These APIs are also used for support tasks like time zone calculations. To find flight connections and the according prices, *Skyscanner* and *eStreaming* APIs are used. This application also contains a database with 634,000 cached flights and information about its airports that are obtained from these APIs. With the *Trip Planner* application the algorithms are evaluated on their runtime and quality according to different settings introduced later in this thesis.

This thesis begins with a short description of the basic terms and notions that are used in this thesis, followed by a short introduction on how to collect the data from the different APIs in

Distance (km)	Airport
41	Cassel (KSF)
91	Paderborn (PAD)
94	Erfurt (ERF)
104	Hanover (HAJ)
159	Leipzig (LEJ)
161	Dortmund (DTM)
168	Munster Osnabruck (FMO)
185	Bremen (BRE)
191	Frankfurt am Main (FRA)

Table 1: Airports within 200 km around Göttingen

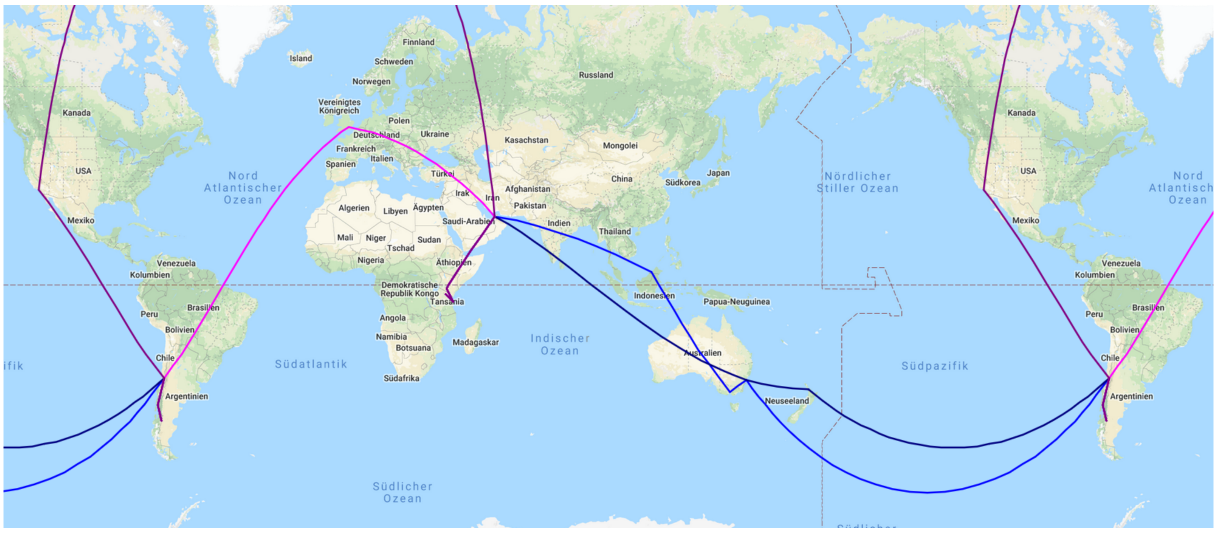


Figure 1: Connections between Arusha and Balmaceda.

Section 3. Afterwards the *Flight Graph* including the caching mechanism of the flights is described in detail in Section 4. Also a description of how to obtain a subset of the airports for the *Hotspot Search* algorithm is explained in this section. Section 5 describes the developing of the different path finding algorithms. In the beginning the traditional *Breadth First Search* is described in short. Afterwards the *Hotspot Search* is described, followed by a detailed description of the *MBFS* algorithm. Section 6 then introduces the *Trip Planner* and describes the possibilities of this application including a short overview about the implementation. The next section investigates the algorithms. Therefore the connections from Göttingen to Coimbra, from Arusha to Porto Alegre, from Arusha to Balmaceda, from Cape Town to Asuncion and from Haikou to Balmaceda which illustrates different issues and effects are used. These connections are queried by the *Trip Planer* application with different settings to evaluate the algorithms. Section 8 discusses the results of the previous section. The last section concludes the thesis and points out some further researches.

2. Basic Notions

Before describing the algorithms and its preliminaries in the next sections, this section starts with a short introduction of the basic terms and notions which are used in this thesis.

Single flight / connected flight A single flight is a direct flight between two airports without transit. A connected flight is a combination of at least two single flights between two airports via at least one other airport. Often the term flight is used instead of single flight.

connection A connection describes a possibility to come from one specified place to another. This can be throughout a single means of transportation like a car, train, bus, plain and so far or throughout a combination of several means of transportation. A connection can have just a single step like a single flight or drive, or several steps for example a drive, afterwards a connected flight and in the end a bus connection. A connection is incomplete if the end of the connection does not reach the intended destination place for this connection.

Travel Graph The *Travel Graph* $G = (V, E, P)$ is the Graph that contains all reachable places and the possible connections between them. The nodes V of the graph are the set of all places. The edges E are the set of all connections with just one step. The paths P in the sense of this thesis are connecting two nodes by using one or more edges while additional constraints like that an edge can only be added to a path if the departure time of the edge is at least one hour after the arrival time from the previous edge has to be followed. A path is a connection in the sense of this thesis between the starting and ending place of the path. By finding a connection usually the term path is used in this thesis, if a path between the origin and destination place is found the term connection will be used instead.

Cached Flights Database The *Cached Flights* database caches flights from *Skyscanner* and *eStreaming* to ensure fast access to the flight data for the path finding algorithms. These data are the basis for the *Flight Graph* introduced in Section 4.

Hotspot *Hotspots* are the 70 airports with the most intercontinental flights listed in the *Cached Flights* database. These airports constitute a subset of all airports on the world and build up a kind of own network reaching the most airports worldwide with direct flights from them. More information about the *Hotspot Airports* can be obtained in Section 4.2 and from the list of all *Hotspots* in Appendix A.2.

Origin (place) / destination (place) The origin and destination is the actual starting and ending point of the algorithm. This can be any point on the earth defined by coordinates. Sometimes it is also named origin / destination place.

Origin airport / destination airport The origin airport is the airport where a flight route starts and the destination airport is the airport where a flight connection ends. Obviously, those points have to be airports. Usually those airports are the closest, fastest or best reachable airports from an origin or destination place.

Origin Hotspot / destination Hotspot The origin *Hotspot* is the first *Hotspot* of a flight path and the destination *Hotspot* is the last *Hotspot* of a flight path. For the connection from the origin airport to the origin *Hotspot* and from the destination *Hotspot* to the destination airport *non-Hotspot* airports are used.

Geographical distance The geographical distance between two places (a, b) is the beeline distance between the coordinates of a and b. Often the geographical distance is just called distance.

Travel distance The definition of the travel distance in this thesis is based on the definition of the distance of a single and connected flight from [Hor10]. The distance of a direct connection between a and b while $a, b \in V$, an edge $(a, b) \in E$ exists, V is the set of all places and E is the set of all direct connections, is determined by the geographical distance between the coordinates of a and b. The distance of a path $p \in P = (e_1, e_2, ..., e_n)$ is determined by the sum of the distances of its single direct connections $\sum e_i$ and not by the geographical distance of the start and end coordinates of the path.

Completed distance The completed distance represents the progress of a path. It can be determined by:

$$distance_{completed} = distance_{full} - distance_{remaining}$$

While $distance_{full}$ is the geographical distance between the origin and destination places and $distance_{remaining}$ is the geographical distance between the current place and the destination place.

Duration The definition of the duration in this thesis follows the definition of the duration of a single and connected flight from [Hor10]. The duration of a direct connection between two places a and b, while $a, b \in V$ for V as the set of all places and $\exists(a, b) \in E$ for E as the set of all direct connections, is the local arrival time at place b minus the local departure time at place a plus the time difference of both places. But the duration of a path of multiple connections is not the sum of all its single connections. It is the sum of all its direct connections and additional all transit times between the connections.

$$duration_{path} = \sum(duration(subConnection)_i + transitTime_i)$$

While i iterates over all sub connections of a path and its associated transit times. That is equal to the duration between the local departure time of the first connection and the local arrival time of the last connection plus the timezone difference of the origin and destination place.

Virtual costs The edges of the *Travel Graph* introduced previously are have multiple weights, the duration and the price. For graph search algorithms a metric how to deal with this multiple weights is required. For this purpose the virtual costs are introduced. They are a metric to combine the price of a connection with the duration by applying a financial penalty for each hour a path requires to reach the destination. This penalty is the price that is one hour worth for the user. The virtual price can determined by:

$$virtualCosts = price_{path} + duration_{path} \cdot x$$

While $price_{path}$ is the sum of the prices of all parts of the path and x is the user defined penalty for each hour the connection requires.

valid path A valid path (also valid connection) is a path from the origin place to the destination place where each single part of the connection departs from the same point and at least one hour after the arrival of the previous part.

Algorithmic step An algorithmic step in the sense of this thesis is the expansion of all paths that have not been terminated or reached the destination already. For example in the first algorithmic step all paths that have not been terminated already or have reached the destination are expanded the first time, in the second algorithmic step all paths that have not been terminated already or have reached the destination are expanded the second time and so forth. The number of algorithmic steps that an algorithm required represents the number of expanding steps from the longest path generated by the algorithm.

3. Collecting Data

Before searching a path in a graph, the graph needs to be filled with data. In the case of this thesis does that mean that data sources for all means of transportation are required. This section describes which data sources are used by *Trip Planner* application and what are there abilities and limitations to obtain the required data for the path finding algorithms.

Flights can be booked from the airline which provides the flight and from travel agencies which are distributing flights from different airlines. The price for the same flight can differ strongly between different suppliers. In the beginning of 2018, 424 airlines were registered by the IATA [IAT], the total number of travel agencies is unknown, but probably a lot higher than the number of airlines. To get a full set of data, an API to all these airlines and travel agencies has to be implemented and maintained. Especially the maintaining part is really extensive due to the fact that the APIs are changing continuously.

Some companies like *Skyscanner*, *ITA Matrix Search*, *eStreaming* and *Travelport* are specialized in gathering these data to sell them afterwards [ITA] [Skyb] [eSt] [Tra]. Their business concept is to get provisions from the airlines and travel agencies for each flight booked via a link provided by them [Skyb] plus their customers have to pay for their API access [eSt] [ITA]. The fee can depend on the possible number of requests per second or day [eSt]. The customers gain from this service is that they have to implement just one API instead of hundreds.

These companies cache all request results from the live price APIs in a database for several days and provide them via cached flight APIs. Some of these companies provide restricted free access to their cached flight APIs. This master thesis has no budget for API accesses, therefore only the free APIs have been used for the *Trip Planner* application.

Additional to the flights the connection from the origin place to the origin airport and the way from the destination airport to the destination place needs to be found. The *Trip Planner* application uses *Google Maps* APIs to solve this problem. The APIs used for the *Trip Planner* application are shortly described in this section. For a more detailed view on the APIs the project report [Hen18] can be considered.

3.1. Google Maps

Google provides a huge range of different APIs for a lot of different applications related to map use. The APIs are splitted in two groups, the web APIs and the web-service APIs. The web APIs are all those APIs which can be embedded through *JavaScript* directly on the client side [Gooe] and the web service APIs are those APIs which have to be accessed by an HTTP call from the server side [Goog].

The *Google Maps JavaScript* API is the most important web API used for the *Trip Planner* application. It provides all methods required for embedding a dynamic *Google Maps* map into a web page by using *Java Script*. This API includes methods for fully customizing the map, like the zoom level, the maps center, which functions are allowed for the user and a lot more. Also functions for drawing on the map are provided by this API, for example is it possible to draw lines and polygons or show content on the map by using markers and symbols. Also info windows with text are possible to show on the map. Together with the *Google Places API JavaScript-Library* it is possible to get detailed information about more than 100 million places worldwide, like cities, countries, businesses, locations and a lot more. This library is able to give place information for a given latitude and longitude and to return the latitude and longitude for a given place [Gooe] [Goof].

The *Google Maps Direction* API is the most important web-service API provided by *Google Maps*. It allows to request a path description for the fastest route including some alternative routes between two or more arbitrary places in the world. For the path calculation different parameters like the means of transportation, including driving and public transport and the time of the trip are considered. The result includes the duration considering the current and expected traffic, fares, a polyline to draw on a map and a detailed route description [Goob].

The *Google Maps Distance Matrix* API returns the distance and the duration between several locations but not a full route description like the *Google Maps Direction* API. This API can be called with a list of up to 25 origin and destination addresses, in this case a result set with the distance and duration for all combinations between origin and destination will be calculated and returned. The huge advantage of this API is, that it is much faster than the *Google Maps Direction API*. A detailed description of this API can be obtained from the project report [Hen18].

An other web-service API provided by *Google Maps* is the **Google Maps Geolocation** API which returns the current location and the accuracy of the calculated location of the user, based on data of mobile towers and the wifi network [Gooa]. The *Google Maps Geocoding* API converts addresses to coordinates and coordinates or place IDs to full addresses [Gooc]. The *Google Places* API allows to access data of over 100 million places in the whole world like cites, countries, businesses, locations and a lot more. For each place data like address, name, opening hours, rating and more are available. This API also allows auto completion of places and the search of places near the current or any arbitrary location [Gooh]. By passing the coordinates of a location and a date to the *Google Maps Time Zone* API it returns information about the time zone and the time difference to UTC time [Good]. More details and a more technical view on the APIs can be obtained from the project report [Hen18].

Beside these APIs Google provides several APIs for iOS and Android as well as for Java, Python, Go and Node.js. This section is limited to a short overview of the possibilities given by the *Google Maps* APIs, there are a lot more functions and APIs provided by *Google Maps*. The project report [Hen18] provides a deeper and more technical view into the possibilities of these APIs.

Google Maps allows free but restricted access to all their services for public, noncommercial use. The lowest access restrictions apply for the mobile APIs. The *Google Maps Android* and *iOS* APIs have unlimited free access. The *Google Place* API has still a very high limit of 150,000 requests a day. The *Google Maps Java Script*, *Static Maps* and *Street View Image APIs* are restricted to 25,000 loaded maps a day. The web services wich are the rest of the APIs like *Google Maps Direction* or the *Elevation* API are restricted to 2,500 requests a day. Google offers a simple way to increase the daily limit without entering a contract by paying 50 cents for each 1,000 requests over the free limit up to 100,000 requests a day. For commercial or access restricted applications, other rules apply. But especially for Android applications, there are still free opportunities available [Gooi].

3.2. Skyscanner

Skyscanner is a flight, hotel and car rental search engine. It operates around the globe in over 30 different languages and 70 different currencies. *Skyscanner* tries to connect all travel agencies and airlines world wide, in order that the users have access to almost all travel offers that are available on the market to make the best deal at any time [Skyb]. *Skyscanner* offers several APIs which provide different kinds of data. The main APIs are distinguished in three different fields. The *Cached Data* APIs provide cached data from previous requests. The *Live Price* API provides the current data of a requested flight and enables a link for booking this flight (This API is currently not available [Skye]). The *Place* API provides static data about places available on *Skyscanner* [Skya]. APIs for car rental and hotel search are available as well but will not be considered in this report because they are out of topic [Skya]. The access to the *Skyscanner* APIs is not restricted by a daily limit. They have only restrictions on requests per minute. The *Skyscanner Cache* API is restricted to 500 requests a minute, while the *Skyscanner Live Price* API is restricted to only 100 requests a minute and the *Skyscanner Car Hire* API is restricted to 100 requests a minute as well [Skyd].

With the *Skyscanner Place* APIs it is possible to request a *Geo Catalog* of all continents, countries, regions, cities and airports with further location and name information that are listed on *Skyscanner*. Also further information of a specific place can be accessed by the *Place Information* API. With the *List of Places* API *Skyscanner* realizes an auto complete for places listed on it [Skya].

The *Skyscanner Cached Data* API is supplied by data from requests on the *Skyscanner* homepage. *Skyscanner* provides different *Cached Data* APIs for different use cases, but in the end there is nothing that could not be done by the *Browse Quotes* API but with one of the other. There is just a difference in the representation of the results that should simplify the handling of the received data [Skya].

The *Skyscanner Cached Data* API requires the origin and destination place, the outbound and a possible inbound date and the country, currency and language of the user. Thereby the origin and destination place can be an airport, city, country or even anywhere and the outbound and inbound date can be a specific date, a month or even anytime. The API returns the cheapest price, the carrier and the date of recording the dataset for one direct flight if available and for one connected flight [Skya].

The *Flights Live Price* API has been under construction for several months. *Skyscanner* promises, that the API will be available soon again [Skye] but it seems as they will have it available just for business partners in the future. Before, it was possible to request very detailed flight information, which returned the exact times of the flight, available seats, and even direct links to different agencies to book the flight [Skya].

3.3. eStreaming

eStreaming is an API which provides cached price data from over 10,000,000,000 daily requests[eSt]. The *eStreaming* API belongs to the *CEE Travel Systems* company which also owns the live price travel API *Travelport* [CEE] [Tra]. From these servers *eStreaming* gets all requested data for updating its cache once a day [Gonb]. *eStreaming* provides different APIs for different use cases. If detailed offers for a specific connection are required the *Cache* API can be used. To access these offers from a view point in the past the *Historical* API is the right choice. For trips where the time period of the trip is flexible the *Flex* API and the *Fly From To* API should be used. And if all connections for an airport are required the *Fly From* API returns the best results [Gonc].

Each API request requires to select a point of sale. This choice has to be made because airlines are distributing their flights in different ways and with different fares to different markets [Gonc]. Depending on the exact request type additional attributes like origin and destination airport, outbound and a probable inbound date or date range, days of stay or a quote date time are required. The API response is a JSON file that includes a parameter base64GzippedResponse. This parameter contains a JSON file again, compressed as .gz and encoded as Base 64 [Gond]. The second JSON file contains for each connection stored in the *eStreaming* database, that matches the request string, values for the fare, carriers, duration, departure and arrival date and time, flight number, legs, and more depending on the exact request type.

By using a free account the access is restricted to 1000 requests a day and one request per second. The pricing system of *eStreaming* is to buy additional requests per second. From the first request per second purchased the daily limit is omitted. Each request per second costs between 3.50$ and 5.00$ a day depending on how many requests are purchased in total [eSt].

Originally, IATA codes were intended to describe airports, but nowadays IATA codes can also describe train stations and bus stations that are connected to airports in any way. For example is it possible to book a flight from Qatar to anywhere in the world while the first part of the trip is a bus ride from Qatar to the airport in Dubai. Therefore the bus terminal in Qatar got an IATA code. These connections that are not a flight, are really hard to distinguish from the real flights. If this would be ignored and the Bus and train stations with an IATA code would be handled as airports several problems could occur. The first one is that all parts of a connection are stored in the database as an own connection as it will be described in more detail in Section4. Therefore would it be possible that this bus or train ride is used to catch another flight later on. But often this connections can not be booked separately. Another problem is that the *Trip Planner* application queries further information of airports like there coordinates which are required for the algorithms. These information are only available for airports but not for bus and train stations with IATA codes. These problems are solved by using the *Skyscanner's Geo Location* API which provides a list of all airports including their IATA code. Therefore all connections found by *eStreaming* are checked against this list, if one stop of this connection is not in the database this part of the connection will not be considered for the result.

4. Flight Graph

After explaining where the travel data can be obtained from, this section describes how to cache the obtained data to get fast access on it when applying the path finding algorithms later. Also how to deal with the problem of incomplete data sets obtained from the APIs is part of this section.

The Flight Graph is the graph that contains all airports and flight connections available for the *Trip Planner* application. The base for the graph is a *PostgreSQL* database. Like each graph ontology the basic notions are vertices, edges and paths. The Flight Graph is a directed, edge-labeled graph of the form $G := (V, E, P)$. V is the set of all vertices, each vertice represents one airport, $E \subseteq V \times V$ is the set of directed edges between the vertices, whereof each represents a flight connection and P is the set of paths. While the set of paths is defined as a transitive sequence of edges in the usual notion of a graph [Hor10], the set P of paths in this graph is a subset of these usual paths since a path has to satisfy certain additional constraints like the aspect that the departure time of a following edge has to be chronologically after the arrival time from the previous edge. Nevertheless, each path $p \in P$ consists of single or multiple connected edges and is therefore also a path in the traditional sense. Each path p that ends in a vertex x can be extended by an edge (x, y), denoted by $p \circ (x, y)$ if the edge (x, y) satisfies the given constraints for a path.

First, this sections deals with the database that stores the Flight Graph. Later in this section it is described how a subset of the vertices $V_h \subset V$ is generated that spans a smaller graph with a small subset of all edges $E_h \subset E$ with the characteristic to reach most other airports as possible with direct flights from the airports of V_h to reduce the run time of the algorithm in the end. The last part of this section describes how it is possible to determine prices of flights in the database that did not come with a price from the APIs.

4.1. Caching Graph Data

It would be a possible solution not to store the whole graph locally but always requesting APIs for the part of the graph that is required. The obvious advantage of this solution is that all data are always up to date and the space complexity would be low. But requesting a flight via an API needs a long time, if for one connection a lot of flights has to be requested from APIs the time effort would be huge, before a result could be returned. Also most of the APIs, especially the flight finding APIs, have a small limit of requests per second, day or even both [Skyd][eSt]. That leads to the need of caching the flight data before they are required. The *Cached Flight Connection Database* stores all flights, including information like the cheapest price, departure and arrival time and the quote date time. The database also stores each airport in the world including its coordinates, IATA code, the related city and the daylight saving time difference to UTC for the wintertime and the summertime as well as the dates for time shifts. Also one table for all cities that have an airport, one table for all countries and one table for all continents is stored in the database.

The core table of the database is the flight_connections table. This table has columns for the origin, destination, departure and arrival date and time, duration, quote date time, weekday, price, approximated price described in Section 4.3, currency, carrier, flight number and a connection number which is an integer ID that is used as a foreign key to a second table that stores all legs that the connection includes. This table has one entry for each flight. Due to the reason that different APIs are used to request flights and some flights are direct and others are connected, the stored connections can be distinguished into three different kinds. The first kind is direct connections where no details are available for. That are flights obtained from *Skyscanner*, for those flights it is only known that a flight exists on the specified day with the origin and destination and the cheapest offer. For these flights is no exact departure and arrival time known also is the duration and flight number unknown, as well as the number of flights on the day between the origin and destination place. These flights are distinguishable from the other flights because the duration is null.

The second kind of flights are also direct flights but with further information. For these flights the exact departure and arrival times are available as well as the flight number, the duration and further information. Some of these flights have a price and others do not. That kind of flights can be distinguished from the other kinds because all these flights have a flight number but the connection number is null since these flights are direct. The last kind of flights are connected flights. These flights have all information like a precise departure and arrival time as well as a duration, price and so on except a flight number since they are connected from flights with different flight numbers. These flights are distinguishable from other flights because they have a connection number.

The APIs return a lot of results for a single flight since a flight is usually offered by different providers. But for the path finding algorithms only the cheapest offer is of interest. Therefor, because of space complexity reasons just the cheapest offer of a flight is stored into the database. While this sounds trivial in the first moment it becomes more complex after a deeper view into it. Since the flight number is not usable as a unique identifier because the same flight number can repeat each day again even for different connections, a flight number can also be the same for a flight that has several layovers where each part can be booked separately like a flight from Paris to Arusha going on to Dar es Salaam and back to Paris with the same flight number. Also a problem is that one flight can have several flight numbers if it is shared by different airlines. Therefore other unique identifiers are required. The *Trip Planner* application uses the origin and destination place as well as the departure and arrival time as unique identifiers for a flight or connection. This does not mean that it can be guaranteed that both flights or connections really are the same if the application identifies them as similar. But the algorithms used for path calculation would always decide for the cheapest flight or connection anyway. If two connections have the same departure and arrival time, it is sufficient to keep just one of them in the database. For example if a connection has three legs and the first and last one are the same but the middle one is different the database would deal with these connections as if it was the same and would just keep the cheaper one. This saves a lot of storage and the path finding algorithm would always chose the cheaper connection of them anyway.

