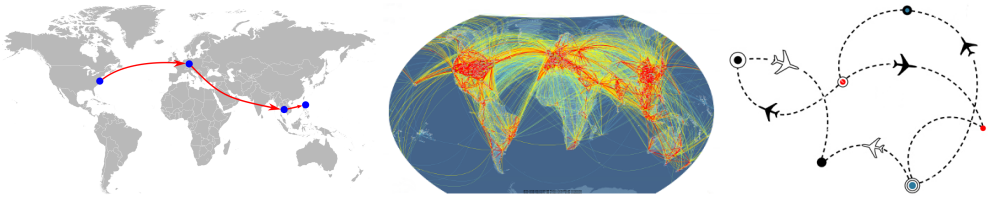
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2018
http://excel.fit.vutbr.cz

Route Planning in Air Transport

Marek Sychra*



Abstract

In this paper we present the problem of multicriteria route planning in air transport and show how it can be handled. It is a problem related to the common problem of finding the shortest path in a graph but with considering more criteria and most importantly, time. We focus on two state of the art methods CSA and RAPTOR and implement them along with some optimisations. Then we test the performance of these methods on real world data, which were supplied by Kiwi.com. We prove the expected pros and cons of both methods by experiments and backed by these results we propose a system for fast routing across the whole world.

Keywords: route planning – public transport – flight graph – CSA – RAPTOR

Supplementary Material: *N/A*

*xsychr05@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Since the dawn of transportation when people were able to travel between places, they also needed to plan their journeys, in order to spend their time effectively or to even get to the desired place at all. First, it started with a single point to point navigation, in static environment (it didn't matter whether you travel in the morning or the next day in the evening, the roads were always the same). But as time went by, public transport started to emerge in a way it's not even possible to imagine life without it. People at first had timetables for their planning but had to remember how exactly should they travel and had to connect the connections by themselves. In the past decades, with the growth of internet possibilities, automatic route planners started to appear. And soon, possibilities for having your journey planned by an internet service were present. While route planning in