The *Trip Planner Server* application has a function to update this database. The first step of this method uses the *Skyscanner Geo Location* API to check if new places are available. In the second step the *Skyscanner Cached Flights* API, which finds all outgoing flights from one specified airport, is used to check for each airport whether there exist flights that are not in the database already. But these flights found by the *Skyscanner* API have a departure date and a price but no exact times and no information about how many times a day this connection is served. Therefore the last step of the update process is that all connections with missing information are requested from the eStreaming API. This API returns detailed information like the exact departure and arrival time. Due to the limitations of the APIs this update process is able to update the flight connections of one specific day each day. Therefore a period of two weeks in April 2018 was chosen as a test period. For six month this two weeks period was updated continuously to use it for the evaluation of the path finding algorithms afterwards.

All database update mechanisms are organized modularly. That means it is simply possible to change single steps like finding all flights that an API should request or to change the API used for the requests just by calling a different method. It is also possible to skip single steps of the update mechanism. Changing the whole database is even simpler just by changing the database connection information stored in one global class.

Currently the database has entries for 4,919 airports worldwide whereof 3,490 airports are served. All in all 32,193 different direct connections and 12,050 different connected flights are stored in the database with 634,655 different flights in a time period of 14 days (up to 61,635 flights a day). Due to the fact that some connected flights need few days some of the flights are outside the time period while the "flights a day" value represents how many flights are really departing on one specific day. The queries required to request these values can be found in Appendix A.1.

4.2. Hotspot Airports

Hotspots are a subset of all airports listed in the *Cached Flights* database. *Hotspots* are representing big airports that can be reached from a lot of other airports in the same region and that provides direct connections to a lot of airports on other continents. Therefore these airports should be particularly suitable for switch over to come from one continent to any other continent. The idea is, to develop an algorithm that finds a fast path to the next *Hotspot* from any airport in the world and to find another *Hotspot* that has a fast path to the destination airport. Afterwards both *Hotspot Airports* can be connected just by using other *Hotspots* or in the best case there is a direct connection between both *Hotspot Airports* or they are even the same airport.

The *Hotspot Airports* are determined by considering for a specific day where the cached flight database has as many flights as possible from all airports all direct connections to an airport on another continent (inter-continental flights). Each connection between two airports is considered once even in case that several flights are available on this day for this connection. These connections are counted for each airport. That result for each airport is the number of airports on other continents that can be reached by direct flights on this day from the considered airport. The 70 airports with the most connections to airports on other continents are chosen as *Hotspot Airports*.

Appendix A.2 shows a list with all *Hotspots* determined by the method previously described. Few things are noticeable on this list. First, all big airports in Europe and North America are on this list but just one airport for whole Russia, just one airport in Africa and only very few airports for South America in the end of the list. If this list would consider only the first 50 airports, South America would have just one airport in the list as well. The reason that Russia has only one airport in the list is that almost all flights in Russia go to Moscow. Almost always to come from one to another Russian city it is necessary to switch in Moscow especially to leave the country. For South America and Africa is the reason that the main *Hotspots* for these regions are located in Europe and the Middle East. The most countries can be reached from there former colonial powers. That means the South American countries are especial reachable from Lisbon (Portugal) and Madrid (Spain) and the African countries are reachable from Paris (France) and London (Great Britain). These are also the regions where a lot of people are willing to go to these regions since the South Americans and Africans are not traveling lot between their countries. For the African region also Istanbul (Turkey) and Dubai serve as *Hotspots*. That is caused by the fact that these countries have big airlines which business model is to serve the European marked and the geographically good location to Africa.

4.3. Approximation of Missing Prices

Often it happens that the APIs are returning connected flights with single legs where the flights of the single legs have not been stored in the database already. In this case the whole connection will be added to the database. But also the single legs are added to the database as direct connections, since it can be expected that the single legs are bookable separately as well. In this case the API provides only a price for the whole connection but no price for each single leg. If later on this direct connection is found by another API call the price will be added to the flight. But often it happens that the APIs do not have an entry for the single flights even so they provide the whole connection. In this case this flight has no price in the *Cached Flights Database*. This applies currently to 460,000 of the 1,300,000 connections stored in the database. For the path finding algorithms the price is a very important evaluation criterion. This leads to the problem that a lot of flights can not be considered for path finding. To prevent that, this section introduces a method to approximate the price of a flight by considering the prices of other similar flights and the general flight price behavior on the day of the flight [Hor10].

In two different cases the price for a flight can be approximated:

- The price of other flights with the same flight number and the same origin and destination is known by the database. This is probably the best case, especially if the number of flights with the same flight number on another day and a price stored in the database is high, because it can be expected that the price of the same flight, with the same airline on another day cost in average roughly the same. In this case the price of the flight is approximated by:

$$price(flight) = avg(price(sameFlightNumber))$$

While $price(sameFlightNumber)$ is the set of all direct flights that have a price and the same flight number, origin and destination like the flight for which the price is approximated for, stored in the database. Later in this section a correction factor for this method is described.

- The price of other flights on the same route (with the same origin and destination airports) is known by the database. If the same route is provided by different airlines, usually the price of the flights are quite similar for all airlines because people are unwilling to pay a much higher fee for the same service if they can receive it also for a lower charge [ETKY03] [Far]. Therefore an approximation for the price of a flight can be calculated by the average over all flights on the same route:

  $$price(flight) = avg(price(sameRoute))$$

  While $price(sameRoute)$ is the set of all direct flights that have a price and the same origin and destination airports like the flight for which the price is approximated for, stored in the database. Later in this section a correction factor for this function is described, too.

  But there are limitations for this method. For example if the same route is provided by a discount airline and a classic operator as well and the prices are available just for one of them. In this case the service is very different and even so both flights have the same origin and destination the price can be very different. Another reason for different prices can be a different departure time [Far] [Che]. Therefor this case should be just the second choice if the first solution can not be applied. However, in average this method will return an approximated flight price.

The prices of flights can differ strongly within different days [ETKY03] [Che] [Far]. For example flights are usually really expensive on Sundays and cheap on Wednesdays [Far]. Also international common holidays like Christmas and Easter or the summer time can influence the flight price massively [ETKY03] [Che] [Far]. To soften these effects a correction factor is used. This factor assumes that the price between the same flights on different days follows the average price difference between the flight prices on all flights in the database and the flight prices on the day of the flight [ETKY03]. Therefore this factor uses the average of the prices from the day of the flight and the average price from all flights stored in the database as follows:

$$correctionFactor(day) = \frac{avg(price(flights_{day}))}{avg(price(allFlights))}$$

While $price(flights_{day})$ is the set of the prices from all direct flights on the considered day and $price(allFlights)$ is the set of the prices of all direct flights stored in the database.

The previously approximated flight price has to be multiplied by the correction factor to soften the flight price difference between different days: That leads to the following functions for calculating the approximated flight price for a certain flight:

$$softedPrice(flight) = avg(price(sameFlighNumber)) \cdot \frac{avg(price(flights_{day}))}{avg(price(allFlights))}$$

if the database has entries for other flights with the flight number that contain a price and

$$softedPrice(flight) = avg(price(sameRoute)) \cdot \frac{avg(price(flights_{day}))}{avg(price(allFlights))}$$

if the database has only entries for other flights on the same route that contain a price.

As it is not a core part of this thesis to discuss how to approximate flight prices, this rough approximation of the price is sufficient. For a better calculation especially the correction factor could be optimized by different aspects.

- The average price difference between flights of the airline that provides the flight for which the price is missing and the airlines of the flights that was used to calculate the approximated price should be considered. This method would consider the price difference between discount airlines and classic airlines [ETKY03] [MPR09].
- The flights used to calculate the correction factor could be limited to flights that have the same origin or destination country or even airport because holidays are often national or even local.
- By calculating the correction factor, flights of the same week day could be considered with a higher priority because the flight prices often depend on the weekdays [Far].
- The remaining seats of a flight have a significant impact on the flight price. If this information is available it could improve the flight price approximation substantially [ETKY03].
- The remaining days until the date of departure could be considered for the approximation. Usually the flight price increases until few weeks before departure. On this point in time the price falls down and afterwards the price is increasing again [ETKY03] [Far] [Che] [MPR09]. If a huge data set of flights from the same airline was available these points might be identified for improving the approximation.
- Especially for discount airlines but also for other airlines the length of the flight is an indicator for the price and could be considered when calculating the price [MPR09].

In general the approximation gets more accurate if a bigger data set is used and more aspects are considered. To define which aspect should considered by which proportion and what are the best approximation methods could be part of a separate thesis.

5. Path Finding Algorithms

After defining the underling data structure, the *Flight Graph* in Section 4 and introducing the different APIs to get the travel data in Section 3, in this Section the path finding algorithms are described in detail. This algorithms are evaluated afterwards in the following sections.

When looking for an algorithm to find connections in a graph, the first intention could be to use a simple shortest path algorithms like *Dijkstra* [MS08], *Breadth First Search* [AG87] or *Depth First Search* [Eve11] algorithms. But after performing a more detailed analysis this approaches turns out as inappropriate, since it is not the goal to find the shortest path or just any of many paths, but to find the best path regarding a lot of criteria. Simple path finding algorithms are not sufficient for this complex task. Especially the price and duration have to be considered finding the best path. Even so that these criteria could be considered by combining both - the price and the duration - as it is done later on by using the virtual costs instead of the price of a connection, further constraints have to be contemplated. Especially the fact that a departure time has to be chronologically after the arrival time is not feasible by using the previously stated algorithms directly.

Therefore, the approach of this thesis is to use *Breadth First Search*-based algorithms with a set-oriented approach to use the runtime enhancement of databases on data operations and further constraints. Also parts of the *Dijkstra* algorithm find use in the algorithms of this thesis, such as for the termination criteria as it will be described in Section 5.3.2. These algorithms are modified for the travel planning use. On the one hand the modification is required to strongly increase the runtime by allowing the algorithm to proceed recursively and parallel and on the other hand to implement additional constraints, allow to find the best paths instead of the shortest path and to allow the algorithm to find more than just one solution. Also especially complex data-intensive tasks are processed on the database level to improve the efficiency of the algorithm.

The first part of this section shortly describes the basics of a traditional *Breadth First Search* algorithm. The second part describes the *Hotspot Search* algorithm which intention it is to use only a predefined subset of all airports to find a connection between two arbitrary places. The third part describes the main algorithm of this thesis, the *Modified Recursive and Parallel Breadth First Search*, in detail. The last part of this section introduces an approach to combine both of the previously described algorithms to improve the runtime.

5.1. Breadth First Search

Given a connected graph $G = (V, E)$ with V as the set of vertexes, E as the set of all edges and a root node $r \in V$ which would be in the case of flight paths the origin airport. The *Breadth First Search* would span a tree that returns the shortest path to any node $n \in V$ from the root r [KS05]. Shortest path in this case means the path from r to any n with the minimum number of edges $e \in E$ over all possible paths from r to n [AG87].

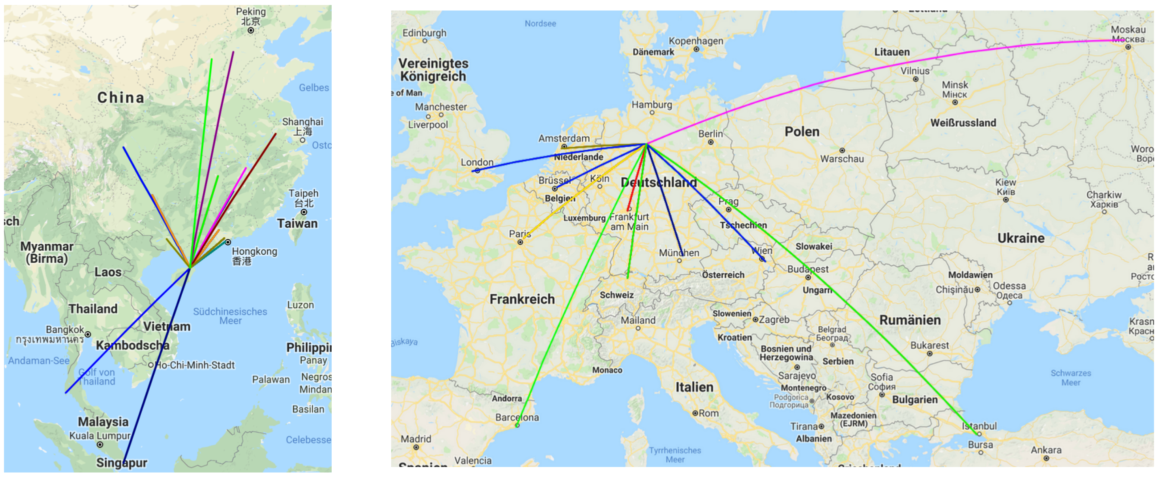
The initial step of the *Breadth First Search* is to mark the root node as visited and put it into a queue. Until the queue is empty, the first node of the queue is removed and has to be marked as visited as well as all its unmarked associated nodes have to be added to the queue. In case not the whole tree is required, but only the shortest path to a specified node $n \in V$, the algorithm can terminate as soon as n was found. The rest of the graph doesnt't need to be expanded in this case [Zus72].

In general the time complexity of the *Breadth First Search* can be expressed as $\mathcal{O}(|V| + |E|)$ because each vertex and each node have to be explored in the worst case. The time complexity is $\mathcal{O}(|V|)$ in the worst case, if all vertexes are connected with the root node [CLRS01]. Another possibility is to set the complexity in relation to the number of hops necessary to reach the searched node. The time and space complexity can be expressed as $\mathcal{O}(b^{d+1})$ while b is the branching factor and d is the distance of the searched node (measured in number of edge traversals) from the root. Especially for huge graphs where the searched node is to be expected to find within a low number of hops this calculation for the complexity is the more accurate one [RNC+03].

5.2. Hotspot Search

A plain *Breadth First Search* from one to another airport grows very fast to a huge number of different paths that have to be expanded. Because the computing power is limited and the algorithm should return a proper result within a short period of time, this expansion of the search graph has to be limited. One idea is to limit the number of open paths by applying termination criteria that terminate paths that seem not to come to a good result in the end. Another idea is to keep the graph lean by using just a few airports instead of the whole graph. Both methods are not trivial. For terminating paths with low chance of success a metric for deciding at which point the chance for success is low is necessary and for using just a subset of all airports the problem is how to determine the best subset.

Section 4.2 describes a possible solution of how to determine a subset of the 70 most important airports out of all airports. These airports are named *Hotspots* and build up their own flight network around the whole world. The *Hotspot Search* algorithm uses mainly these airports to find a path between the origin and destination airports. Only if the origin or destination airport is not connected to one of the *Hotspot Airports*, connections to other airports are considered to find a path from the origin airport to a *Hotspot Airport* and from the last *Hotspot Airport* to the destination airport. Another special case is that a connection to the destination airport is found by the expanding process before the first *Hotspot Airport* is found. In this case this connection is used instead of using *Hotspots*. If for example a connection from Hanover, a town in the northern part of Germany to Varna, a costal city in Bulgaria is searched. The airports of both cities are non-*Hotspot Airports*, but a direct connections between both exists. In this case this connection is used instead of using a connection via *Hotspots*. The path that is found by this algorithm is not necessarily the best, since it does not consider all possible connections, but if there exist



(a) (b)

Figure 2: Inbound connections from Hotspots to Haikou (a) and outbound connections to Hotspots of Hanover (b).

a connection between the origin and destination airport this method finds a path. The idea is to find the path with the least possible number of required stops by using the *Hotspot Airports* for the connection. Only if the origin or destination airport is not directly connected to one of the *Hotspots* the other airports are considered for the path calculation as well. This section describes this algorithm in detail.

The algorithm will be visualized through an ongoing example throughout the whole section. For the example a flight from Hanover, a town in the northern part of Germany, to Haikou a town on Hainan, an island in the south of China, popular for its tourism, is used. Both airports are not *Hotspot Airports*.

The *Hotspot Search* algorithm can be separated into four different parts. The algorithm starts with finding *Hotspot Airports* which has paths to the destination airport. It is required to search the *Destination Hotspots* before searching the *Origin Hotspots*. One reason for this is to enable that a connection without *Hotspots* is identified if a direct connection to it was found by searching a path from the origin airport to the *Origin Hotspot*. Another reason is that the path from the *Origin Hotspot* to the *Destination Hotspot* has to be applied right after the path finding from the origin airport to the *Origin Hotspot* for performance purposes. Another part is to find a path from the origin airport to the first *Hotspot Airport*, the third part is to connect the *Hotspot* reached from the origin and the *Hotspot* that reaches the destination, if they are not the same. The last part is to combine all connections and find the path from the last *Hotspot Airport* to the destination airport. The database includes connections from and to 3,481 different airports, while 1,693 (49%) airports are *Hotspots* or direct connected to them; the other 1,797 (51%) airports needs at least two flights to reach a *Hotspot Airport*.

Hotspot to destination airport The first part of the algorithm is to find a *Hotspot* that is connected to the destination airport with least hops as possible. If the destination airport is already a *Hotspot* the algorithm jumps to the next step, otherwise a modified *Breadth First Search* is used to find a path between a *Hotspot* airport and the destination airport. Two list are required for this algorithm. One list is named connection list. This list stores for each airport that was visited by the *Breadth First Search* and all its outbound connections. The other list is named airport list. This list contains one entry for each airport that was added to the graph during the last cycle of the following loop, these are the airports which have to be expanded in the next step of the loop. The airport list is initialized with the destination airport.

The loop starts with checking for all airports listed in the airport list, whether the connection list contains entries for this airport as well. If the airport is listed in the connection list as well, this airport was visited before by the *Breadth First Search* already and does not need to be expanded again. Otherwise all inbound connections to this airport that have a *Hotspot* as origin are queried from the database. Each new origin airport is now added to the airport list and each connection is added to the according origin entry in the connection list.

```
$\forall x \in$ airportList
  if(Not connectionList.contains(x)
    $\forall y \in$ inboundHotspotFlights($x$)
      add $y$ to connectionList(originAirport(y))
      add origin of $y$ to airportList
```

Code Snippet 1: Pseudo-code to find the next connections from *Hotspots*.

$inboundHotspotFlights(x)$ queries all inbound flights for airport x that have a *Hotspot Airport* as origin from the database. If after looping throughout all airports from the airport list one or more new connections from a *Hotspot* were found, the algorithm terminates and returns the connection list. Otherwise no connection from a *Hotspot Airport* to one of the airports in the airport list exists. In this case the same loop will be applied again, but instead of querying all inbound connections coming from a *Hotspot Airport* now connections from all airports are queried.

```
$\forall x \in$ airportList
  if(Not connectionList.contains(x)
    $\forall y \in$ inboundFlights($x$)
      add $y$ to connectionList(originAirport(y))
      add origin of $y$ to airportList
```

Code Snippet 2: Pseudo-code to find the next connections regardless whether they are from *Hotspots* or not.

$inboundFlights(x)$ queries all inbound flights for airport x from the database. If again no connection was found, the algorithm terminates and returns null because there is no connection available between the origin and destination airport. Otherwise the algorithm will be applied again until at least one connection from a *Hotspot airport* was found.

For the previously introduced example in this section between Hanover and Haikou a *Hotspot Airport* close to the destination has to be found first. Therefore the closest airport to the destination will be chosen and added to the airport list. In the case of Haikou this is *Haikou Meilan International Airport*. In the next loop run this airport is expanded. First all inbound connections within five days are queried from the database. If no connections are found the same query would be done by querying all inbound connections of the airport regardless whether these connections are from *Hotspots* or not. But in this case several connections from *Hotspots* are available. Figure 2a shows all available inbound connections to Haikou from *Hotspots*. One of the connections is the connection from Hong Kong to Haikou. Friday 2:15 pm and Saturday 2:10 pm are some of the departure times.

The connection list caches flight connections from the destination *Hotspot* to the destination airport. It makes sense not to query all flights regardless its departure time from the database when querying the inbound connections of an airport. Since the arrival time at the destination *Hotspot* can not be known before knowing which are the destination *Hotspots* there is no earliest departure time known for the connection. Instead the given departure time for the whole connection could be used, because a connecting flight can not depart before the departure of the first flight. This already excludes all flights in the database before this time. The latest possible departure time can just roughly be approximated. In this thesis it is expected that the duration of a whole connection is not longer than five days. Therefore the request is limited to flights that depart between the requested departure time of the whole connection and five days afterwards. This limits the flights that have to be cached massively but does not limit the possible connections.

Origin airport to Hotspot The second part of the algorithm is to find a connection from the origin airport to a *Hotspot* with least hops as possible. If the origin airport is already a *Hotspot* the algorithm jumps to the next step, otherwise paths to *Hotspots* will be searched by expanding all paths until a *Hotspot* is reached. This algorithm requires just a list for all paths. This list is initialized with a connection to the origin airport.

The algorithm expands all existing paths. This can be done in parallel by applying the following steps for all paths in parallel. In the first step it is tried only to consider flights to *Hotspots*. First, each path that is expanded queries all flights departing between one hour after the arrival of the previous flight and 24 hours afterwards. The minimal transit time on an airport is set to one hour and the maximum waiting time for the next flight is set up to 24 hours, thus flights that depart just once a day are considered as well. Afterwards it is checked for each new connection whether it has to be added to the path. A flight has to be added if the next airport has not been visited on the path already, to prevent cycle. Also for all flghts with the same origin and destination airport only the best flight is chosen. The best flight is defined by a metric. That could be the first flight, the cheapest flight, or a combination of both, the best price performance ratio for example. More about metrics to choose the flight can be found in Section 5.3.2.

If it was not possible to find a direct flight to a *Hotspot*, the same algorithm is applied again but this time for each path all outbound flights are queried from the database and not only these

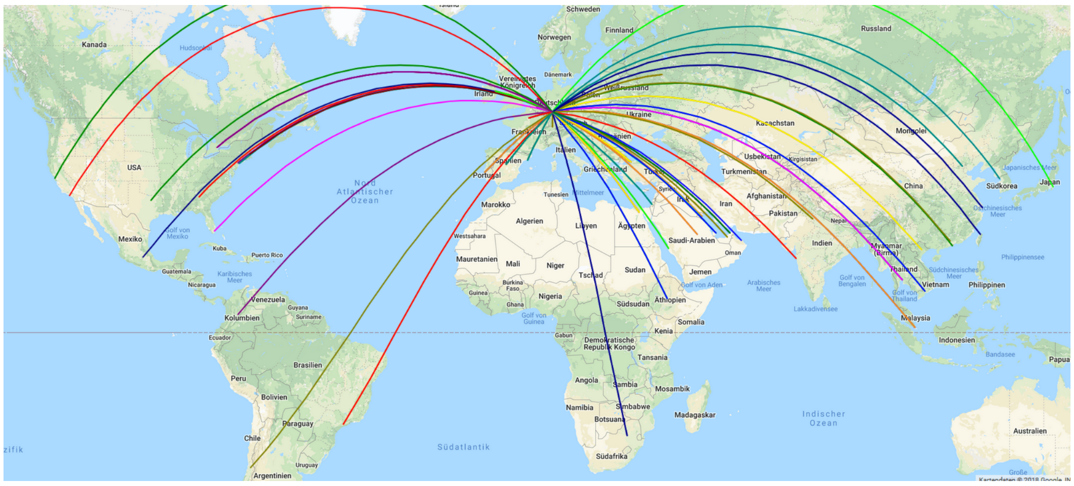


Figure 3: Outbound connections to Hotspots of Frankfurt.

flights to *Hotspots*. It is again possible to parallelize this step like described above. If again no flight was found to add to a path, there is no connection between the origin and destination airport and the algorithm terminates.

If at least one connection to a *Hotspot* was found the algorithm turns over to the next step otherwise this loop is repeated until a *Hotspot* is found. It also can make sense to limit the maximum number of loops allowed. This prevents that the algorithm expands a path again and again, because no path to a *Hotspot* is available but many other connections are. Even so that this case is really unlikely. For the special case that the algorithm finds a flight to the destination airport, this flight is added to the previous path and the connection is returned as the result. The algorithm terminates afterwards. If a connection to a departure *Hotspot* is found the algorithm overturns the next step and directly jumps to the last step (Connecting *Hotspot* and destination airport).

In the case of the ongoing example the closest airport to Hanover, which is *Hanover Airport*, is searched. Then all outbound connections of *Hanover Airport* to a *Hotspot Airport* within one day after the transit time are queried from the database. Figure 2b shows that ten connections to *Hotspots* was found. In the case that no connection to a *Hotspot* would be available, all outbound connections from Hanover would be queried and in the next loop, *Hotspot* connections from this places would be searched. This is done until a connection to a *Hotspot* was found. But in the case of Hanover there are flights to *Hotspots* available. Therefore the next step is applied.

Connection between Hotspots To find a path from the origin *Hotspot* to the destination *Hotspot* the same algorithm is applied as for finding a path from the origin airport to the first *Hotspot* as described above. This step is finished as soon as at least one path has been found to one of the destination *Hotspots* or even the destination airport. All other paths are discarded now. The only possibility to keep several paths in the path list is, that several destination *Hotspots* are reached in the same cycle of the loop. In this step only connections to other *Hotspots* should be considered by expanding the paths. But for the very unlikely case, that no connection between the origin and destination *Hotspot* is available by using *Hotspot* connections only, other airports could be considered as well.

One of the paths in the example above was arriving to Frankfurt. Even so the other paths are expanded as well this example concentrates on this path. All outbound connections to *Hotspots* from Frankfurt are queried from the database. The result can be seen in Figure 3. One of the connections is a flight to Hong Kong, arriving at 6:50 am on Saturday morning. Beside others, this flight is added to the path from Hanover to Frankfurt. Because Hong Kong is one of the destination *Hotspots*, all paths that have not been reached a destination *Hotspot* yet, are terminated and the next step is applied to find the connection between Hong Kong and Haikou to finaly get a full connection between Hanover and Haikou. Otherwise the same step would be applied again.

Connecting Hotspot and Destination Each time when a path reaches a destination *Hotspot*, it is checked whether it is the final destination airport already or not. If it is the final destination airport the way from the origin place to the origin airport and the way from the destination airport to the destination place have to be added. Otherwise a path from the destination *Hotspot* to the destination airport has to be found.

Previously to searching, the destination *Hotspots* a connection list were generated with an entry for each airport visited with all its outbound flights. This connection list was used to cache the flights from the destination *Hotspot* to the destination airport. This cache can be used now instead of querying the database again. This step needs again a new list for the current paths, which has been initialized by the path that reached the destination *Hotspot*.

```
while(destination not found)
  $\forall x \in$ pathlist
    $\forall y \in$ connectionList(destination of $x$)
      if(destination of $y \notin airportsOf(x)$ and bestChoise(y))
        pathlist.add(x.copy().add(y))
```

Code Snippet 3: Pseudo-code for connecting *Hotspot* and destination.

While x and y are connections, bestChoice() is a metric that returns whether x is the best flight, of a set of flights from the same origin and to the same destination. airportsOf(x) is the set of all airports on the connection of x. The connection list contains all outbound flights to an airport cached by the first step (Hotspot to destination airport).

The first line of this algorithm applies the algorithm as long as no path to the destination airport is found. The second line takes all open paths that was not terminated already. The next line takes all outbound flights which depart within one hour after the arrival and 24 hours afterwards stored in the connection list to the airport on which the path from the previous line are ending. Line four checks whether adding a connection from the connection list would result in adding an airport that is already on the path to prevent cycles and whether there are better connections on the connection list between the same origin and destination airports. If both criteria do not apply line five copies the existing path and adds the new connection to the copied path. Line two as well as line three can be parallelized.

For the previous example, beside other all outbound connections from Hong Kong within one hour after the arrival date and 24 hours afterwards, cached by the connection list are considered to find a connection to the destination airport - Haikou. Previously two connections from Hong Kong to Haikou were found. One on Friday 2:15 pm and another one on Saturday 2:10 pm. Since the flight on Friday departs before the arrival of the flight from Frankfurt, this flight is not considered. The flight on Saturday at 2:15 pm is added to the path.

After reaching the destination airport, the connections from the origin to the origin airport and the connection from the destination airport to the final destination have to be queried from the *Google Maps Direction API*, to finally adding them to the path.

5.3. Modified Recursive and Parallel Breadth First Search

This subsection explaines the *MBFS* algorithm n detail before the next subsection introduces a possibility to use the result of the *Hotspot Search* as input for this algorithm to obtain an upper bound. The basic intention of a *Breadth First Search* is to span a tree for finding the shortest path in an unweighted graph with bidirectional edges [KS05]. The environment for the Flight Graph is almost the opposite of that. The edges are unidirectional, weighted by multiple weights and to add an edge to a path certain constraints have to be considered. Also the intention of this algorithm is not to find the path with at least nodes as possible but find the best path by considering multiple criteria. hence, the tree condition of a traditional *Breadth First Search* is violated. Nevertheless, features from the traditional *Breadth First Search* are the base of the algorithm, used to find the best path for a connection in this thesis. This section introduces the *Modified Recursive and Parallel Breadth First Search* (MBFS) algorithm and points out the differences to the traditional *Breadth First Search* and other implementations of parallel *Breadth First Searches*. To illustrate the mechanisms an ongoing example will be employed.

Before describing the algorithm in detail the communication between the different threads which is required by this algorithm is introduced. Since each path runs on a separate thread it is necessary to coordinate all these threads. That could be done by a superior authority such as an object which is known by all threads. To ensure communication between the different threads this superior authority provides shared memory and functions. It includes a destination airport list with all destination airports to query whether an airport is a destination airport or not. The destination airport list can be accessed by several threads at the same time since only read accesses are required. Also the best connections that have already reached the destination need to be stored by the superior authority. Preferable this has to be done in any kind of sorted list, ordered by their virtual costs. The list capacity has to be equal to the number of connections which should be returned in the end. To ensure thread safety, this list requires locks for avoiding race conditions between different threads which write on the list at the same

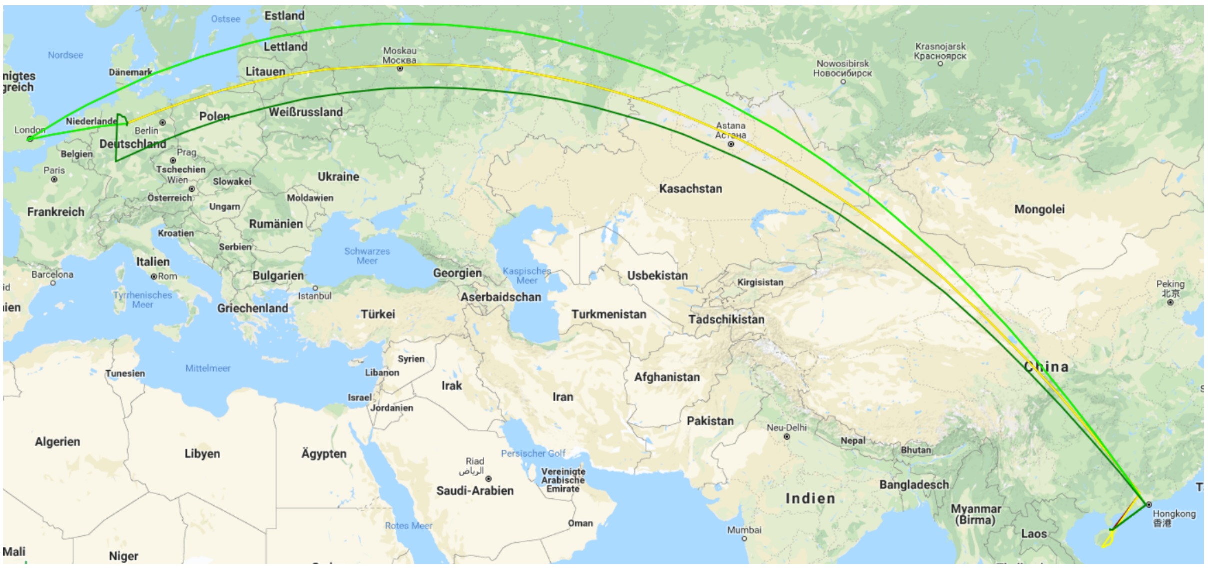


Figure 4: Full result of applying the MBFS algorithm for a connection between Hanover and Haikou.

time. Additional for each airport that has been visited by any path of the algorithm the beeline distance to the destination has to be stored as well as a list with all inbound paths that reached the considered airport already. To increase the runtime of the algorithm the inbound paths that should be ordered by their arrival time to this airport. The distance can be calculated once when an airport is visited for Abgabnthe first time. Afterwards, this value has to be read only. Therefore it is thread safe anyway. But the list of all inbound paths for an airport has to be updated continuously since new paths can reach this airport, thus blocking is required for this list.

5.3.1. Origin and Departure Airports

The first step of the MBFS algorithm is to find the closest airports from the origin place. Depending on the user input this can be the closest airport or up to the 25 closest airports. The closest airports could be measured by the distance or by the duration. Since the distance would affect the algorithm just by the fact that each driven km costs a certain amount of money, this thesis will always make use of the duration measurement method because this measurement shortens the duration of the full path as much as possible. The next step determines the closest airports to the destination with the same method as described above. For each origin airport a connection from the origin place to the airport is generated. All destination airports are added to a hashset to allow quick access to them later on.

The example used for the *Hotspot search* is used again for this algorithm. Remembering the connection from Hanover, a town in the northern part of Germany to Haikou, a town on Hainan, an island in the south of China which is a famous Chinese holiday destination. This connection between two small airports ensures that no direct connection is available. Figure 4 shows the full result of this request. Because of illustration purposes only the five closest airports to the origin and destination place are considered.

The first steps of the algorithm is to find the five closest airports to the origin and destination place. The results are shown in Table 2 and Table 3. Since the duration to the airports differs at different times because of changing traffic, the airports are ordered by the distance and not by the duration in the list. Due to the significant differences of the distance between the single airports usually this list should have the same order being ordered by the duration.

Afterwards, the recursive part of the algorithm is called in parallel for each of the previously generated incomplete paths. Remembering that the nodes of the graph are the places, which are usually airports and the edges of the graph are the connections between them, which are usually flights. A common parallel *Breadth First Search* algorithm would use a queue or a related data structure for the nodes that have to be expanded in order to guarantee that the nodes are expanded in the right order [KS05] [BM06] [LS10]. Approaches with other data structures exist, but they are still willingly to achieve the same goal by guaranteeing the expanding order of the nodes [YCH$^+$05]. Since it is possible that a node is visited several times, if certain criteria are applied in the approach of the algorithm of this thesis, there is no need to guarantee the right order of expanding the nodes. This can happen for example if a direct connection between Hannover and London exists which is quite expensive and a cheaper connection exists with a stopover in Amsterdam which is arriving in London later than the direct connection. In this case both connections are considered by the algorithm even though the airport of London was used by several paths. Therefore, it is absolutely fine if some paths are running ahead, while other paths are still expanding on lower levels. This allows the combination of a recursive as well as parallel algorithm. Why this property of the *Breadth First Search* can be violated in certain cases and how to monitor this in parallel is described in this section later on. The recursive part of the algorithm contains the following steps which will be described in detail in following.

- Check termination criteria.
- Get rewarding outbound connections of current airport.
- Expand the new connections.

For the previously introduced example, this recursion will be applied for the connections between the origin place and each origin airport.

Distance (km)	Airport
10	Hanover (HAJ)
98	Bremen (BRE)
110	Cassel (KSF)
113	Paderborn (PAD)
141	Hamburg (HAM)

Table 2: The five closest airports to Hanover.

Distance (km)	Airport
30	Haikou (HAK)
103	Bo’ao (BAR)
139	Zhanjiang (ZHA)
191	Beihai (BHY)
210	Sanya (SYX)

Table 3: The five closest airports to Haikou.

5.3.2. Termination Criteria

For each node that is visited by the MBFS algorithm, its chance to reach its destination in a proper time-price ratio is checked. If this is already impossible or at least very unlikely this path will not be expanded anymore and thus terminate. Otherwise the path is expanded as described in the next subsections. The algorithm terminates if all paths are terminated. Now, the criteria for terminating and expanding are described. The following criteria are processed in the same order as described in this section.

Airport has already been reached by better connections This criterion should eliminate all paths to a place for which a better path exists already. A connection is better in terms of this criterion if the connection is cheaper than all existing connections to the same airport which are arriving earlier or if the connection arrives earlier than all other connections to this airport.

$$x \in X : \nexists(y \in Y | y.arrivalTime < x.arrivalTime) : y.price < x.price$$

For X as the set of all new connections to this place and Y as the set of all paths which have visited this place already and were expanded. Only better connections are expand, other paths are terminating. This criterion is based on the idea of the Dijkstra algorithm [MS08].

This means if the algorithm has the choice between two different paths to the same airport while both paths arrive at different times, the algorithm will always choose the earlier flight, in case it is the cheaper of both one flights. By catching this flight it would be possible to catch all connected flights that would be possible to catch with the later flight as well but this flight would save also money. If the second connection had the cheaper price the algorithm would consider both paths. In this case it could be possible that a very cheap connecting flight exists that can only be caught by using the first connection and for which it is worth to pay even more money for the first part of the connection. But if a later connection would be the better choice which can be caught by the later connection to this airport, the later path would be the better choice. Therefore, in this case both paths are expanded. It is also possible that using an earlier connection is profitable after few further steps. Therefore, it is not enough just to expand the first or cheapest connection but all connections that are cheaper than all previous ones. Figure 5 illustrates this procedure.

This criterion requires a list which contains one entry for each airport that was visited by the algorithm. For each airport this list contains a second list that has one entry for each path which had visited this airport and was expanded. This path list contains the arrival time and the price for the connection to this airport. The path list is in ascending order by the arrival date. Due to the constraints of this criterion this implies that the path list is in descending order by the price as well. Consequently it is sufficient to request the price for the last connection arriving before the considered connection to decide whether the path should be expanded or not. In that case that the price of the considered connection is cheaper as the price of the earlier connection, this connection has to be expanded. That request is done in $\mathcal{O}(1)$. But the request needs locks to ensure thread safety. Otherwise race conditions between reading from the path list and writing

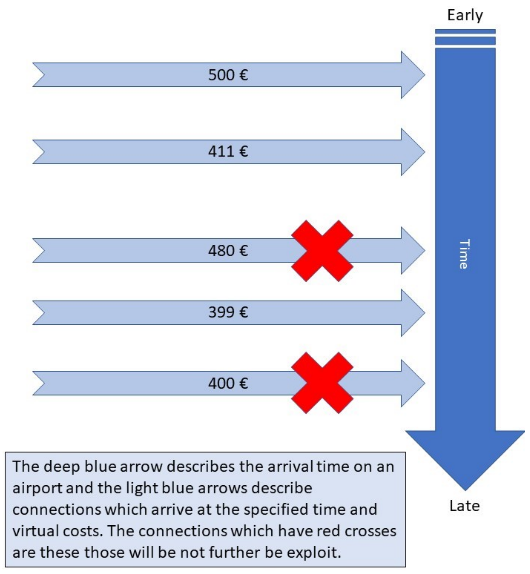

Early
500 €
411 €
480 €
399 €
400 €
Time
Late
The deep blue arrow describes the arrival time on an airport and the light blue arrows describe connections which arrive at the specified time and virtual costs. The connections which have red crosses are these those will be not further be exploit.

Figure 5: Illustrates which flights are terminating because better flights have reached the same airport already.

to it would occur.

Consider again the example above for the connection from Hanover to Haikou. After all five airports from Table 2 were expanded the first time, several flights are reaching Frankfurt. I.a. one flight from Hanover and one from Bremen. The required duration to reach the airport of Hanover from Hanover center is 24 minutes. Expecting that the user specifies that saving one hour of the trip time is worth 47 Euro and one km driving by car costs 10 cents (which is approximately the price for the fuel), the virtual costs to get to the airport of Hanover are 20.85. The required time to drive from the center of Hanover to the airport of Bremen is 1:27 (at the rush hour in this case), this leads to a virtual cost of 108.42. At both airports a transit time of one hour is required which leads to additional virtual costs of 47. The flight from Bremen to Frankfurt costs 62.70 Euro and requires 1 hour which leads to virtual costs of 109.70. The virtual costs for the whole connection are 177.55 now. The flight arrives in Frankfurt at 09:25 am. The flight from Hanover to Frankfurt costs 127.03 Euro and requires 55 minutes. That leads to virtual costs of 170.11. The virtual costs for the whole connection are accordingly 325.53. The arrival time of this connection at Frankfurt is 11:25 am. Since this connection arrives later at Frankfurt than the connection coming from Bremen and is also more expensive regarding the virtual costs, the connection coming from Hanover terminates on this point even so it is faster than the other connection. The connection coming from Bremen would be expanded. The same consideration apply for the following steps.

One of the flights that can be caught after the flight from Bremen to Frankfurt is a flight to Moscow. After two hours of transit, this flight departs at 11:25 am from Frankfurt and needs 3 hours to arrive in Moscow at 03:25 pm (all times are in local time). This flight costs 110.10 Euro which leads to virtual costs of 522.65 for the whole connection. A direct flight from Hanover to Moscow is also available. This flight requires 2:45 hour and arrives at 05:45 pm in Moscow. The costs for this flight are 134.82 Euro. That leads to virtual costs of 378.92 for the whole connection including transfer to the airport and transit times. In this case no connection would terminate, even though that the connection via Frankfurt is much more expensive: Because this connection is arriving earlier in Moscow and could catch a flight that can not be caught by the direct flight coming from Hanover both flights are kept.

If after evaluating this criterion the considered path has to be terminated, this is done immediately and all the next steps are not executed anymore. But if the path has to be expanded after this criterion the next steps also have to be checked.

Airport is destination airport If the airport reached in some step is one of the destination airports obtained in the first step of this algorithm, the connection from the airport to the destination requested from the *Google Maps Direction API* is added to the path. Afterwards the full connection is added to the list of connections that have reached the destination. This list is implemented as a list in ascending order by the virtual costs. The list can hold up to that number connections as the user input was pointing out as the maximal number of connections. If the list is already full and another connection is added, the list drops the connection with the highest virtual costs. This could also be the newly added connection. Since all destination airports are in a hashset, it can be checked in $\mathcal{O}(1)$ whether an airport is a destination airport or not. But this step again needs locks to ensure thread safety. Otherwise race conditions between reading the list of connections to check whether the new connection has to be added or not and writing on it could arise. After evaluating all following termination criteria the connection will be marked as destination found but not automatically terminate immediately as it will be explained in detail in the next termination criterion.

The last airport was a destination airport If a destination airport was found, it is not necessarily the best destination airport. Therefore the algorithm has the chance to reach a better destination airport in one more step. Especially for an increasing number of origin and destination airports chosen by the user, the probability to find an additional connection with a better departure airport arises. Due to the order of processing the termination criteria it is checked whether the airport is a destination airport before checking if the last airport was a destination airport as well.

For example in case Haikou is the destination place and the airport of Haikou as well as the airport of Hong Kong are in the list of destination airports. When a flight arrives to Hong Kong this is already a destination airport and for the remaining distance public means of transportation or a car could be used. The path could terminate then. But probably it is better to use one more flight from Hong Kong to Haikou in this case instead of driving whole the way. Therefor this criterion allows after reaching a destination airport to use one more flight. In this case a flight to Haikou can be caught and only a very short remaining distance has to be solved by public transport or car.

Maximum virtual costs reached The virtual costs describe the price-time ratio of the path. The benefit by using virtual costs instead of the price and the duration of a path is that the algorithm has to deal with only one weight instead of multiple weights. The virtual costs are determined by:

$$virtualCosts = price_{path} + duration_{path} \cdot x$$

While x is the price that is one hour worth for the user.

This criterion compares the virtual costs of the current path with the virtual costs of the best so far found connections. The maximal virtual costs are the virtual costs of the last connection (most expensive connection) of the list of the connections which have reached the destination already. If the virtual costs of the current path are higher than the maximal virtual costs the path has to terminate. Since the price as well as the duration are only able to grow and not to decrease, it is impossible that the path is added to the solution anymore, because the virtual costs are too high. This criterion only applies if the solution list has the maximum number of connections as defined by the user input. Otherwise also connections with higher virtual costs than the maximal virtual costs will be added to the list and therefore these paths will be expanded as well.

The distance to the destination in relation to the required time is too high With the previous criterion it is possible that a path uses a lot of very cheap and short discount connections, probably all in the same area, without coming closer to the destination.

For example when a path from Hanover to New York is searched. Assume the cheapest flight from Europe to any US airport costs 300 Euro. A lot of connections in Europe are available for 20 Euro. Or even cheaper. But assuming for this example, it exists a path which is always able to find flights between European cities for 20 Euro, this path could use 15 single flights and would still be in Europe, before the price of the flight would increase to the price of the flight between Europe and the US.

A possible solution to prevent this scenario would be to use a mechanism that ensures that each newly added flight to a connection decreases the remaining distance to the destination. The problem of this mechanism is that sometimes it is better or even indispensable to catch a flight which goes further away from the destination first, to catch a significantly better flight heading to the destination afterwards.

For example a connection from Saint Johns which (Newfoundland/Canada) to Frankfurt or any other place in Europe. The airport of Saint Johns provides only flights to the Canadian mainland and is the most eastern city of Canada and even the whole American continent except Greenland. The only possibility to reach Europe is first to move further away from the destination by catching a flight to the Canadian mainland and to be able to catch a flight to Europe afterwards. Section 7 will deal with detailed examples of connections between Africa and South America. For these connections this problem occurs really often. But also in Europe this phenomenon can be observed. For example by booking a flight from Germany to any Asian place and using a connection via Amsterdam.

Instead of measuring the remaining geographical distance to the destination, the already completed distance can be measured and set in ratio to the elapsed time since the departure. The already completed distance is measured by subtracting the geographical distance of the current place to the geographical destination from the distance of the origin place to the destination:

$$distance_{completed} = distance_{full} - distance_{remaining}$$

While $distance_{full}$ is the geographical distance between the origin and destination place and $distance_{remaining}$ is the remaining geographical distance from the current place to the destination. By defining a minimum average speed of the connection (including the transit times) a function $s \cdot x$ while s is the average speed, can be defined to determine the minimum completed distance at any point of time x where x is measured in hours. To allow that a connection is starting heading the opposite direction as the direction of the destination, the function is extended by a constant d that specifies the distance that the connection is allowed to further get away from the destination at the time zero. For $d = distance_{full}$ the equation looks as follow:

$$y > 2sx - d$$

While y is the already completed distance. This equation specifies all connections which are within the range of allowed completed distances.

By using this function a path has to head pretty fast to the destination. But on some connections it is essential to head to the wrong direction for a longer time first, to catch a flight that directly heads to the destination afterwards. Section 7 demonstrates this with connections between Africa and South America, which are usually connected via Europe. These connections need a long time before coming closer to the destination. Therefore a connection needs to be able to spend more time in the negative area in the beginning, which it can catch up on later through a good intercontinental flight for example. This problem is solved by using a quadratic function instead of a linear function. To ensure that the average speed still applies and the function scales well for short-distance connections as well as for long-distance connections the pure speed as the factor is replaces by a which will be described in following.

$$y > ax^2 - d$$

The factor a has to be chosen in a way that guarantees that the equation fulfills the average speed by reaching the destination. This is done when the quadratic function intersects the secant describing the average speed at the maximal time the connection needs by fulfilling the average speed. a has to be chosen as follows:

$$\begin{aligned} ax^2 - d &= sx \\ a &= \frac{sx + d}{x^2} \end{aligned}$$

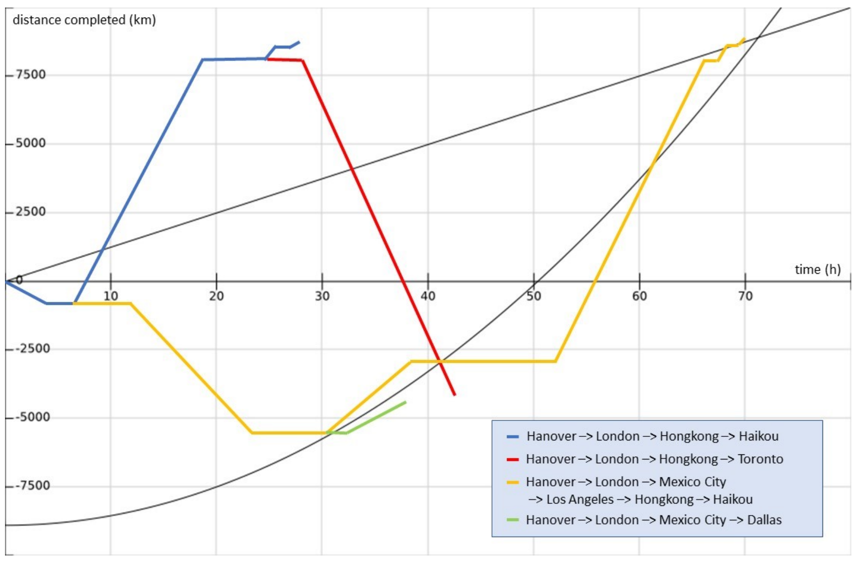


Figure 6: Some paths between Hanover and Haikou in relation to time and completed distance.

holds for x at the time that the connection would need, if it used exact by the average speed. That leads to $x = \frac{d}{s}$:

$$
\begin{aligned}
a &= \frac{s \cdot \frac{d}{s} + d}{(\frac{d}{s})^2} \\
&= \frac{2s^2}{d}
\end{aligned}
$$

The factor for the quadratic function has to be calculated just once for each run of the MBFS, since it depends only on the average speed and the full geographical distance between the origin and destination. The average speed is part of the user input and will be evaluated in Section 7. A path has to terminate if the upper quadratic equation for y is not fulfilled anymore. If the equation is fulfilled the next criterion will be checked.

Consider the previously introduced example about the connection between Hanover and Haikou to illustrate this termination criterion. The distance between Hanover and Haikou is 8,900 km. Using an average speed of 125 km/h does lead to the following calculation of a:

$$
a = \frac{2s^2}{d} = \frac{2 \cdot 125^2}{8900} = 3.51
$$

This example looks at four of many thousand paths that are checked to find the best connections and makes only use of this termination criterion. By applying the full algorithm these paths could be terminated by other criteria at an earlier point of time. Figure 6 illustrates this example. The black curve shows the minimum completed distance for any point of time when a flight arrives to an airport. The linear black function shows the completed distance for the average speed.

All considered paths are starting in Hanover and reaching London after 3:10 hours (blue path). At this moment the distance to the destination is 714 km longer as from the origin place. Different flights can be caught from London now. Next to other flights, one flight to Mexico City (yellow path) is caught and one flight heading to the destination and arriving in Hong Kong (blue path). The path to Mexico City will be dealt with later. First the path to Hong Kong will be considered. This path solved already 8,438 km of the full 8,900 km geographical distance in 19:15 hours. From Hong Kong a direct flight to Haikou can be caught after a transit time of 4:15 hours (blue path). After arriving in Haikou a car drive of 45 minutes is required to reach the destination. The full connection requires 26:40 hours, this is much faster than the maximal allowed time by average speed even so the first part of the connection was heading to the opposite direction as the destination place. Another connection that can be caught from Hong Kong is a flight to Toronto which arrives there 42:30 hours after starting the trip (red path). The distance from Toronto to Haikou is 12,859 km, accordingly 3,959 km more than from the origin place. Since $3.51 \cdot 42.5^2 - 8900 = -2560$ is the minimum completed distance after 24:30 hours and the completed distance from Toronto is -3,959 this path is terminated at that point. The values for the distance are negative in this case because the remaining distance to the destination at this point is longer than the distance from the origin place to the destination.

Back to the path from Hanover via London to Mexico City arriving after 23:20 hours (yellow path). Even so it looks like this path is totally out of the direction, which is also underpinned by the geographical distance from Mexico City to Haikou of 14,590 km, this path will not be terminated. The minimum completed distance after 23:20 hours is -6,989 km and the completed distance from Mexico City is -5,690 km, therefore this path still will be expanded. After 11:00 hours of transit a flight to Dallas could be caught (green path). This flight would arrive in Dallas after a total time of the trip of 37:15 hours. The completed distance at Dallas is -4,550 km. Since the minimum completed distance after 37:15 hours is -4,055 km this path will terminate at that point.

Another flight that can be caught from Mexico City is a flight to Los Angeles arriving after 36:50 hours of travel (yellow path). As the minimum completed distance after this time is -4,138 and the completed distance from Los Angeles is -3,210 this path will be expanded again. Now a flight to Guangzhou, arriving after 66:00 hours is available. Remembering that only the arrival time has to be in this function. The time of departure does not matter for this criterion. In Guangzhou 8,400 km are completed already and after one more hour a flight to Haikou departs. After a duration of 70:05 hours the final destination is reached within the maximum time interval defined by this criterion. Even so, this connection has done a huge detour and was always close to the termination it was able to reach the destination in the end. But this criterion does not consider the price of the connection. By applying all criteria it would be possible that the yellow path would have been terminated by another criterion.

A short example of the connection between Hamburg and Rome shows the scalability of the algorithm even for short-distance connections. Figure 7 illustrates this example. A direct con-

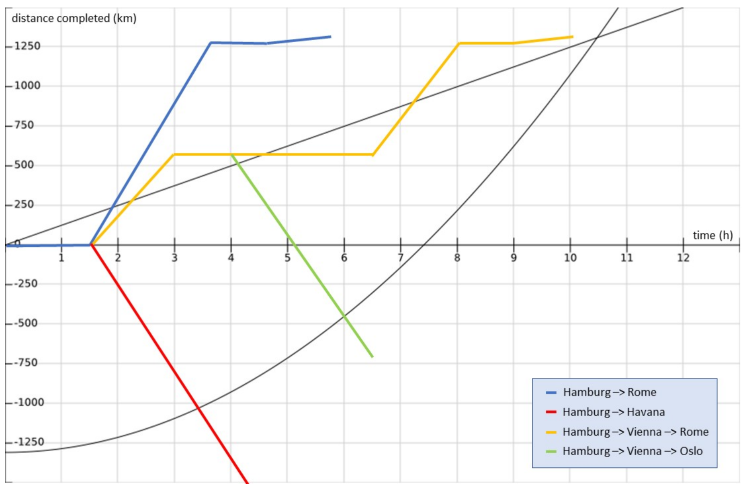


Figure 7: Some paths between Hamburg and Rome in relation to time and completed distance.

nection (blue path) requires about 6 hours, including transfer from and to the airports and transit times. A connection via Vienna (yellow path) which is quite a detour but still heads to the right direction is possible as well. Also a connection via Copenhagen, London or even Dubai, which would start heading to the opposite direction as the destination is possible, if the transit times are not to long. However, a flight that heads to a destination far away like Havana (red path) terminates immediately. Also a connection which flies too much zigzag, as for example the connection from Hamburg via Vienna to Oslo (green path) will be terminated quickly. Such a start of a connection could be acceptable for a long-distance flight but not for short distances.

In the beginning there were approaches to use the ratio of the number of flights of a connection and the proportional completed distance. But it turned out that this approach was not scalable for short- and long-distance connections.

To many steps This criterion terminates a path after a certain number of algorithmic steps. An algorithmic step is each call of the recursion to find the path to the destination place. This prevents paths with a huge number of very cheap and short flights and is a security guard that guarantees that no endless loops can be applied.

5.3.3. Receive Outbound Connections

To expand a path the database is queried for all outbound connections of the according airport. Requesting all outbound connections for one airport instead of requesting each single connection saves runtime since the operations are very fast on the database level. At this point it is already possible to reject some flights. Therefore only the best flights to each airport will be chosen and cyclic connections are removed. Cyclic connections are connections that use the same airport several times.

Recall the example of the connection from Hanover to Haikou. There exists a flight from

Hanover to London. When expanding this path the next time, the return flight from London to Hanover will show up. For a human brain it is obvious that it makes no sense to take this flight afterwards. Since flights never can cost a negative amount of money and it can be expected that a flight is also not offered for free, it would be cheaper just to wait in Hanover instead of flying to London and returning. For a search algorithm this behavior has to be considered. Another flight that can be caught from London is a flight to Moscow. In Moscow a flight back to Hanover is available again. Also taking this flight would end up in flying a cycle.

To prevent this behavior for each flight that should be added to a path, it is checked whether the destination airport is on the path already. In this case, this flight will not be added to the path. Now, adding all remaining flights would result in generating a lot of connections that will terminate after the next step. Therefore all flights without a chance to survive will be removed as well. One solution would be to add just the cheapest connection to each airport or adding for each destination airport the connection with the earliest departure time. But both approaches can eliminate the best connections as shown in the following examples.

Consider two flights between Hanover and London and two flights between London and Hong Kong. The first flight from Hanover to London costs 100 Euro and arrives at 1 pm and the second one costs 50 Euro and arrives at 3 pm. The first flight from London to Hong Kong costs 500 Euro and departs at 2 pm while the second one costs just 300 Euro and departs at 3 pm. If always the first flight would be chosen, in both cases the more expensive flight would be chosen without any time benefit. Now assume that the first flight to London costs 100 Euro and the second one costs 50 Euro. The cheapest connection would be to choose the more expensive flight first to be able to catch the cheaper one afterwards. But if always only the cheapest flight is chosen, the algorithm would catch the second flight to London and would not even be able to catch the cheaper flight to Hong Kong because this flight was departing before the flight from Hanover was arriving.

Section 5.3.2 introduced a mechanism that decides whether a connection to an airport should be terminated or not, in the case that other connections were reaching the same airport already. The same procedure can be applied for this case as well. For each airport a flight will be added to the connection if all previous flights to the same airport are more expensive than this flight or if there is no earlier flight. Accordingly the first flight to each airport is added generally. Figure 5 illustrates this mechanism. In contrast to the cycle detection this can be done very fast, set oriented on the database level.

If the connection added in this step was the first flight on the path, the full connection from the origin place to the origin airport has to be obtained from the *Google Maps Direction API* and is added to the path. Because the connection to the airport depends on the departure time of the first flight this can not be done at an earlier point.

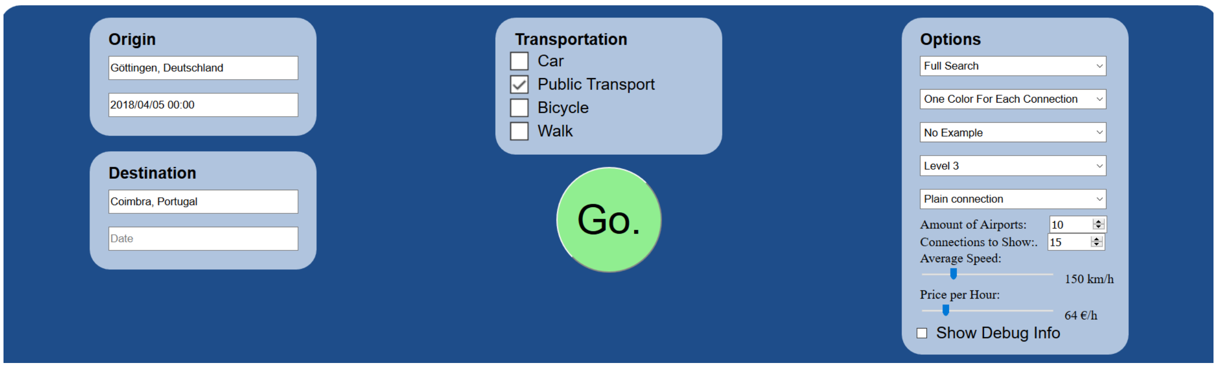
5.3.4. Expanding of New Connections

After all connections that should be used to expand the path are found, for each newly found connection the old connection has to be cloned and the new connection needs to be added to the cloned old connection. Afterwards the recursion can be called with each of this connections. This step can be done in parallel.

5.4. Combining both algorithms

To enhance the efficiency of the algorithm it is possible to combine both previous described algorithms, the *Hotspot Search* and the *Modified Recursive and Parallel Breadth First Search* (MBFS). In case of using just the *MBFS* some connections need to reach the destination, before connections can terminate because their virtual costs are higher than the virtual costs of the best connections.

When the *Hotspot Search* is applied before using the *MBFS* algorithm, the results of the *Hotspot Search* can be used as the input for the *MBFS* algorithm to have a valid path from the beginning, to apply the maximum virtual costs criterion from the first moment. While usually the *Hotspot* Search requires only a few seconds for processing, a huge runtime enhancement can be expected. The next section will examine this enhancement in more detail.


Origin

Göttingen, Deutschland

2018/04/05 00:00

Destination

Coimbra, Portugal

Date

Transportation

☐ Car

✓ Public Transport

☐ Bicycle

☐ Walk

Options

Full Search

One Color For Each Connection

No Example

Level 3

Plain connection

Amount of Airports: 10

Connections to Show:. 15

Average Speed: 150 km/h

Price per Hour: 64 €/h

☐ Show Debug Info

Figure 8: Input box of the *Trip Planner* user interface.

6. The Trip Planner System

To test the previously described algorithms the *Trip Planner* application was developed. The application generates the *Flight Graph* as described in Section 4, provides access to the different APIs as described in Section 3, offers the different path finding methods introduced in Section 5 and visualizes the result connections via a user interface. Before using this application in the next section for evaluation purposes, this section introduces the most important parts of the application.

The first part of this section concentrates on the description of the graphical user interface of the *Trip Planner* application. Afterwards the implementation of the application is roughly described. In the end the use of the different APIs introduced in Section 3 in the *Trip Planner* application is described.

6.1. Graphical User Interface

The user interface of the *Trip Planner* application is provided by the *Trip Planner Client* application. The interface allows the user to insert a query comfortably and to display the results in a nice and clear way. The functions of the client application are therefore limited to functions for validating the user input, generating a request for the server, connecting to the server and processing the response from the server application to display it for the user. The window of the user interface is split into three different parts. The first part is the input window where the user inputs the request, the second part is the map that visualizes the result and the last part is the text output that gives detailed information about the journey.

Input Window The input window, shown in Figure 8, allows the user to insert the origin and destination place. These fields dispose over a *Google* auto complete function which completes the place name as well as adding coordinates to it. This function ensures that a place used for the request is unique. Also the inbound and outbound departure dates and times can be inserted from a calendar with valid date and time values. The next box allows the user to choose which further means of transportation should be considered by calculating the path. Car, public transport, walk and bicycle are possible selections.

The last box provides further options about the calculation algorithm and how to display the results. The first option lets the user choose one of the following path finding methods:

- The *Google Distance* and *Google Direction* methods are just forwarding the request to the according *Google Maps* APIs and returning the results. The *Skyscanner* and *eStreaming* methods are querying the database for the closest airport (measured in beeline distance) to the origin and destination place and query the according API for a connection between those two airports afterwards. If no connection can be found between these airports an empty connection list is returned.

- The *Query Database* method returns all flight connections stored in the *Cached Flights Database* between the closest airport to the origin place and the closest airport to the destination place (measured in beeline distance). The *Query Database* method returns all flights regardless the date, the *Query Database (Time)* method considers the date and time and returns all flights within 24 hours after the given departure time. If no connection between these airports was found these methods return an empty connection object.

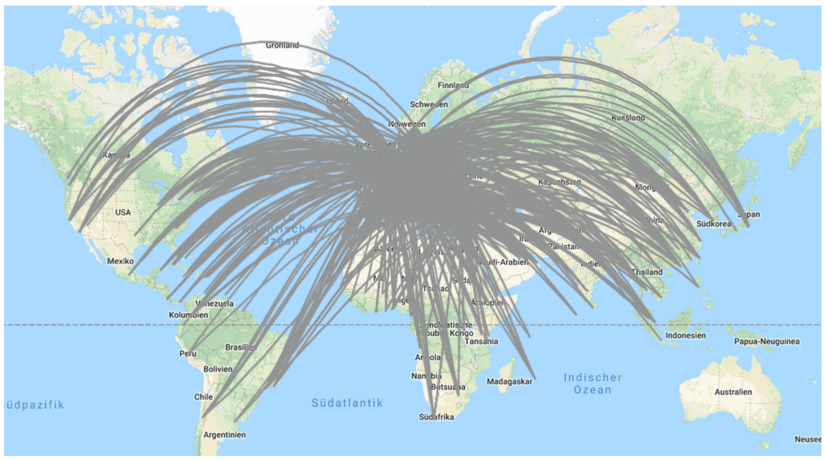
- The *Connected Airports* method returns all outbound connections of the closest airport to the origin place and all inbound connections of the closest airport to the destination place (measured in beeline distance), regardless the time. The *Connected Airports (Time)* limits the results to the outbound connections within 24 hours after the departure time and the inbound connection within five days after the departure time. The inbound connections are returned for a time period of 5 days instead of 24 hours because the *Hotspot Search* requires this time frame for inbound connections. The *Connected Hotspots (Time)* in turn limits these results to outbound connections to *Hotspots* and inbound connections from *Hotspots* stored in the *Cached Flights Database*.

- The *Hotspot Connection Search* starts searching a connection between the origin and destination place on the specified departure date by using the *Hotspot Search* algorithm introduced in Section 5.2.

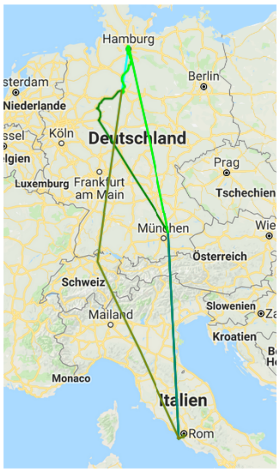
- The *MBFS* method uses the *Modified Recursive and Parallel Breadth First Search* algorithm introduced in Section 5.3 to find a path between the origin and destination place on the specified day by considering the given settings.

- The *Full Search* applies both, the *Hotspot Search* and the *MBFS* as described in Section 5.4. By applying the *Hotspot Search* before the *MBFS* algorithm.

With the next option the user can select, whether each sub connection should get a new color, or just each alternative connection should get a new color on the map. With the third option it is possible to use a predefined example. This allows the user to define an example once and use it afterwards all the time instead of inserting all input fields lot of times for test purposes. The level defines how detailed the lines and the text output should be. If the level is one there is just a line drawn between the origin and the destination place. If the level is four, it will be



(a)



(b)

Figure 9: The map on the *Trip Planner* user interface shows the used connection in colors (b) and the terminated paths in gray (a), for the connection between Hanover and Rome.

described how to proceed for each corner. The text output part of this section describes the different levels in detail. The next option selects whether only the result should be shown on the map and the text output (plain connection) or also all terminated paths, that were generated by calculating the connection (all connections). In this case all previously terminated paths are displayed in grey.

The next part of the input box allows to set some values that have impact on the procedure of the *MBFS* algorithm. These settings are required for the experimental evaluation in Section 7. Therefore the number of airports used as origin and destination airports can be specified. Also the maximum number of result can be chosen. Valid values for both fields are integers between one and 20. Moreover, the average speed can be chosen from a scale between 50 and 500 and the price a user would pay to save one hour of travel time can be inserted. This can be a value between zero and 1000. By selecting "Show Debug Info" the transmitted JSON string is printed as well. Pushing the green *Go.* button in the end requests the results.

Map The map displays the paths of all connections found for a request or even all connections used to calculate the path by the path finding algorithms depending on the settings. Figure 9 shows the result by using the *Full Search* algorithm to query a flight from Hanover to Rome. All paths that have been used for the calculation but that are not part of the result connections, are displayed in grey (Figure 9a). All other paths have different colors for each step or for each connection (Figure 9b), depending on the previous selections. If the user follows the path with the cursor, further information like the means of transportation, stops or the flight number of a flight occurs in an info window on the path.

Text Output The text output of the *Trip Planner* user interface provides detailed information about the trip and its single steps. The detail level of the output depends on the selected options in the input window.

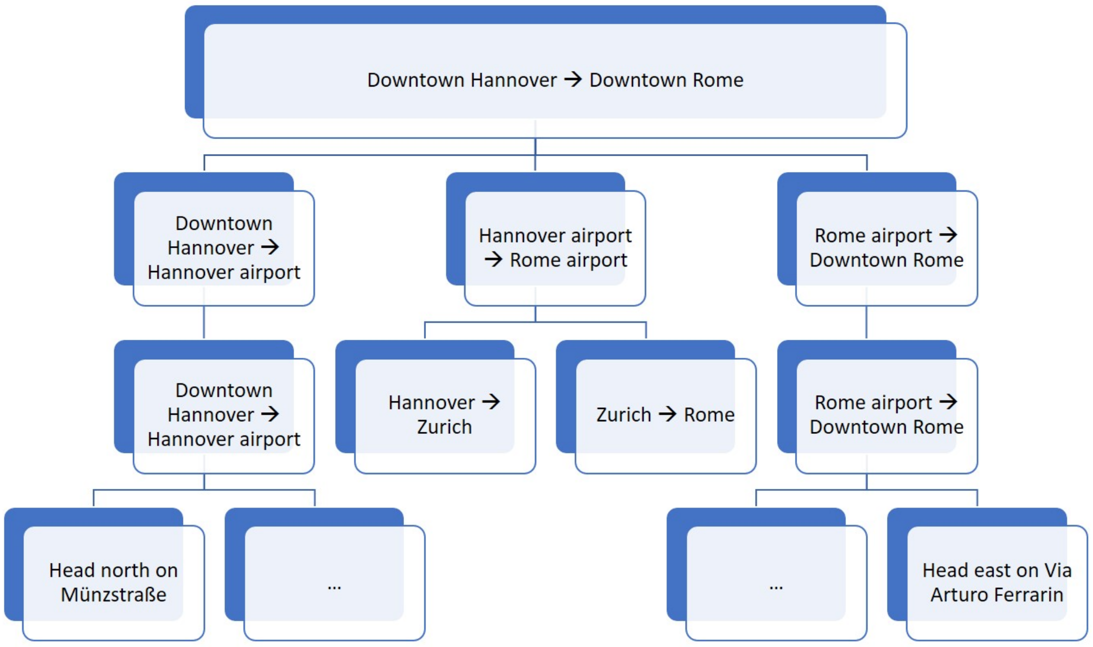
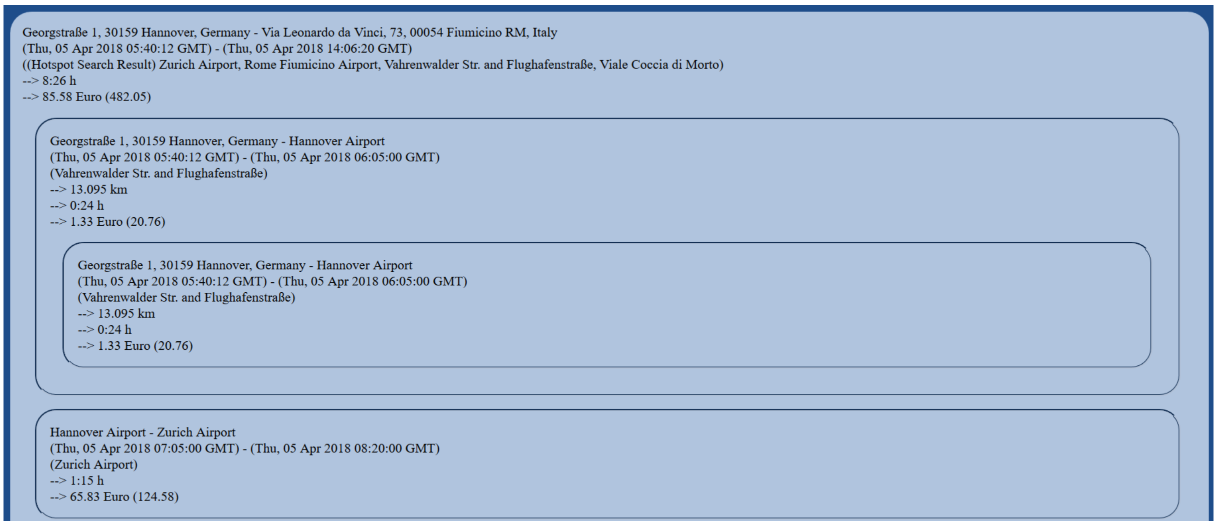

Downtown Hannover → Downtown Rome
Downtown Hannover → Hannover airport
Hannover airport → Rome airport
Rome airport → Downtown Rome
Downtown Hannover → Hannover airport
Hannover → Zurich
Zurich → Rome
Rome airport → Downtown Rome
Head north on Münzstraße
...
...
Head east on Via Arturo Ferrarin

Figure 10: The four levels of a connection.

Each connection is separated in up to four different levels. The highest level only includes the origin and destination of the whole connection. The second level splits the connection into different parts for each means of transportation. All means of ground public transportation are aggregated to one means of transportation in this case. The third level splits the means of transportation into one connection for each ride. This level splits a connected flight for example into several direct flights. The fourth level splits the connection into very small steps where each step has a detailed route description. This level is mainly used by *Google Maps Direction* for navigation.

For example the connection from Hanover to Rome by plane shown in Figure 10 would have just one step for the first level (downtown Hanover to downtown Rome). The second level would split this connection into three parts: The drive from the downtown Hanover to the airport of Hanover, the flight from Hanover to Rome and the drive from the airport of Rome to downtown Rome. The third level separates the flight connection into its single direct flights, in case the flight is not direct already. For example into the flights from Hanover to Zurich and from Zurich to Rome. The fourth level would separate the car connection to Hanover airport and from Rome airport into small steps with a detailed route description. The fourth level is introduced because details about the path of the flights are often required but a detailed route description of the drive would overload the response and would lead to an unclear text output of the journey.

The first line of the text output of each step shows the origin and destination place of this step. This can be an address or an airport name. The next line shows the departure and arrival dates.



```
Georgstraße 1, 30159 Hannover, Germany - Via Leonardo da Vinci, 73, 00054 Fiumicino RM, Italy
(Thu, 05 Apr 2018 05:40:12 GMT) - (Thu, 05 Apr 2018 14:06:20 GMT)
((Hotspot Search Result) Zurich Airport, Rome Fiumicino Airport, Vahrenwalder Str. and Flughafenstraße, Viale Coccia di Morto)
--> 8:26 h
--> 85.58 Euro (482.05)

    Georgstraße 1, 30159 Hannover, Germany - Hannover Airport
    (Thu, 05 Apr 2018 05:40:12 GMT) - (Thu, 05 Apr 2018 06:05:00 GMT)
    (Vahrenwalder Str. and Flughafenstraße)
    --> 13.095 km
    --> 0:24 h
    --> 1.33 Euro (20.76)

        Georgstraße 1, 30159 Hannover, Germany - Hannover Airport
        (Thu, 05 Apr 2018 05:40:12 GMT) - (Thu, 05 Apr 2018 06:05:00 GMT)
        (Vahrenwalder Str. and Flughafenstraße)
        --> 13.095 km
        --> 0:24 h
        --> 1.33 Euro (20.76)

    Hannover Airport - Zurich Airport
    (Thu, 05 Apr 2018 07:05:00 GMT) - (Thu, 05 Apr 2018 08:20:00 GMT)
    (Zurich Airport)
    --> 1:15 h
    --> 65.83 Euro (124.58)
```

Figure 11: A part of the text output of the connection between Hanover and Rome on the user interface of the *Trip Planner* application.

Usually a third line appears with further information about the connection like the intermediate stops. If this line starts with the term "Hotspot Search Result" this indicates that the connection was found by the *Hotspot Search*. Afterwards the travel distance of the connection is shown if available. This is followed by the duration. The last information is the price of the connection. The value behind the price in bracelets represents the virtual costs of this part. Each higher level summarizes the values of all levels below. Figure 11 shows a part of the text output of the connection from Hanover to Rome.

6.2. Implementation

In this section only the core features of the implementation are described very shortly. For a detailed description of the implementation the technical report [Hen18] can be consulted. First the basic structure of the different application parts are introduced and afterwards some core mechanisms of the implementation are described.

The *Trip Planner* application consists of two parts. One part is the server application which is written in *Java* and runs on an *Apache Server*. The second part is the client application which is mostly written in *JavaScript* and runs on the client's web browser. This part is mainly written in *Java Script* because it uses the *Google Maps Web API* which requires the client to be implemented in *JavaScript*. While the task of the client application is mainly to visualize the output and take over the graphical user interface part of the *Trip Planner* application, the server application takes over the main tasks of the application. The server application consists of three different parts. The first one is the access to the different APIs. The second one is to create the flight graph with cached flight connections to use them later on for the path finding. The third part is the flight path finding which consists of different path finding algorithms to get the best connections between two given places, which is the core task of the application.

The different path finding methods are modularly implemented in the *Trip Planner Server* application, therefore it is simple to change, add or remove a method. The path finding is implemented on the server side. A detailed description of how to add, change or remove methods can be found in the project report [Hen18]. Also the different APIs are implemented modularly, therefore it is easy to replace one API by another if the new API used the same parameters as the previous one.

Connections are implemented throughout a Connection object. This object contains fields for all means of transportation and is therefore applicable very flexibly. For each kind of connection in the whole *Trip Planner Server* and *Client* application this object is used. Also for paths, this object is used. In this case, the Connection object is used to describe an overview over the whole path and a list of sub connections describes the single parts of the connection in detail. This can be nested into a lot of levels. The paths are implemented throughout thread-safe queues that can contain several nested *Connection* objects.

6.3. API Access

APIs from three different companies are used in the *Trip Planer* application. The server application uses APIs from *Skyscanner*, *eStreaming* and *Google*, the client application uses *Google* APIs only. The first idea was that all APIs from the server application are accessible by a simple interface class that allows the access to all APIs by using the same attributes. For example an origin place, a destination place and a departure date. This would enable replacing one API by any other very easily. But during the development process it turned out that each API requires different input parameters and has very different possibilities. For example *Google Maps* distinguishes between different means of transportation [Goob] while *Skyscanner* just provides flight connections [Skya]. On the other hand *Skyscanner* and *eStreaming* require a given market place [Skya][Gona] while *Google Maps* just optionally allows to pass a language [Goob] but it is not possible to pass a response language to *eStreaming* [Gona]. That leads to the problem, that it is hard to simplify each API access to such a simple method call. Therefore, the classes for accessing the APIs have own specified methods to call the APIs and additionally there is one simplified interface for all APIs. But in the end this cannot be used for the most cases.

The *eStreaming* API access has some specials that should be explained on this point. Regardless that the same APIs are used, the *Trip Planer* application provides different functions that process the same received data in different ways. The simplest methods return only the cheapest direct or connected flight. Another method returns all direct connections and splits the connected flights into single flights and returns them as direct connections. In this case the direct connections obtained from the connected connections do not have a price. The last method returns all direct and connected flights. But the connected flights are returned twice. Once the whole connection including the single legs and once they are split up into the single flights as described above. This is required because a lot of direct connections are not stored in the *eStreaming* database as direct connections but as parts of connected flights. This method is used to generate the *Cached Flights Connection Database* described in Section 4.

7. Experimental Investigation

In the previous section, different flight path search algorithms with various settings were introduced. In this section these algorithms are investigated. The *MBFS* is investigated in terms of its performance and quality of different settings, while the *Hotspot Search* is basically investigated with regard to its quality and its runtime benefit in contrast to the *MBFS* algorithm. Therefore experiments with different settings are applied.

The first part of this section describes the different settings the user can perform to influence the algorithms. Afterwards the logging mechanisms of the algorithms for evaluation purpose are described. The next part introduces the experimental environment. Before applying the different experiments, the connections that are investigated in this section are introduced. For the experiments five different connections of different lengths are introduced. Each of the connections has its special characteristic, like an origin place without an airport or an airport with just one flight daily. The last part describes and investigates the experiments of this section. Seven different experiments are applied to investigate the impact of different settings on the runtime of the algorithms and the quality of the results.

7.1. Settings

The user is able to define several parameters which have impact on the behavior of the *MBFS* algorithm. The first is to choose the plain *MBFS* algorithm or the *Full Search* which performs the *Hotspot Search* before applying the *MBFS* to use the results from the *Hotspot Search* as input for the *MBFS* algorithm. The next parameter sets the number of origin and destination airports to use. The third parameter sets the number of connections for the result set. The fourth parameter sets the minimum average speed that a connection has to perform and the last parameter sets the amount of money that the user is willing to pay for each hour of travel time saved. Those are the parameters that are evaluated in this section.

7.2. Logging Mechanisms

The *Trip Planner* counts and logs different properties for the *MBFS* algorithm which can be used for evaluation purposes. Therefore for each expanding step it is measured for each termination criterion how many paths were terminated by it. Next to the termination criteria, for each expanding step it is measured how many connections have not been considered for expanding because they are not promising. This logging function distinguishes between a flight that was not considered because it would lead to a cycle in the connection and a flight that was not considered because other flights on the same route are better. Also the number of valid connections that were found in each expanding step and the number of open paths after each expanding step are measured. The total number of terminated paths and paths that reached the destination is the total number of generated paths. After processing the algorithm the *Trip Planner* application returns an array for all these logs.

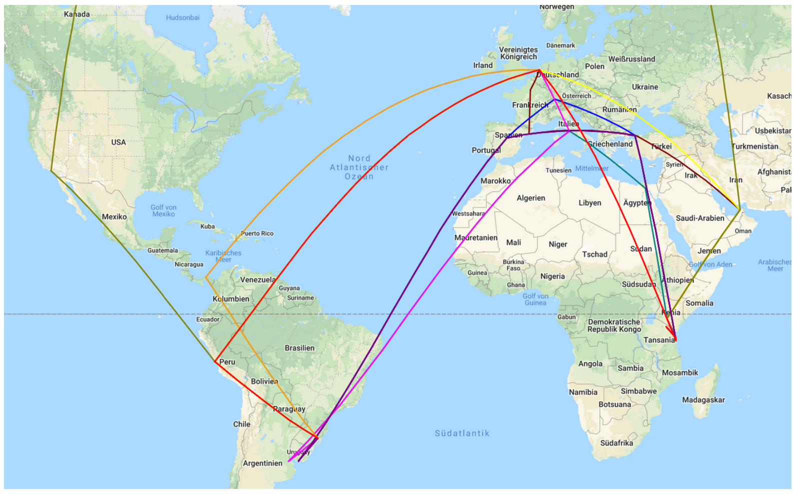


Figure 12: Connections between Arusha and Porto Alegre.

Furthermore, the total number of visited airports throughout the whole search process is measured as well as the number of *Google Maps API* calls required to get the connection from the origin place to the origin airport and from the destination airport to the destination place. Moreover, the pure runtime of both algorithms - the *Hotspot Search* and the *MBFS* - is measured individually.

7.3. Experimental Environment

The *Trip Planner Server* application and the *Trip Planner Client* application are running on the same machine: A *DELL* laptop computer with 8 GB DDR4 Ram and an Intel Core i7 processor in the mobile edition with up to 3.20 GHz on two physical and 4 virtual cores. The operating system running on the laptop is a 64 bit Windows 10. The database system used for the *Cached Flights Database* is *PostgreSQL 9*. As there does not run a pure research system on the laptop, other tasks were running on the same machine as well, even though these tasks were reduced to a minimum, it is possible that some runtimes are affected by them.

7.4. Evaluated Connections

To evaluate the algorithms on their behavior with the different settings introduced previously, different connections are used for the experiments. Among those connections are short and long-distance connections as well as connections between big airports and between very small airports. Therefore, the five connections between Göttingen and Coimbra, Cape Town and Asuncion, Arusha and Porto Alegre, Haikou and Balmaceda and Arusha and Balmaceda are investigated. This section introduces these connections and their purpose before they are investigated in detail in the next section.

7.4.1. Göttingen to Coimbra

Göttingen is a city in the center of Germany popular for its university, with a population of 119,000 inhabitants. Coimbra is a city in the center of Portugal with a population of 143,000 in-

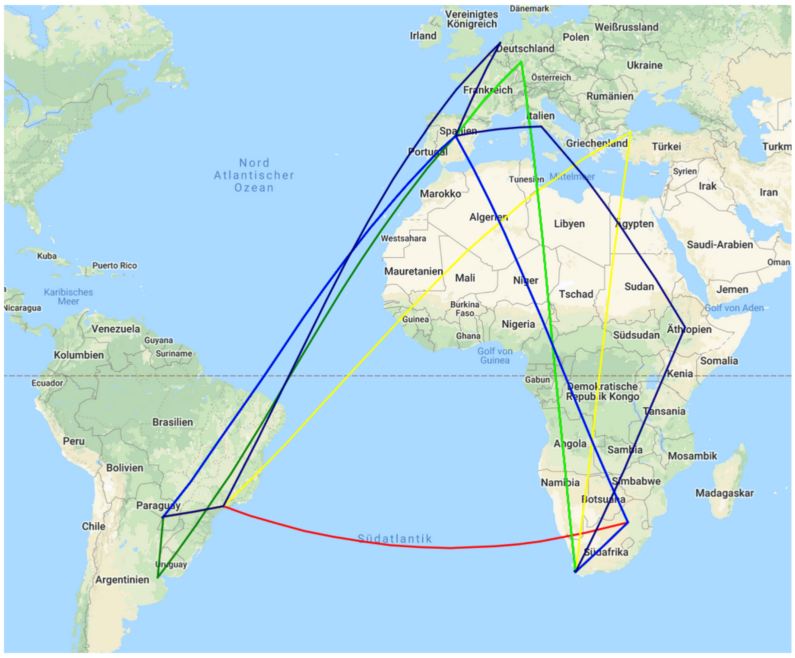


Figure 13: Connections between Cape Town and Asuncion.

habitants. Both cities do not have an airport, but they are served by high speed trains. Another similarity of both cities is that both are hosting very old traditional universities. Since both universities are members of the Coimbra Group it is realistic that people are traveling between both places.

Finding a connection between these cities shows how the algorithms are searching for the best origin and destination airports by using public transport to get to the airports. Furthermore this connection shows how the algorithms find short connections within a very closely meshed flight network as it is available in Europe. Figure 14 shows that the algorithm uses different origin and destination airports which are reached by train from the origin and destination cities. Hanover, Erfurt and Frankfurt are used as origin airports while Lisbon and Porto are used as destination airports. Afterwards a lot of different connections are available from each airport.

7.4.2. Cape Town to Asuncion

Cape Town is the second largest city of South Africa with a population of 433,000 people, but the metropolitan area of Cape Town has a population of 3.7 million people. Asuncion is the capital and the largest city of Paraguay. Its population is 525,000 people and in its metropolitan area live 2.2 million people.

A connection between these two cities would connect an African metropolitan region with a South American metropolitan region, but there is no direct connection available. As it can be seen in Figure 13 these cities are connected via a flight between Johannesburg and Sao Paulo, the Middle East or Europe.

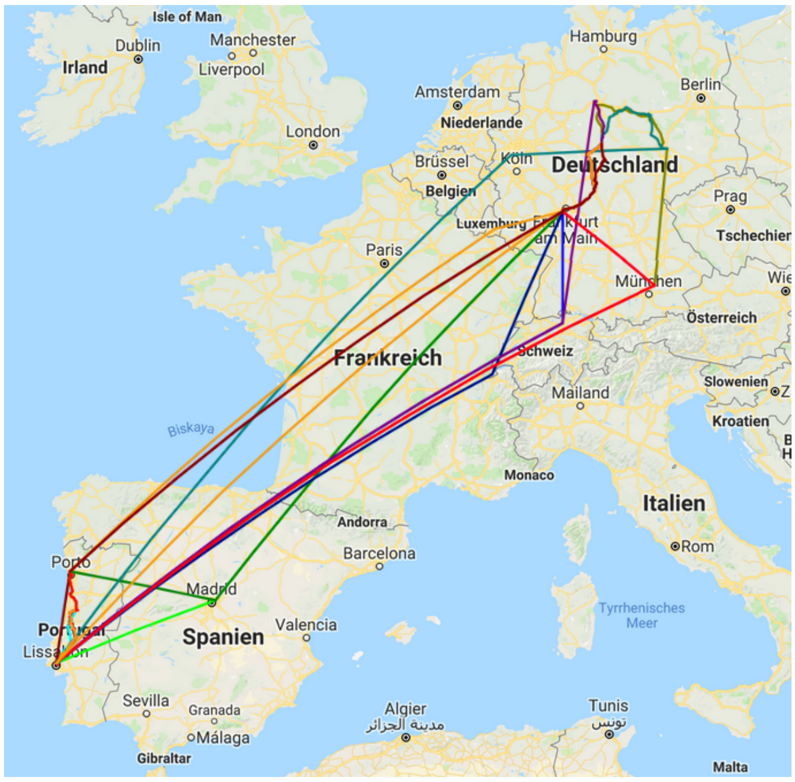


Figure 14: Connections between Göttingen and Coimbra.

7.4.3. Arusha to Porto Alegre

Arusha is an African town in the north of Tanzania with a population of 416,000 people. Arusha is located on the foot of the Mount Meru and close to the Kilimanjaro, Africa's highest mountain. Also the Serengeti National Park which is popular for its animal population is right next to the town. Therefore, Arusha is a popular tourist destination. Beside the tourism, Arusha also has an important international diplomatic impact since the East African Community's headquarter is located in Arusha. Porto Alegre is located on the coast of the southernmost part of Brazil. It is on rank 10 of the most populated cities of Brazil with 1.5 million inhabitants. Its metropolitan area has a population of 4.4 million inhabitants. Nevertheless, its airport provides only one inter continental flight a day, which goes to Lisbon. Figure 12 shows different connections between these cities, but in contrast to the connection between Cape Town and Asuncion it can be observed that no direct flight between Africa and South America is available for this connection. Therefore the connections are using flights via Europe or the Middle East.

7.4.4. Haikou to Balmaceda

Haikou is a town on an island in the south of China and Balmaceda is a small village with a population of 500 people, high in the mountains of Chile. Each day there is only one flight to Balmaceda coming from Puerto Montt which itself is a town on the coast of Chile with a population of about 240,000 people. Its airport provides several flights from Puerto Montt to Santiago each day.

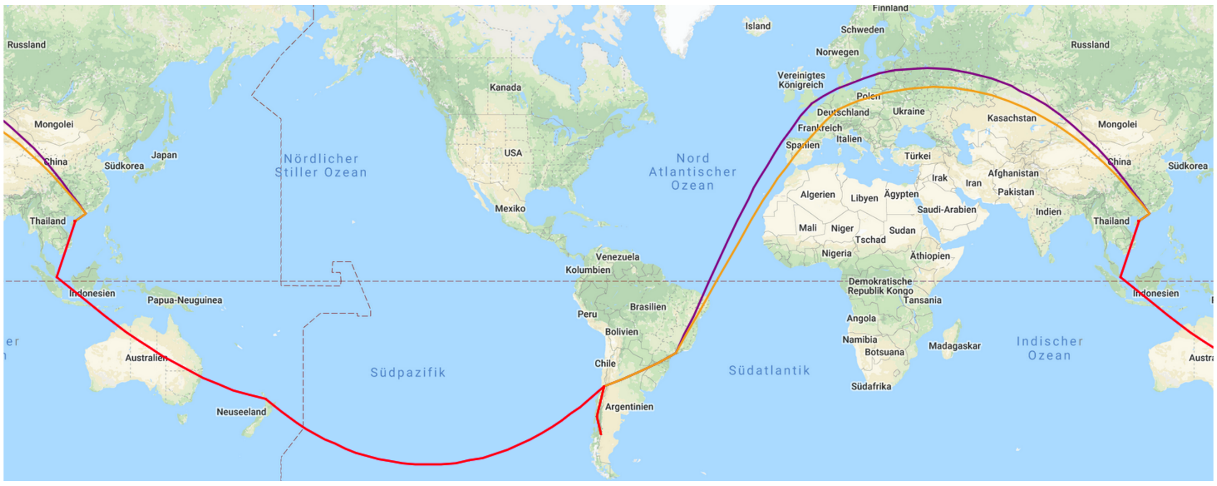


Figure 15: Connections between Balmaceda and Haikou.

A flight between these two cities connects an Asian island with the countryside of South America. Figure 15 shows that it is possible to head east around the world from China to South America via North America or Australia as well as using the west route around the world to South America via Europe. A special characteristic of this connection is that for the connection from Haikou to Balmaceda usually the path via Australia or North America is suggested while for the connection from Balmaceda to Haikou the path via Europe is the more common one.

7.4.5. Arusha to Balmaceda

Arusha and Balmaceda were introduced in Section 7.4.4 and Section 7.4.3 already. However, this connection is especially interesting because routes heading west and routes heading east around the world are available as it can be seen in Figure 16.

The special feature of both cities is that their airports provide no connections to *Hotspot Airports*. Hence, to reach a *Hotspot Airport* at least two flights are required. While the only connected airport from Balmaceda is Santiago, Arusha provides connections to Dar es Salaam, Zanzibar and Kilimanjaro airport. These airports then provide flights to *Hotspots*.

7.5. Experiments

This part describes and investigates the experiments of this section before they are discussed in Section 8. Therefore, seven experiments are applied. First, for each experiment its purpose and the connections used for it are introduced. Afterwards, the settings that were applied for the experiment are described and then the experimental results are investigated. The first experiment deals with the impact of the path length on the runtime. Experiment 2 investigates the impact of the different termination criteria. The next experiment concentrates on the impact of using the results of the *Hotspot Search* as input for the *MBFS* algorithm on the runtime and the quality of the *Hotspot Search* results. Experiments 4 to 7 investigate the impact of the result size, the number of origin and destination airports, the price for each hour of travel time and

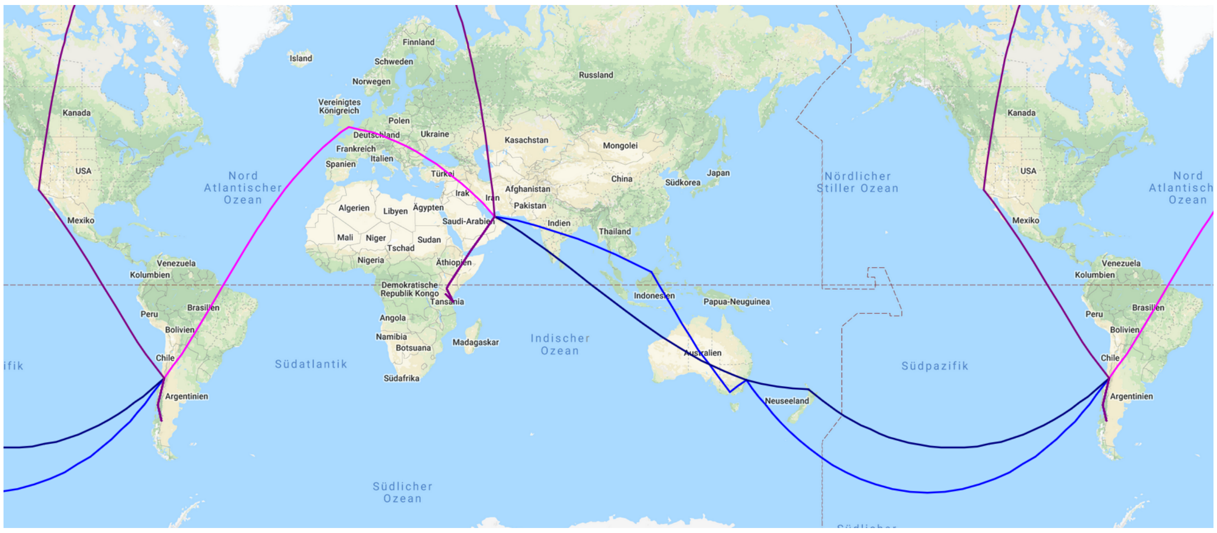


Figure 16: Connections between Arusha and Balmaceda.

the average speed on the runtime. For Experiments 5 to 7 their impact on the quality of the results is also investigated.

7.5.1. Experiment 1 - Path Length Impact on Runtime

This experiment investigates the runtime on different lengths of paths, while length in this case means the number of steps required between origin and destination. Two different connections are investigated: The connections between Arusha and Balmaceda (Figure 16) introduced in Section 7.4.5 for Experiment 1a and between Haikou and Balmaceda (Figure 15) introduced in Section 7.4.4 for Experiment 1b. For both connections the outbound connections as well as the inbound connections are considered to investigate whether there is a difference between them. Both connections are complex since a lot of steps are required to get a path between them. To investigate the impact of the length of a path on the runtime, not only the full connections are queried by the *Trip Planner* application, but also subsets of these connections as it will be described in the corresponding experiment. To investigate the relation of the runtime it is set in relation to the number of algorithmic steps required for the connection and the number of paths that were generated by the *MBFS* algorithm.

For this experiment the number of origin and destination airports is set to 1 since the connection between exactly those airports should be inspected. The number of results to show is set to 4. The average speed is set to 150 km/h and the price per hour is set to 64 Euro.

Experiment 1a - Arusha to Balmaceda A common connection from Arusha to Balmaceda uses flights via Dar es Salaam, Nairobi, Dubai, London, Santiago and Puerto Montt to finally reach Balmaceda. To compare the runtime of long paths with the runtime of shorter paths for this experiment, in addition to the full connection between Arusha and Balmaceda also the shorter parts of the connection between Dar es Salaam and Puerto Montt and between Dubai and Santiago are investigated.

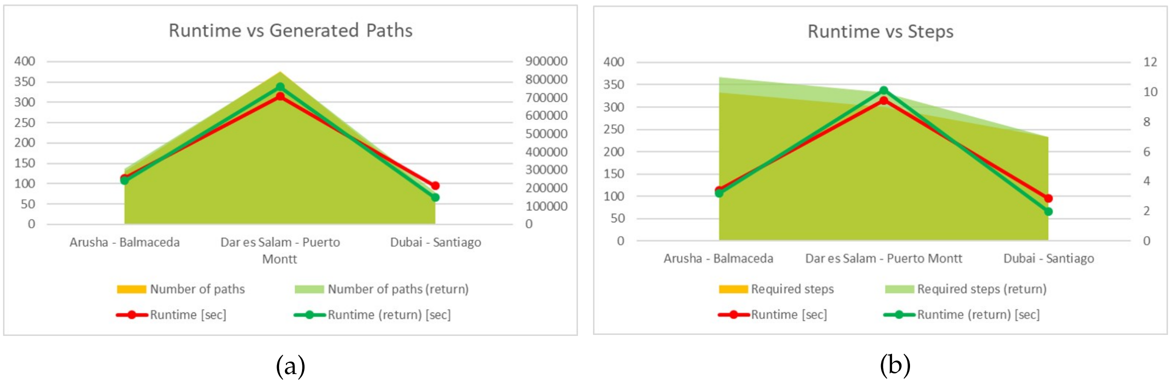


Figure 17: Relation of the runtime and number of generated paths (a) and relation of the runtime and the number of steps required by the *MBFS* algorithm (b), for the connection between Arusha and Balmaceda.

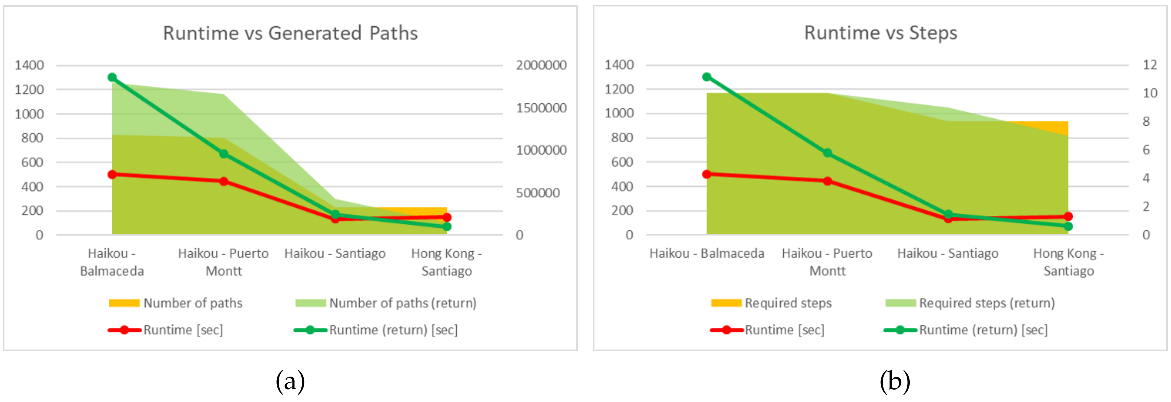


Figure 18: Relation of the runtime and number of generated paths (a) and relation of the runtime and the number of steps required by the *MBFS* algorithm (b), for the connection between Haikou and Balmaceda.

Figure 17 illustrates the runtime for a query for these connections in relation to the generated paths and the required algorithmic steps. It can be observed in Figure 17a that the number of generated paths for the inbound as well as the outbound connections are approximately equal and that the runtime scales very well with the number of paths the *MBFS* algorithm was generating. Figure 17b shows that for the connection between Arusha and Balmaceda the number of required algorithmic steps is pretty high while the runtime is low, the connection between Dar es Salaam and Puerto Montt requires less algorithmically steps but has a much higher runtime. The last connection between Dubai and Santiago has a better runtime as well as less algorithmic steps. The inbound and outbound connections differs only slightly for the first two connections. The underlying data can be obtained from Appendix A.3.1.

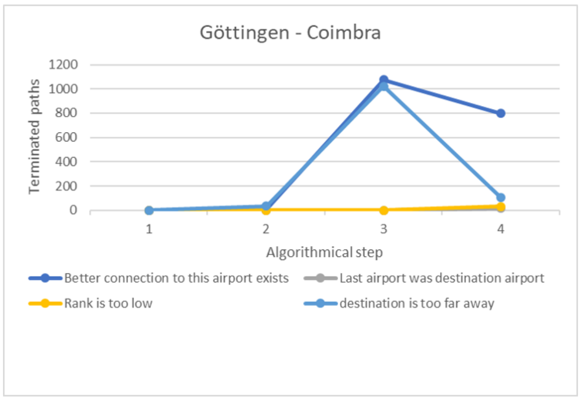
Experiment 1b - Haikou to Balmaceda A common connection from Haikou to Balmaceda is the path via Hong Kong, Auckland, Santiago and Puerto Montt. An interesting fact is that while the outbound connection prefers a path via Australia, usually a path via Europe is suggested by the algorithm for the inbound connection. For example from Balmaceda to Puerto Montt, Santiago, Sao Paulo, Frankfurt, Hong Kong and Haikou. However, the sub connections queried to the *Trip Planner* application for this experiment are Haikou to Balmaceda, Haikou to Puerto Montt, Haikou to Santiago, Hong Kong to Santiago and the return connections of them. While for the first two return connections the most result connections use paths via Europe, the connections from Santiago to Haikou and Santiago to Hong Kong use only connections over the Pacific Ocean.

Figure 18 shows the impact of the number of generated paths and the number of algorithmic steps on the runtime as well as the differences of the runtime, number of generated paths and required algorithmic steps between the outbound connections and inbound connections. First a huge difference of the generated paths between the outbound and inbound connections especially between Haikou and Balmaceda as well as Haikou and Puerto Montt can be observed in Figure 18a. Beside the fact that the runtime corresponds pretty well with the number of generated paths in general, there can be observed one exception in Figure 18a. While the number of generated paths between the connection from Puerto Montt to Haikou and Balmaceda to Haikou increases only very slightly, the runtime increases disproportionally. From Figure 18b it can be observed that the runtime slightly follows the number of algorithmic steps but the number of algorithmic steps for the connection from Puerto Montt to Haiko is similar to the required steps on the connection from Balmaceda to Haikou while the runtime is increasing very fast. The underlying data can be obtained from Appendix A.3.1.

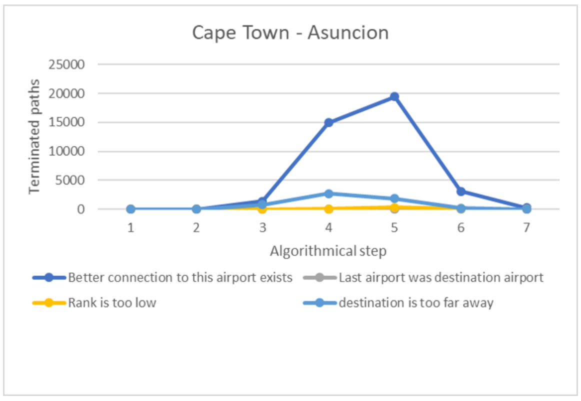
7.5.2. Experiment 2 - Impact of Different Termination Criteria

Section 5.3.2 introduces different termination criteria for paths. This experiment investigates how big the impact of each single termination criterion is by analyzing the number of connections that are terminated by each criterion. This is investigated on three different connections. The first connection is a short flight between the two European cities Göttingen and Coimbra introduced in Section 7.4.1. The difficulty of this flight is that both cities have no own airport. Therefore, the number of origin and destination airports considered is set to five. This connection requires four algorithmic steps. The second connection is the connection between Cape Town and Asuncion two metropolis with big airports as introduced in Section 7.4.2. Calculating this connection requires seven algorithmic steps. The last connection is the one between Arusha and Porto Alegre, one small African city and a Brazilian metropolis as introduced in Section 7.4.3. Finding this connection requires nine algorithmic steps. For all experiments the number of connections to show was set to 10, the average speed was set to 150 km/h and the price for each hour of travel time was set to 64 Euro.

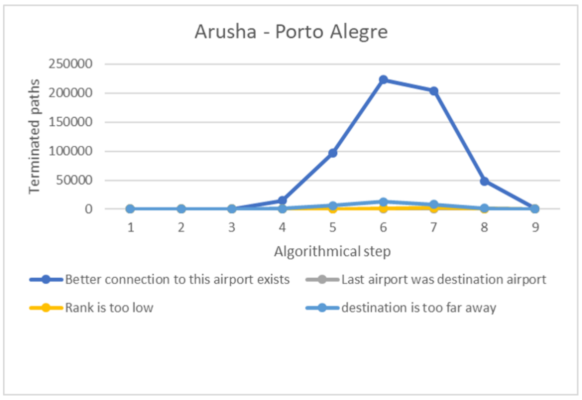
Figure 19 shows the results of the experiment. It can clearly be seen that most paths were terminated because there were other paths reaching the same airport earlier or with a better price. But especially for the short distance flight (Figure 19a) but also for the other connections the termination criterion that measures the remaining distance to the destination terminates a lot of paths as well. But even though the termination criterion that terminates paths when their rank is too low, that means if their virtual costs are already higher than the virtual costs for the worst connection of the result set, seems to be unimportant since its line on the chart is always



(a)



(b)



(c)

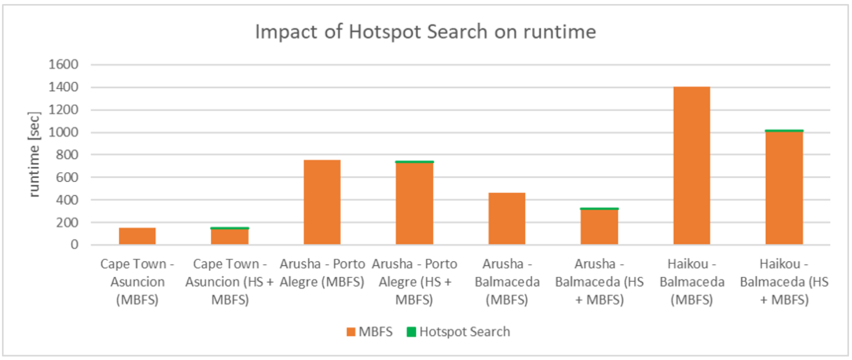
Figure 19: Shows on different connections for each termination criterion the number of paths that were terminated in the corresponding step because of this criterion.

close to zero but this termination criterion terminates up to 5,114 paths in total for the connection between Arusha and Porto Alegre. Some of the termination criteria are investigated with different settings again in other experiments later on.

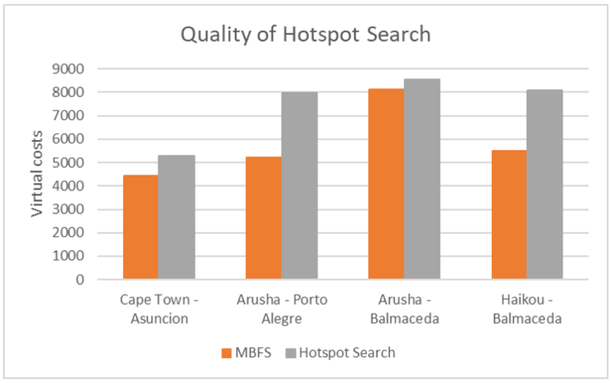
Not considered by these criteria are connections that have never been added to a path because they were rejected by expanding the path as described in Section 5.3.3. Thereby, about three to four times that many connections are rejected before being added to a path as paths that gets terminated. For the connection between Arusha and Porto Alegre for example 624,391 paths are terminated by the termination criteria and 2,379,973 paths are not even been generated because they are rejected before. The underlying data for this experiments is given in Appendix A.3.2.

7.5.3. Experiment 3 - Runtime Impact of Using Hotspot Search Results as Input for MBFS

This experiment shows the impact of the runtime by using the *Hotspot Search* before applying the *MBFS* algorithm. Therefore, the connections between Arusha and Porto Alegre, Arusha and Balmaceda, Cape Town and Asuncion and Haikou and Balmaceda are first evaluated by to the *MBFS* algorithm only and then by the *MBFS* algorithm after applying the *Hotspot Search* before. Also the quality differences between both algorithms are investigated in this experiment. Therefore, the previously listed connections are queried once only with the *Hotspot Search* algorithm and once only with the *MBFS* algorithm. Then the virtual costs of the best path found



(a) Illustration of the impact of the *Hotspot Search* on the runtime of the path finding.



(b) Differences of the virtual price of the best connection found by the according algorithm.

Figure 20: Runtime impact of applying the *Hotspot Search* before applying the *MBFS* algorithm and compares the quality of the results of the *Hotpots Search* and the *MBFS* algorithm.

in both algorithms are compared. Since the goal of the algorithms is to find a path with least virtual costs as possible, it can be assumed that a result with lower virtual costs has a higher quality.

For this experiment the number of origin and destination airports is set to one, the number of results shown is set to four, the average speed is set to 100 km/h and the price for each hour of travel time is set to 64 Euro.

Figure 20a shows the impact of the runtime by calling the *Hotspot Search* before using the *MBFS*. The chart shows that the *Hotspot Search* itself only requires very few seconds. The exact runtimes can be obtained from Appendix A.3.3. It can also be observed that the runtimes when calling the *Hotspot Search* first are lower than without using it. For the connections from Cape Town to Asuncion and from Arusha to Porto Alegre the runtime enhancement is only few seconds while the runtime enhancement of the connections from Arusha to Balmaceda and Haikou to Balmaceda comes up to 40%.

The virtual costs for the best path found for each connection by each path finding algorithm is illustrated in Figure 20b. Always the virtual costs for the best path are lower by using the *MBFS* algorithm, but for the connection from Cape Town to Asuncion and from Arusha to Balmaceda the difference is minimal. The difference of both for the other connections is quite high. But

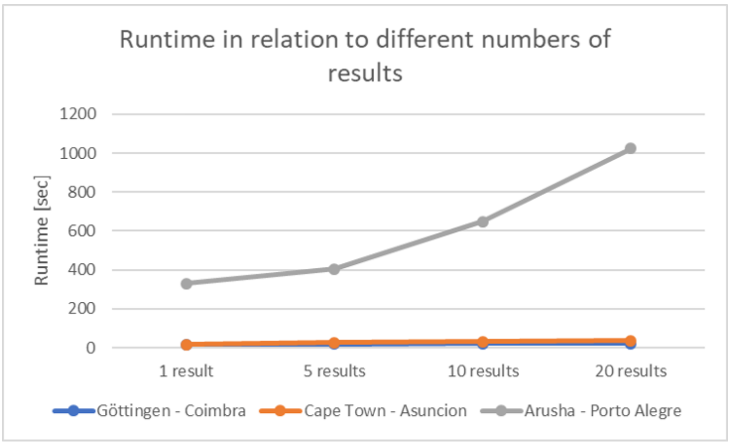


Figure 21: Impact of the result size on the runtime.

it turned out that the intermediate stops were mostly the same by using the *Hotspot Search* as for using the *MBFS*, but the *Hotspot Search* was choosing a more expensive flight for single sub connections as the *MBFS* algorithm because the *Hotspot Search* always takes the next possible connection instead of waiting for a maybe cheaper connection as described in Section 5.2. That results in a much higher price for the whole connection by using the *Hotspot Search* in a lot of cases.

7.5.4. Experiment 4 - Impact of Result Size on the Runtime

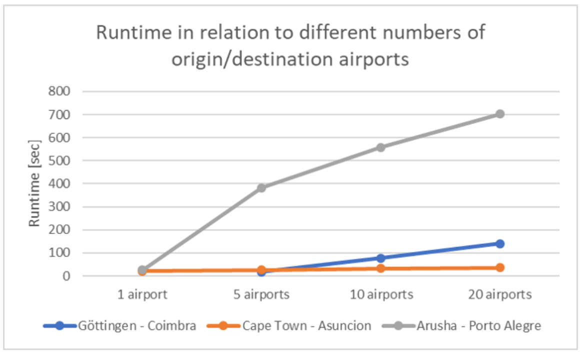
This experiment investigates the impact of the maximal number of results returned by the *MBFS* algorithm on its runtime. Since paths are terminating if their virtual costs exceed the virtual costs of the worst connection from the current result list, a runtime impact from the result size can be expected. For this experiment the connection from Göttingen to Coimbra (Section 7.4.1), the connection from Cape Town to Asuncion (Section 7.4.2) and the connection from Arusha to Porto Alegre (Section 7.4.3) are investigated.

For this experiment the number of origin and destination airports is set to five, the average speed is set to 150 km/h, the price for each hour of travel time is set to 64 Euro and for the result size the values one, five, ten and 20 are used.

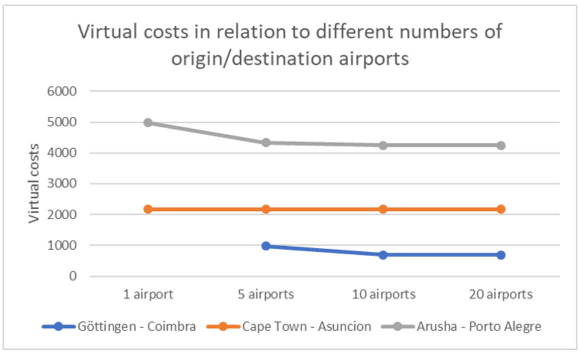
The connection from Arusha to Porto Alegre in Figure 21 shows clearly that the runtime increases with a higher number of results. The runtime for the other connections is slightly increasing for a higher number of results as well, but due to the small increase it can not be seen on the chart. Appendix A.3.4 contains a table with all detailed results.

7.5.5. Experiment 5 - Impact of the Number of Origin and Destination Airports on the Runtime and Result Quality

In this experiment the impact of the number of origin and destination airports that are considered by the *MBFS* algorithm on the runtime and the virtual costs is investigated. The virtual



(a)



(b)

Figure 22: Impact of the number of origin and destination airports on the runtime (a) and the virtual costs (b).

costs again represent the quality of the result. Lower virtual costs for the same connection indicates a higher quality of the result. Therefore three different connections are investigated, the connection from Göttingen to Coimbra, from Cape Town to Asuncion and from Arusha to Porto Alegre. The virtual costs are always measured for the best (cheapest virtual costs) result connection.

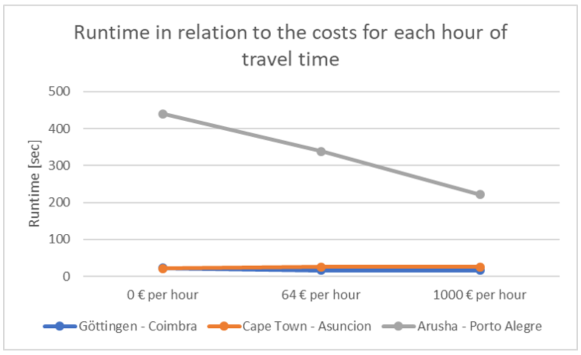
For this experiment the number of results is set to four, the average speed is set to 150 km/h and the price for each hour of travel time is set to 64 Euro. The experiment investigates the impact of one, five, ten and 20 origin and destination airports.

Figure 22a shows that for all connections the runtime increases for a higher number of airports. Especially for the connection from Arusha and Göttingen the runtime is increasing with a higher number of origin and destination airports. The runtime for more origin and destination airports from Cape Town which provides an airport with a lot of connections already is increasing only very slightly. According to Figure 22b the virtual costs are decreasing in the beginning but from five to ten airports on, the virtual costs are on a stable level and do not decrease anymore. The detailed test results can be obtained from Appendix A.3.5.

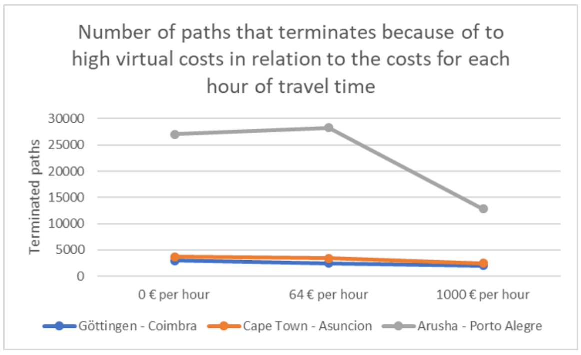
7.5.6. Experiment 6 - Impact of Used Price per Hour on the Runtime and Result Quality

This experiment investigates the impact of the price for each hour of travel time on the duration and price of a trip, the runtime of the *MBFS* and the number of paths that were terminated because the virtual costs were too high. For this, the connections from Göttingen to Coimbra, Cape Town to Asuncion and Arusha to Porto Alegre are investigated.

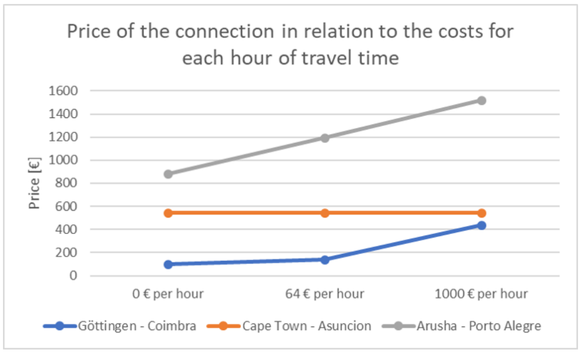
For this experiment the number of results is set to four, the average speed is set to 150 km/h and the number of origin and destination airports is set to five. The price for each hour of travel time is set to zero Euro, 64 Euro and 1000 Euro. While zero Euro means that the duration of the trip does not matter, which could be for example the case for a student with less money in the long summer break, the 1000 Euro can address the business manager who has less time and needs to be as fast as possible at the next place and does not care about the price. The 64 Euro



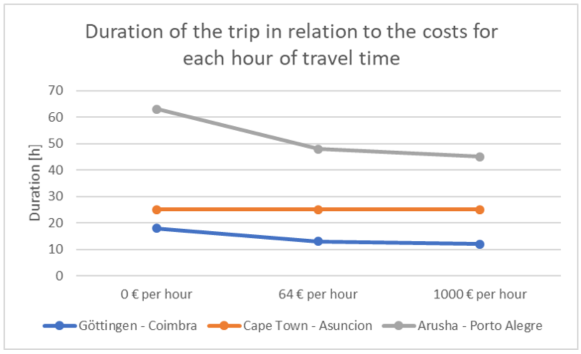
(a)



(b)



(c)



(d)

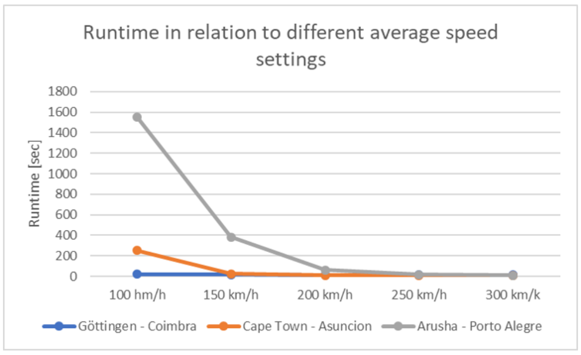
Figure 23: Impact of the chosen price for each hour of travel on the runtime (a), the termination criterion for too high virtual costs (b), the price (c) and the duration (d) of the trip.

for one hour could represent the holiday traveler who has not that much money but also not a lot of time.

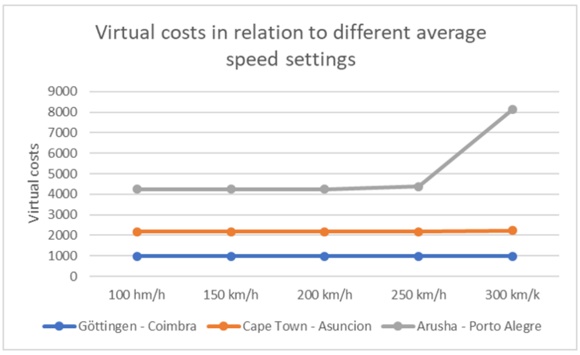
Figure 23a shows that the runtime is decreasing for a higher price for each hour of travel in general. But for the connection between Cape Town and Asuncion the runtime is stable. Figure 23c and Figure 23d point out that for this connection the duration and the price are stable as well. For the other connections the travel price is increasing and the travel duration is decreasing for a higher amount of money for each hour of travel time. Hence, if the price and the duration are stable the runtime is stable as well. Otherwise the runtime changes proportionally to the duration of a trip and non-proportionally to the price of a trip. Accordingly to Figure 23b the terminated paths because of too high virtual costs were generally decreasing with a higher price for each hour of travel time but they do not really correspond to the price for each hour of travel time. The detailed test results can be seen in Appendix A.3.6.

7.5.7. Experiment 7 - Impact of Used Average Speed on the Runtime and Result Quality

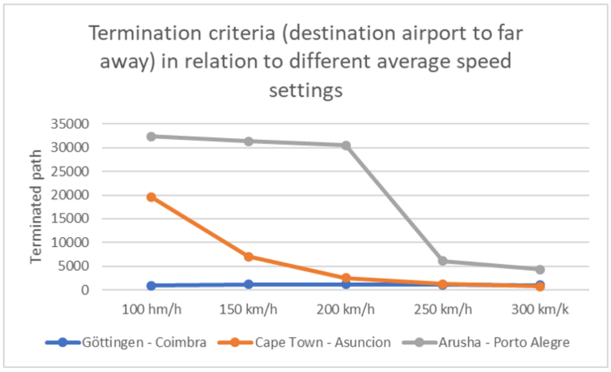
The last experiment investigates the impact of the average speed used on the runtime and the quality of the *MBFS* algorithm. The quality again is measured by the virtual costs of the best result connection. Also the number of paths that were terminated because their distance to the destination in relation to the required time is investigated. For this experiment again the connections from Göttingen to Coimbra, from Cape Town to Asuncion and from Arusha to Porto Alegre are used.



(a)



(b)



(c)

Figure 24: Impact of the used average speed on the runtime (a), virtual costs (b) and terminated paths because of the path is to far away from the destination (c)

For this experiment the number of results is set to four, the price for each hour of travel time is set to 64 Euro and the number of origin and destination airports is set to five. The average speed of the connection is set to all values from 100 km/h to 300 km/h in steps of 50 km/h.

Figure 24a shows a clear runtime enhancement for the connection from Cape Town to Asuncion and Arusha to Porto Alegre when the average speed is increasing. But also the runtime for the connection between Göttingen and Coimbra is decreasing even though not too strong because it already starts on a very low level. According to Figure 24b the virtual costs are very stable for the first time but start increasing then. The virtual costs for the connection from Arusha to Porto Alegre start increasing by an average speed of 250 km/h. While the connection between Göttingen and Coimbra stays stable even on an average speed of 300 km/h, the connection between Cape Town and Asuncion starts increasing its virtual costs on this point. Figure 24c shows that the number of terminated paths because of the distance to the destination airport is decreasing with a higher average speed, except the connection between Göttingen and Coimbra which has always approximately 1,000 terminated paths for this criterion. Appendix A.3.7 provides detailed test results for this experiment.

8. Discussion

After investigating the experiments in the previous section, this section discusses these results. It is preferable to find the optimal settings for the *MBFS* algorithm which guarantees a very high quality of the results, which means that it finds the best available connections and has a low runtime as well. But not for all settings it is possible to find proper generally valid settings since some settings depend on personal factors, as for example the money a person is willing to spend in order to save one hour of travel time. Also the number of results the user wants to return can maybe be influenced by the application but is a user preference in the end. However, for the average speed and the number of origin and destination airports, values could be suggested by the algorithm. The discussion evaluates proper values for these settings to obtain a very high quality level but also a low runtime. Also the purpose of the *Hotspot Search* is evaluated in this section. But due to the small number of test cases investigated in Section 7, the evaluation results give a tendency how the proper settings could look like but does not represent a meaningful overall result for the best settings.

8.1. Experiment 1

In Experiment 1 (Section 7.5.1) the impact of the length of a connection was investigated on the runtime of the *MBFS* algorithm. The experiment investigated the runtime differences of the inbound and outbound connections. For the connections between Arusha and Balmaceda (Experiment 1a) the inbound connections used the reverse paths of the outbound connections. The runtime for the outbound and inbound connections are really similar in this experiment. For the connection between Haikou and Balmaceda the outbound connections use paths via North America or Australia while the inbound connections are using paths via Europe, except the sub connections between Haikou and Santiago and Hong Kong and Santiago. While the runtime for these inbound and outbound connections is quite similar, the runtime of the full connection and the connection between Haikou and Puerto Montt differs strongly between the outbound and inbound connections. Therefore, it can be expected that if the inbound connections are using the reverse paths of the outbound connections the runtime is approximately the same, otherwise the runtime can differ heavily.

In Experiment 1 the runtime corresponds exactly to the number of generated paths while it does not correspond to the number of required algorithmic steps. Most of the runtime is required by expanding paths. Consequently, the runtime increases if the number of generated paths is increasing and the runtime decreases if the number of generated paths is decreasing, as it can be seen in Figure 17a. Figure 18a shows that the runtime for the connection between Balmaceda and Haikou increases disproportionally to the number of generated paths in relation to the connection between Puerto Montt and Haikou in Experiment 1b. As described in Section 6.2 each Connection object stores a lot of values and therefore requires some memory for storing. If a lot of these Connection objects are generated the amount of required memory exceeds the available memory and the memory starts to swap parts of it to the hard drive. Even if this is a SSD the swapping requires a lot of time and slows the application down. After exceeding

1.5 million generated paths for the connection from Balmaceda to Haikou the memory starts swapping and the runtime thus increases disproportionally. Finally, it can be observed that the runtime depends on the number of generated paths but increases disproportionally as soon as the available memory is not sufficient.

8.2. Experiment 2

Experiment 2 investigated the impact of the different termination criteria. Therefor for each termination criterion it was measured how many paths it has terminated in each algorithmic step to find a connection. Experiment 6 and 7 also investigated specific termination criteria. Experiment 6 investigated how many paths were terminated because of too high virtual costs and Experiment 7 investigated how many paths have been terminated because the remaining distance to the destination was too far in relation to the already required travel time.

It turns out really clearly that the most important termination of paths is the rejecting of connections before they are even added to a path. Because not even generating a path that has no chance to reach its destination in time saves a lot of runtime as figured out in Experiment 1. Figure 19 shows that the most paths were terminated because they were reaching an airport which was reached by a better connection already. Since this termination criterion terminates most paths it can be expected that it has the biggest impact on the runtime. But also due to the distance to the destination and too high virtual costs a lot of paths were terminated. Figure 23b from Experiment 6 shows that by using more origin and destination airports and less result connections the number of paths that are terminated because of too high virtual costs can increase heavily but has still no direct impact on the runtime. Figure 24c from Experiment 7 shows that the number of terminated paths because of a too long remaining distance to the destination is increasing when using a lower average speed. Figure 24a shows that the number of terminated paths due to the remaining distance to the destination corresponds to the runtime of the algorithm. Therefore, it can be expected that this termination criterion has a major impact on the behavior of the algorithm. That only very few connections are terminated because the last airport was one of the destination airports shows that this criterion has not a huge impact on the algorithm but is required anyway to ensure that paths that have reached their destination do not run onward for a long time.

8.3. Experiment 3

Since valid paths are required for the *MBFS* algorithm to terminate paths that have too high virtual costs, the idea is to apply the *Hotspot Search* before applying the *MBFS* algorithm to use the results of the *Hotspot Search* as an upper bound for the virtual costs in the *MBFS* algorithm. By doing this a runtime gain is expected. The purpose of Experiment 3 is to show this gain and compare the quality of the results from the *MBFS* algorithm and the *Hotspot Search*.

Experiment 3 pointed out that the *Hotspot Search* itself required only very few seconds for all connections. Therefore, also for very fast runs of the *MBFS* the runtime of the *Hotspot Search* has not a major impact on the total runtime of the search process. The experiment also shows that the total runtime by applying the *Hotspot Search* before using the *MBFS* was always faster than applying only the *MBFS* algorithm, in half of the cases a huge runtime enhancement was observed. Therefore, it can be recommended to always apply the *Hotspot Search* before using the *MBFS* algorithm.

In Experiment 3 it was also observed that the *Hotspot Search* found the same routes as the *MBFS* algorithm in a lot of cases. But due to the fact that the *Hotspot Search* only uses the next departing outbound connection to each reachable destination when expanding a path, in a lot of cases not the best flights for this connection were chosen. This results in higher costs and thus also higher virtual costs of the connection as by applying the *MBFS* algorithm. This proofs the significance of the *Hotspot Airports* but also points out possibilities for improving the algorithm. By using not only one, but more outbound connections between the same airports when expanding a path the algorithm could find better results. But that would turn into more generated paths and therefore in a higher runtime as discovered in Experiment 1. Even if the result quality of the *Hotspot Search* would be improved this way it could not be ensured that its connections are really the best available connections since the *Hotspot Search* always prefers to use a path via the *Hotspot Airports* which is only a small subset of all airports. Therefore even after improving the algorithm it still can only be used as input for the *MBFS* algorithm or to give a tendency for a good connection. However, whether the runtime improvement of the *MBFS* by using an improved *Hotspot Search* algorithm exceeds the runtime penalty of a better *Hotspot Search* could be part of a separate study.

8.4. Experiment 4

Only the best connections, that means the connections with the lowest virtual costs are stored as results by the *MBFS* algorithm. The number of paths that have to be stored is defined by the user as the number of connections that have to be shown. All paths whose virtual costs are worse than the virtual costs of the connection with the lowest rank, that means the highest virtual costs from the result list, are terminated. For a smaller list it can be expected that the virtual costs of the worst connection are lower than for the worst connection if the list was be longer. Consequently paths are terminated earlier because of their virtual costs by using a smaller result list. Since Experiment 1 shows that the number of generated paths corresponds to the runtime a better runtime is expected by using a smaller number of result connection. Experiment 4 investigated this expectation. It turned out that for all connections the runtime was increasing for a higher number of results. But the impact on the runtime increases with the complexity of the connection. Therefore, the runtime for the connection between Arusha and Porto Alegre was increasing much faster than the runtime for the other connections.

8.5. Experiment 5

Experiment 5 investigated the impact of the number of considered origin and destination airports on the runtime and the result quality of the *MBFS* algorithms. With a higher number of origin and destination airports more different connections are possible. It is possible that the closest airports provides no proper connections while a further airport provides a direct connection where it is worth for to have a longer trip to the airport. In such cases it is expected that the result quality is increasing when the number of origin and destination airports is increasing. But it is also possible that the best connection departs from the closest airport already. In this case a higher number of origin and destination airports is expected to have no influence on the result quality. Experiment 5 underpins these assumptions. However, by increasing the number of origin and destination airports, new possible connections appear, this leads to a higher number of paths which leads to a higher runtime as shown in Example 1. Experiment 5 underpins this thesis as well. The difficulty is to decide how many origin and destination airports are worth to use, to get a good result but also to keep the runtime low. In Experiment 5 there is no quality enhancement by using more than ten origin and destination airports but the runtime is still increasing clearly. Therefore, it is suggested to use not more than 10 origin and destination airports.

8.6. Experiment 6

Experiment 6 investigated the impact of the price a user is willing to pay to save one hour of travel time on the runtime and quality of the results. While it is expected that the price of a connection increases and the duration decreases if the user is willing to pay more for saving one hour of travel time, this behavior is observed by this experiment for the connections from Göttingen to Coimbra and from Arusha to Porto Alegre. But the connection from Cape Town to Asuncion stays stable on both, the duration and the price. This behavior can be observed if the fastest connection is also the cheapest connection. Experiment 6 pointed out that for this connection the runtime stays stable as well while for the other connections the runtime decreases proportionally to the duration. The runtime improvement when using a high price for each hour of travel time saved can be explained with the prompt termination of all paths which require more time by using a detour even if they save a lot of money by doing so. The lower number of terminated paths for a very high price for each hour of travel time saved, seen in Figure 23b, can be explained by the very low number of all generated paths because a lot of paths have already been terminated in a very early stadium.

8.7. Experiment 7

The last experiment investigated the impact of the minimum average speed on the runtime and the quality of the results. A clear runtime enhancement was shown by this experiment for all connections by using a higher average speed. Especially in the range from 100 km/h to 200 km/h huge runtime enhancements were observed. The enhancement can be justified by the higher number of paths that are terminated at an early point, because the distance to the destination is too high in relation to the already required travel time of a path, for the higher average speed. It can be confusing that Figure 24c shows that the total number of paths that were terminated due to this termination criterion is decreasing. The reason for that is that the paths by using a higher average speed are terminated pretty early. By terminating the paths in the beginning the total number of generated paths is kept very low and therefore even if less paths were terminated in total by this criterion the impact of this criterion was increasing and the total number of generated paths was pretty low.

Terminating more paths by increasing the average speed leads at any point to the problem that paths which would be a good result are terminated as well. This experiment investigated this issue as well. It was observed that until an average speed of 200 km/h the virtual costs for all connections were stable but then they started to increase, which means that the best connections were terminated. To prevent this the average speed should be keep low enough to receive good results but also high enough to obtain a good runtime. Since the runtime for an average speed of 200 km/h is already pretty low and the virtual costs are still stable at this point, according to this experiment an average speed of 200 km/h can be suggested.

9. Conclusion and Further Research

First the conclusion gives a short review about the topics dealt with in this thesis. Afterwards the results of this thesis are summarized and in the end some further research fields are pointed out. In this thesis the development of a graph data structure to cache flight data has been depicted. Also a subset with the most important airports worldwide for inter-continental flights has been found and a method to approximate flight prices has been developed. The development of different path finding algorithms has been done in this thesis as well. The *Hotspot Search* algorithm and the *MBFS* algorithm was developed. The main purpose of the *Hotspot Search* algorithm is to fastly find a possible connection between two arbitrary places which does not necessarily need to be the best connection it is used as an upper bound for the *MBFS* algorithm to increase its runtime. The purpose of the *MBFS* algorithm is to find the best connections between any two places worldwide. Furthermore, experiments on the algorithms with different settings to evaluate their performance and to find the best settings for different properties have been done. To perform the experiments an application which uses the algorithms has been developed.

In Section 7 and 8 it turned out that the main impact of the runtime comes from the number of paths that were generated by the search algorithm. Therefore, it is required to keep this number as low as possible. Experiment 2 shows that the termination criteria are closing a lot of paths which would not reach the destination in a proper time or with a proper price. It was also determined that the use of the *Hotspot Search* as input for the *MBFS* algorithm is reasonable in general, since it improves the runtime. As a number of origin and destination airports, ten turned up as a reasonable value. By using less than ten origin and destination airports the result quality was decreasing and by using more than ten origin and destination airports only the runtime was increasing, however, no quality enhancement could be observed anymore. Experiment 7 determined for the average speed that 200 km/h is a good value since the result quality was stable until this point and the runtime has already been reduced significantly. As pointed out before, for the price each hour of additional travel time costs the user no specific price can be suggested since it strongly depends on personal factors like the amount of available money and time. But by choosing a lower amount of money a higher runtime can be expected. If the user is willing to pay more money for his flights by saving time on the travel duration it is reasonable also to increase the average speed for the query to terminate slow connections at an early point. The result size also depends on the personal preferences of the user. If a user is fine with only getting the connection with the best price and time ratio, it is sufficient to return only the best connection. But often the user wants to be able to decide between several good connections and does maybe not take the best price and time ratio connection but another one because of personal experiences or other reasons. Thereby the user can expect a higher runtime for a larger result set.

The introduction listed two basic requirements that the developed algorithm has to fulfill:

- Find the best fitting origin and destination airports for the given origin and destination places and find a connection to these airports.

- Find a flight connection between these airports in case a connection exists.

Experiment 5 (Section 7.5.5) shows that the *MBFS* algorithm dynamically reacts on different origin and destination airports and finds better connections if the algorithm considers more origin and destination airports. Section 7.4.5 shows on behalf of the connection between Arusha and Balmaceda that even for connections where conventional flight search engines like *Skyscanner* or *eStreaming* can not find a connection anymore, the *MBFS* algorithm as well as the *Hotspot Search* algorithm are able to find connections by combining offers from different suppliers.

In the end of this thesis some further research areas and limitations that turned out during writing this thesis but would exceed the scope of it will be outlined. A massive problem during the whole time of research was to obtain proper test data since especially the flight price APIs are strongly restricted and provide only cached data [Skyd] [eSt]. Especially to obtain live price data would increase the result quality and would be essential for a productive system. Since the runtime tests were done on a private laptop instead of a dedicated server system, the results are not that accurate because a background task was running and the performance of a laptop is worse than the performance of a server system. Therefore, it could be assumed to achieve better runtimes on a proper server system.

Section 4.3 introduced a method to approximate flight prices. Since it was not the scope of this thesis to develop price approximation algorithms the approximation is limited to a minimum. But earlier studies discovered that it is possible to approximate flight prices pretty accurately when considering enough attributes of the flight [ETKY03] [MPR09]. Especially with the available computing power in our days new possibilities are arising. Section 8.3 dealt with the question whether it is reasonable to increase the quality of the *Hotspot Search* algorithm by accepting a runtime disadvantage. To improve this algorithm and maximize the runtime enhancement of the full search process could be part of a separate thesis. As pointed out above in this section it is reasonable to increase the minimum average speed value as well when increasing the price for each hour of travel time. A further research could deal with the idea of automatically increasing the average speed value when increasing the price for one hour of travel time. To determine an accurate dependency between both values is a non-trivial problem.

Experiment 1 pointed out that the amount of required memory for the application is quite high and a problem when it comes to complex connections because the RAM starts to swap to the hard drive at any point and thus slows down the application massively. To avoid this problem the Connection object introduced in Section 6.2 could be minimized and certain properties could be loaded from the database or the APIs when required to decrease the required amount of RAM. The problem is that the properties stored in this object are required by the search algorithms or that they are parts of API responses which are required at an early point because of other required properties. Since the APIs are pretty slow and limited only to a certain number of calls for each day it has to be avoided to query them several times for the same properties. A possible solution could be to outsource some data to a database level, compress the Connection objects or proceed even more operations on the database level to reduce the required amount of memory. Another idea could be to cache full results and parts of it in the database to reuse them for a later similar query. Then a lot of path finding steps could be avoided just by using the cached results. This could also increase the runtime of the algorithms significantly.

List of Tables

List of Figures

Code Snippets

References

[AG87] B. Awerbuch and R. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Transactions on Information Theory*, 33(3):315–322, 1987.

[BM06] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pp. 523–530. IEEE, 2006.

[CEE] CEE. CEE Travel Systems. Available at `http://www.cee-systems.com/`.

[Che] Cheapair.com. When to buy airline tickets ? Based on 1.5 Billion Airfares. Available at `https://www.cheapair.com/blog/travel-tips/when-to-buy-airline-tickets-based-on-1-5-billion-airfares/`.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. *The MIT Press*, 2001.

[eSt] eStreaming. eStreaming API home page. Available at `http://estrapi.cee-systems.com`.

[ETKY03] O. Etzioni, R. Tuchinda, C. A. Knoblock, and A. Yates. To buy or not to buy: mining airfare data to minimize ticket purchase price. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 119–128. ACM, 2003.

[Eve11] S. Even. *Graph algorithms*. Cambridge University Press, 2011.

[Far] Farecompare.com. Cheapest Days to Fly and Best Time to Buy Airline Tickets. Available at `https://www.farecompare.com/travel-advice/tips-from-air-travel-insiders/`.

[Gona] Y. Goncharenko. Cache API - CEE eStreaming API. Available at `https://docs.travelcloudpro.eu/cached-api.html`.

[Gonb] Y. Goncharenko. Introduction to eStreaming API - CEE eStreaming API. Available at `https://docs.travelcloudpro.eu/`.

[Gonc] Y. Goncharenko. Point Of Sale - CEE eStreaming API. Available at `https://docs.travelcloudpro.eu/pointofsale.html`.

[Gond] Y. Goncharenko. Using Postman Application - CEE eStreaming API. Available at `https://docs.travelcloudpro.eu/using-postman-application.html`.

[Gooa] Google. Die Google Maps Geolocation API | Google Maps Geolocation API | Google Developers. Available at `https://developers.google.com/maps/documentation/geolocation/intro`.

[Goob] Google. Entwickler-Leitfaden | Google Maps Directions API | Google Developers. Available at `https://developers.google.com/maps/documentation/directions/intro?hl=de`.

[Gooc] Google. Entwickler-Leitfaden | Google Maps Geocoding API | Google Developers. Available at `https://developers.google.com/maps/documentation/geocoding/intro#ComponentFiltering`.

[Good] Google. Erste Schritte | Google Maps Time Zone API | Google Developers. Available at `https://developers.google.com/maps/documentation/timezone/start`.

[Gooe] Google. Google Maps APIs for Web | Google Developers. Available at https://developers.google.com/maps/web/.

[Goof] Google. Google Maps JavaScript API V3 Reference. Available at https://developers.google.com/maps/documentation/javascript/reference.

[Goog] Google. Google Maps Web Service APIs | Google Developers. Available at https://developers.google.com/maps/web-services/.

[Gooh] Google. Google Places API Web Service | Google Developers. Available at https://developers.google.com/places/web-service/.

[Gooi] Google. Preise und Nutzungsmodelle | Preise und Nutzungsmodelle für Google Maps APIs | Google Developers. Available at https://developers.google.com/maps/pricing-and-plans/.

[Hen18] F. Henke. Google Maps, Skyscanner and eStreaming APIs for collecting and presenting travel route information, 2018.

[Hor10] T. D. Hornung. *Query workflows over web data sources.* PhD thesis, University of Freiburg, 2010.

[IAT] IATA. IATA - IOSA Registry. Available at http://www.iata.org/whatwedo/safety/audit/iosa/Pages/registry.aspx.

[ITA] ITA Software by Google. ITA Software by Google. Available at http://www.itasoftware.com/.

[KS05] R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *AAAI*, volume 5, pp. 1380–1385, 2005.

[LS10] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pp. 303–314. ACM, 2010.

[MPR09] P. Malighetti, S. Paleari, and R. Redondi. Pricing strategies of low-cost airlines: The Ryanair case study. *Journal of Air Transport Management*, 15(4):195–203, 2009.

[MS08] K. Mehlhorn and P. Sanders. *Algorithms and data structures: The basic toolbox*. Springer Science & Business Media, 2008.

[RNC$^+$03] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

[Skya] Skyscanner. API Reference. Available at https://skyscanner.github.io/slate/#geo-catalog.

[Skyb] Skyscanner. Skyscanner - Discover Skyscanner. Available at https://www.skyscanner.net/aboutskyscanner.aspx.

[Skyc] Skyscanner. Skyscanner Homepage. Available at https://www.skyscanner.de/.

[Skyd] Skyscanner. Which services can I access and what are the rate limits? Available at https://support.business.skyscanner.net/hc/en-us/articles/206800359-Which-services-can-I-access-and-what-are-the-rate-limits.

[Skye] Skyscanner. Why do I get an error when accessing the Flights Live Pricing API? Available at `https://support.business.skyscanner.net/hc/en-us/articles/212682245-Why-do-I-get-an-error-when-accessing-the-Flights-Live-Pricing-API`.

[Tra] Travelport. Travelport Home Page. Available at `https://www.travelport.com/`.

[YCH+05] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 25. IEEE Computer Society, 2005.

[Zus72] K. Zuse. *Der Plankalkül*. Gesellschaft für Mathematik und Datenverarbeitung, 1972.

A. Appendix

A.1. Statistic Queries

Queries to the cached flights database, to give statistical information about the stored data.

A.1.1. All Airports

Returns all airports listed in the database.

```sql
SELECT *
FROM airports;
```

A.1.2. All Served Airports

Returns all served airports listed in the database.

```sql
SELECT DISITINCT origin
FROM flight_connections;
```

A.1.3. Different Direct Flights

Returns all different direct connections in a certain time period listed in the database.

```sql
SELECT DISITINCT origin, destination
FROM flight_connections
WHERE connection_number IS NULL
AND departure_date BETWEEN '2018-04-05 00:00:00'::timestamp AND '2018-04-18
     23:59:00'::timestamp;
```

A.1.4. Different Connected Flights

Returns all different connected connections in a certain time period listed in the database.

```sql
SELECT DISITINCT origin, destination
FROM flight_connections
WHERE connection_number IS NOT NULL
AND departure_date BETWEEN '2018-04-05 00:00:00'::timestamp AND '2018-04-18
     23:59:00'::timestamp;
```

A.1.5. Total Flights in Time Period

Returns all served airports listed in the database.

```sql
SELECT origin, destination
FROM flight_connections
WHERE connection_number IS NULL
AND departure_date BETWEEN '2018-04-05 00:00:00'::timestamp AND '2018-04-18
     23:59:00'::timestamp;
```

A.1.6. Number of Flights

Returns for each day on the defined time periode the amount of flights listed in the database.

```
SELECT departure_date , count(departure_date)
FROM  (
  SELECT origin , destination , connection_number , CAST(CAST(departure_date
     AS date) AS timestamp) AS departure_date
  FROM flight_connections ) AS connections
WHERE departure_date BETWEEN '2018-04-05 00:00:00'::timestamp AND '
   2018-04-18 23:59:00'::timestamp
AND connection_number IS NULL
GROUP BY departure_date;
```

A.2. Hotspot Airports

Table 4: Hotspot Airports

Position	Airport name	IATA	Inter-continental flights
1	Istanbul Ataturk	IST	135
2	Dubai	DXB	118
3	Frankfurt am Main	FRA	84
4	Hong Kong International	HKG	82
5	Amsterdam	AMS	74
6	Bangkok Suvarnabhumi	BKK	71
7	Paris Charles de Gaulle	CDG	69
8	London Heathrow	LHR	63
9	Hamad International	DOH	63
10	Abu Dhabi International	AUH	61
11	Singapore Changi	SIN	60
12	Miami International	MIA	55
13	Madrid	MAD	51
14	Ben Gurion Intl	TLV	48
15	Los Angeles International	LAX	45
16	Taipei Taiwan Taoyuan	TPE	42
17	Seoul Incheon Int'l	ICN	40
18	Munich	MUC	40
19	Panama City Tocumen International	PTY	38
20	Kuala Lumpur International	KUL	37
21	Guangzhou	CAN	36
22	Istanbul Sabiha	SAW	36
23	Tokyo Narita	NRT	35
24	Brussels International	BRU	34
25	Atlanta Hartsfield-Jackson	ATL	33
26	Zurich	ZRH	33
27	Shanghai Pu Dong	PVG	32
28	Manila Ninoy Aquino	MNL	32
29	San Francisco International	SFO	30
30	Cairo	CAI	30
31	New York John F. Kennedy	JFK	30
32	Sydney	SYD	29
33	Bogota	BOG	29
34	Addis Ababa	ADD	28
35	Toronto Pearson International	YYZ	28
36	Mumbai	BOM	27
37	New Delhi	DEL	27
38	Jeddah	JED	26
39	Muscat	MCT	26
40	Moscow Sheremetyevo	SVO	25

Position	Airport name	IATA	Inter-continental flights
41	New York Newark	EWR	25
42	Lisbon	LIS	24
43	Rome Fiumicino	FCO	23
44	Beijing Capital	PEK	23
45	Mexico City Juarez International	MEX	22
46	Melbourne Tullamarine	MEL	21
47	Havana	HAV	21
48	Riyadh	RUH	21
49	Kuwait	KWI	20
50	Auckland International	AKL	19
51	Ho Chi Minh City	SGN	19
52	Vienna	VIE	19
53	Sao Paulo Guarulhos	GRU	19
54	Tbilisi	TBS	19
55	Punta Cana	PUJ	18
56	Fort Lauderdale International	FLL	18
57	Bahrain	BAH	18
58	Houston George Bush Intercntl.	IAH	18
59	Boston Logan International	BOS	18
60	Barcelona	BCN	18
61	Colombo Bandaranayake	CMB	18
62	San Salvador	SAL	17
63	Dallas Fort Worth International	DFW	17
64	Hanoi	HAN	17
65	Johannesburg O.R. Tambo	JNB	17
66	Casablanca Mohamed V.	CMN	17
67	Lima	LIM	17
68	Santiago Arturo Merino Benitez	SCL	17
69	Tokyo Haneda	HND	17
70	Dublin	DUB	16

A.3. Experimental Data

This part of the appendix provides the log outputs for the program fetches that was evaluated in Section 7. The output contains the runtime and the amount of found valid connections for the *Hotspot Seacrch* if applied. Also the total amount of generated paths, visited airports, *Google API* calls and the runtime of the *MBFS* algorithm as well as a table with stepwise information about the generated paths is provided.

The table contains information about which termination criteria was applied how often for each step as well as how many paths was reaching the destination (Dest. 1) and how many paths was open in each step (Open path). Therefor the table contains the following termination criteria. The first column describes how often a path was terminating because a better connection was existing to the same airport already (BCSA). The next column describes, how many paths was terminating because the maximum amount of steps was applied (Steps). (Dest. 2) describes how many paths was terminating because a destination airport was found in the previous step. The fourth column describes for each step, how many paths was terminating because there virtual costs are higher than the costs from the worst path of the current result set (rank). (too far) describes how many paths was terminating because there distance to the destination was too far accordingly to the termination criteria. The third last column contains the amount of flights that that was not even been added to a path because it would result in a cycle (Cycle) and the second last column counts the amount of flights was never been added to a path because other flights on the same connection was better (Bad con.).

For experiments that used a lot of different program calls, but required only very few of the logging values, not the whole log output but only the required properties organized in tables are shown to ensure clarity.

A.3.1. Experiment 1

This experiment investigates the impact of the length of a path on the runtime.

Arusha - Balmaceda :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 4
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 8 seconds
Number of *Hotspot Connections* found: 4

Step	1	2	3	4	5	6	7	8	9	10	Total
BCSA	0	0	2	1049	19412	76085	110429	55048	9917	333	272275
Steps	0	0	0	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	0	0	0	0	0	0
Rank	0	0	0	3	28	116	419	512	175	1	1254
Too far	0	0	0	246	2088	5871	6844	3517	628	21	19215
Dest. 1	0	0	0	0	0	0	0	8	8	0	16
Cycle	0	1	89	3413	19612	34656	25827	4849	58	0	88505
Bad con.	2	57	5610	72363	283709	391003	182738	35547	982	0	972009
Open paths	1	1	14	1680	23324	84827	119481	59582	10744	355	300008

Total number of generated paths: 292760

Total number of visited airports: 842
Google API calls: 17
Time needed for *MBFS*: 114 seconds

Balmaceda - Arusha :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 4
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 6 seconds
Number of *Hotspot Connections* found: 8

Step	1	2	3	4	5	6	7	8	9	10	11	Total
BCSA	0	0	0	10	1197	15685	67927	124863	72684	9121	169	291656
Steps	0	0	0	0	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	0	0	0	0	0	0	0
Rank	0	0	0	0	5	76	395	431	146	20	0	1073
Too far	0	0	0	2	124	1221	3655	6945	3223	397	5	15572
Dest. 1	0	0	0	0	0	0	3	2	0	0	0	5
Cycle	0	0	13	331	5267	15327	28740	22900	4187	102	0	76867
Bad con.	0	4	230	5724	67122	268278	479129	236455	28264	559	0	1085765
Open paths	1	1	2	46	1750	18665	74833	134072	76313	9544	174	315400

Total number of generated paths: 308306
Total number of visited airports: 864
Google API calls: 6
Time needed for *MBFS*: 107 seconds

Dar es Salaam - Puerto Montt :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 4
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 7 seconds
Number of *Hotspot Connections* found: 4

Step	1	2	3	4	5	6	7	8	9	Total
BCSA	0	4	759	41168	277759	345450	127288	24700	2409	819537
Steps	0	0	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	1	0	0	0	1
Rank	0	0	2	33	1073	4336	3258	971	120	9793
Too far	0	0	57	2052	8791	6717	1272	78	0	18967
Dest. 1	0	0	0	0	10	4	0	0	0	14
Cycle	0	82	6857	71616	148414	81447	21278	4451	0	334145
Bad con.	159	4679	162598	985280	1223471	478292	105146	13389	0	2972855
Open paths	1	27	1647	49016	297042	360426	132484	25810	2529	868981

Total number of generated paths: 848312
Total number of visited airports: 961

Google API calls: 41
Time needed for *MBFS*: 315 seconds

Puerto Montt - Dar es Salaam :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 4
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 3 seconds
Number of *Hotspot Connections* found: 4

Step	1	2	3	4	5	6	7	8	9	10	Total
BCSA	0	0	19	1126	22317	166072	363513	215619	21617	3	790286
Steps	0	0	0	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	2	2	0	0	0	4
Rank	0	0	0	10	1325	12100	23990	12742	1147	0	51314
Too far	0	0	3	100	897	1053	142	0	0	0	2195
Dest. 1	0	0	0	0	5	14	1	0	0	0	20
Cycle	0	16	449	5141	29436	89328	68115	9577	2	0	202064
Bad con.	12	228	6008	99329	670570	1479532	894673	92421	0	0	3242761
Open paths	1	3	61	1763	27990	186918	392659	228844	22765	3	861006

Total number of generated paths: 843819
Total number of visited airports: 877
Google API calls: 23
Time needed for *MBFS*: 338 seconds

Dubai - Santiago :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 4
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 4 seconds
Number of *Hotspot Connections* found: 8

Step	1	2	3	4	5	6	7	Total
BCSA	0	3	8686	65777	60704	9295	123	144588
Steps	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	0	0	0
Rank	0	2	297	1957	2075	325	6	4662
Too far	0	0	0	0	0	0	0	0
Dest. 1	0	0	5	5	1	0	0	11
Cycle	0	998	18283	23941	6145	102	0	49469
Bad con.	350	37246	225720	198579	33442	256	0	495243
Open paths	1	194	10958	70368	63375	9633	129	154657

Total number of generated paths: 149261
Total number of visited airports: 929
Google API calls: 205
Time needed for *MBFS*: 95 seconds

Santiago - Dubai :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 4
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 4 seconds
Number of *Hotspot Connections* found: 6

Step	1	2	3	4	5	6	7	Total
BCSA	0	5	1397	29537	82633	40978	1498	156048
Steps	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	0	0	0
Rank	0	0	363	5102	9617	4090	137	19309
Too far	0	0	0	0	0	0	0	0
Dest. 1	0	0	7	23	15	0	0	45
Cycle	0	389	5592	13519	7888	390	0	27778
Bad con.	143	7697	112590	283342	155287	6320	0	565236
Open paths	1	46	2405	35879	92856	45087	1635	177908

Total number of generated paths: 175402
Total number of visited airports: 806
Google API calls: 91
Time needed for *MBFS*: 66 seconds

Haikou - Balmaceda :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 4
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 6 seconds Amount of *Hotspot Connections* found: 4

Step	1	2	3	4	5	6	7	8	9	10	Total
BCSA	0	2	289	14760	142401	419571	411583	159645	24585	1991	1174827
Steps	0	0	0	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	0	0	0	0	0	0
Rank	0	0	0	2	34	1438	4864	3286	645	59	10328
Too far	0	0	0	4	26	115	111	21	0	0	277
Dest. 1	0	0	0	0	0	4	3	0	0	0	7
Cycle	0	71	2176	33049	115874	147802	75801	14463	1261	0	390497
Bad con.	47	1921	61597	507728	1383187	1261210	495898	79960	5382	0	3796883
Open paths	1	12	626	18789	157178	440020	425215	164468	25358	2050	1233716

Total number of generated paths: 1185439 Visited airports: 1017 Google API calls: 19 Time needed for *MBFS*: 502 seconds

Balmaceda - Haikou :

Departure date: April 5th 2018
Number of origin and destination airports: 1

Result size: 4
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 5 seconds
Amount of *Hotspot Connections* found: 4

Step	1	2	3	4	5	6	7	8	9	10	11	Total
BCSA	0	0	1	11	964	23361	176362	526136	648291	274110	29050	1678286
Steps	0	0	0	0	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	0	0	0	0	0	0	0
Rank	0	0	0	0	5	1115	13045	39283	45333	16522	1328	116631
Too far	0	0	0	0	0	21	55	60	6	0	0	142
Dest. 1	0	0	0	0	0	6	10	0	0	0	0	16
Cycle	0	1	6	258	4207	34001	113540	162227	85623	10268	0	410131
Bad con.	0	3	125	4696	87802	631695	1792629	2088762	874735	93750	0	5574197
Open paths	1	1	2	41	1432	28187	199567	576372	698115	291088	30378	1825183

Total amount of generated paths: 1795075
Total amount of isited airports: 951
Google API calls: 17
Time needed for *MBFS*: 1302 seconds

Haikou - Santiago :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 4
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 15 seconds
Number of *Hotspot Connections* found: 28

Step	1	2	3	4	5	6	7	8	Total
BCSA	0	0	508	14058	87966	143462	63919	5253	315166
Steps	0	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	0	0	0	0
Rank	0	0	34	749	4137	5412	1726	63	12121
Too far	0	0	0	0	0	0	0	0	0
Dest. 1	0	0	0	4	0	0	0	0	4
Cycle	0	71	2237	20186	38057	16536	1117	0	78204
Bad con.	47	2958	59047	325227	523617	231626	21132	0	1163607
Open paths	1	12	876	17476	96135	150491	65750	5316	336056

Total number of generated paths: 327291
Total number of visited airports: 945
Google API calls: 16
Time needed for *MBFS*: 134 seconds

Santiago - Haikou :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 4
Averrage speed: 150km/h

Price per hour: 64 Euro

Time needed for *Hotspot Search*: 3 seconds
Number of *Hotspot Connections* found: 4

Step	1	2	3	4	5	6	7	8	9	Total
BCSA	0	0	1094	24115	141992	188345	61133	4401	165	421245
Steps	0	0	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	0	0	0	0	0
Rank	0	0	37	667	3614	3817	1183	55	1	9374
Too far	0	0	0	0	0	0	0	0	0	0
Dest. 1	0	0	0	10	5	1	0	0	0	16
Cycle	0	218	5756	36629	59908	27497	2953	212	0	133173
Bad con.	122	6113	97050	481687	591452	207947	15595	764	0	1400608
Open paths	1	38	1820	28619	151100	194188	62485	4463	166	442879

Total number of generated paths: 430635
Total number of visited airports: 974
Google API calls: 54
Time needed for *MBFS*: 173 seconds

Hong Kong - Santiago :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 4
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 5 seconds
Number of *Hotspot Connections* found: 8

Step	1	2	3	4	5	6	7	8	Total
BCSA	0	10	3650	44998	122965	117228	26097	1241	316189
Steps	0	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	1	0	0	0	0	1
Rank	0	1	186	1449	3108	2902	763	30	8439
Too far	0	0	0	0	0	0	0	0	0
Dest. 1	0	0	1	12	1	0	0	0	14
Cycle	0	938	10228	37389	36078	10190	491	0	95314
Bad con.	235	15865	152235	411765	403256	93886	3895	0	1080902
Open paths	1	106	5170	51045	129982	121109	26926	1272	335610

Total number of generated paths: 324643
Total number of visited airports: 1009
Google API calls: 120
Time needed for *MBFS*: 152 seconds

Santiago - Hong Kong :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 4
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 6 seconds
Number of *Hotspot Connections* found: 11

Step	1	2	3	4	5	6	7	Total
BCSA	0	0	754	20142	73200	43421	2257	139774
Steps	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	0	0	0
Rank	0	2	229	2581	5020	2218	79	10129
Too far	0	0	0	0	0	0	0	0
Dest. 1	0	0	3	1	0	0	0	4
Cycle	0	207	4227	14483	10567	417	0	29901
Bad con.	122	4922	80914	264935	146995	6720	0	504486
Open paths	1	38	1478	24315	79246	45716	2336	153129

Total number of generated paths: 149907
Total number of visited airports: 840
Google API calls: 42
Time needed for *MBFS*: 72 seconds

Haikou - Puerto Montt :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 4
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 4 seconds
Number of *Hotspot Connections* found: 4

Step	1	2	3	4	5	6	7	8	9	10	Total
BCSA	0	2	293	14954	139169	452915	414308	89090	4637	58	1115426
Steps	0	0	0	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	0	0	0	0	0	0
Rank	0	0	0	44	1493	12695	13450	2399	152	1	30234
Too far	0	0	0	0	11	11	1	0	0	0	23
Dest. 1	0	0	0	0	4	2	1	1	0	0	8
Cycle	0	71	1751	30292	116267	119656	30070	2333	31	0	300471
Bad con.	47	1921	63387	503018	1558114	1321357	264360	14643	238	0	3727038
Open paths	1	12	626	19143	154167	476797	430036	91705	4790	59	1177335

Total number of generated paths: 1145691
Total number of visited airports: 1032
Google API calls: 20
Time needed for *MBFS*: 446 seconds

Puerto Montt - Haikou :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 4
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 3 seconds
Number of *Hotspot Connections* found: 4

Step	1	2	3	4	5	6	7	8	9	10	Total
BCSA	0	0	19	1810	34627	205520	620219	566373	113097	5811	1547476
Steps	0	0	0	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	4	0	0	0	0	4
Rank	0	0	0	31	1090	7136	44812	49558	10002	613	113242
Too far	0	0	0	1	36	371	583	82	0	0	1073
Dest. 1	0	0	0	0	3	7	2	0	0	0	12
Cycle	0	16	556	7615	45032	156878	187826	44259	3436	0	445618
Bad con.	5	262	8559	132063	719898	2149614	2002522	406224	22004	0	5441146
Open paths	1	4	83	2712	40765	226452	677692	618711	123285	6424	1696128

Total number of generated paths: 1661807
Total number of visited airports: 1034
Google API calls: 16
Time needed for *MBFS*: 676 seconds

A.3.2. Experiment 2

This experiment investigates the impact of different termination criteria

Arusha - Porto Alegre :

Departure date: April 5th 2018
Number of origin and destination airports: 1
Result size: 10
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 7 seconds
Number of *Hotspot Connections* found: 4

Step	1	2	3	4	5	6	7	8	9	Total
BCSA	0	0	356	14862	96833	223438	203742	48764	898	588893
Steps	0	0	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	0	0	0	0	0	0
Rank	0	0	0	14	219	1430	2354	1048	49	5114
Too far	0	0	43	1304	6245	12812	8426	1527	27	30384
Dest. 1	0	0	0	0	5	8	10	0	0	23
Cycle	0	75	1715	21596	65657	75223	24673	1841	0	190780
Bad con.	64	1755	65858	385385	883671	711902	137591	3031	0	2189193
Open paths	5	12	680	18442	108599	242528	215857	51443	974	638535

Total number of generated paths: 624414
Total number of visited airports: 821
Google API calls: 35
Time needed for *MBFS*: 243 seconds

Göttingen - Coimbra :

Departure date: April 5th 2018
Number of origin and destination airports: 5
Result size: 10
Averrage speed: 150km/h
Price per hour: 64 Euro

Step	1	2	3	4	Total
BCSA	0	6	1077	797	1880
Steps	0	0	0	0	0
Dest. 2	0	0	0	15	15
Rank	0	0	0	32	32
Too far	0	35	1021	104	1160
Dest. 1	0	0	14	3	17
Cycle	0	93	147	0	240
Bad con.	293	6369	2669	0	9038
Open paths	3	59	2111	948	3118

Total number of generated paths: 3104
Total number of visited airports: 316
Google API calls: 76
Time needed for *MBFS*: 18 seconds

Cape Town - Asuncion :

Departure date: April 5th 2018
Number of origin and destination airports: 5
Result size: 10
Averrage speed: 150km/h
Price per hour: 64 Euro

Time needed for *Hotspot Search*: 3 seconds
Number of *Hotspot Connections* found: 4

Step	1	2	3	4	5	6	7	Total
BCSA	0	9	1383	14979	19417	3131	246	39165
Steps	0	0	0	0	0	0	0	0
Dest. 2	0	0	0	0	3	0	0	3
Rank	0	0	1	85	329	163	22	600
Too far	0	4	821	2687	1851	200	13	5576
Dest. 1	0	0	0	8	10	3	0	21
Cycle	0	470	2810	4797	2261	447	0	10785
Bad con.	289	8869	62623	76463	13394	569	0	161918
Open paths	1	48	2519	18264	21795	3539	281	46446

Total number of generated paths: 45365
Total number of visited airports: 606
Google API calls: 69
Time needed for *MBFS*: 32 seconds

A.3.3. Experiment 3

Runtime :

Connection	MBFS	Hotspot Search
Cape Town - Asuncion (MBFS)	149	0
Cape Town - Asuncion (HS + MBFS)	144	4
Arusha - Porto Alegre (MBFS)	756	0
Arusha - Porto Alegre (HS + MBFS)	732	5
Arusha - Balmaceda (MBFS)	464	0
Arusha - Balmaceda (HS + MBFS)	318	5
Haikou - Balmaceda (MBFS)	1405	0
Haikou - Balmaceda (HS + MBFS)	1010	4

Virtual Costs :

Connection	MBFS	Hotspot Search
Cape Town - Asuncion	4443	5290
Arusha - Porto Alegre	5200	7993
Arusha - Balmaceda	8111	8546
Haikou - Balmaceda	5498	8086

A.3.4. Experiment 4

Runtime :

Connection	1 result	5 results	10 results	20 results
Göttingen - Coimbra	16	18	21	21
Cape Town - Asuncion	17	26	30	35
Arusha - Porto Alegre	329	405	648	1025

A.3.5. Experiment 5

Runtime :

Connection	1 airport	5 airports	10 airports	20 airports
Göttingen - Coimbra		17	77	141
Cape Town - Asuncion	21	26	32	36
Arusha - Porto Alegre	25	382	559	703

Virtual costs :

Connection	1 airport	5 airports	10 airports	20 airports
Göttingen - Coimbra		973	696	696
Cape Town - Asuncion	2167	2167	2167	2167
Arusha - Porto Alegre	4982	4339	4254	4254

A.3.6. Experiment 6

Runtime :

Connection	0 Euro per hour	64 Euro per hour	1000 Euro per hour
Göttingen - Coimbra	22	16	16
Cape Town - Asuncion	21	25	25
Arusha - Porto Alegre	440	339	221

Price for the connection :

Connection	0 Euro per hour	64 Euro per hour	1000 Euro per hour
Göttingen - Coimbra	98	138	437
Cape Town - Asuncion	541	541	541
Arusha - Porto Alegre	880	1194	1519

Duration required by the connection :

Connection	0 Euro per hour	64 Euro per hour	1000 Euro per hour
Göttingen - Coimbra	18	13	12
Cape Town - Asuncion	25	25	25
Arusha - Porto Alegre	63	48	45

Terminated paths because of too high virtual costs :

Connection	0 Euro per hour	64 Euro per hour	1000 Euro per hour
Göttingen - Coimbra	2923	2453	2031
Cape Town - Asuncion	3704	3399	2432
Arusha - Porto Alegre	27036	28314	12809

A.3.7. Experiment 7

Runtime :

connection	100 hm/h	150 km/h	200 km/h	250 km/h	300 km/k
Göttingen - Coimbra	21	17	13	15	15
Cape Town - Asuncion	252	26	10	9	8
Arusha - Porto Alegre	1551	382	62	16	9

Virtual costs :

Connection	100 hm/h	150 km/h	200 km/h	250 km/h	300 km/k
Göttingen - Coimbra	974	974	974	974	974
Cape Town - Asuncion	2167	2167	2167	2167	2222
Arusha - Porto Alegre	4254	4254	4254	4385	8131

Number of paths that were terminated because the destination was too far compared to the duration :

Connection	100 hm/h	150 km/h	200 km/h	250 km/h	300 km/k
Göttingen - Coimbra	893	1112	1110	1061	966
Cape Town - Asuncion	19600	6997	2506	1204	756
Arusha - Porto Alegre	32389	31389	30542	6099	4299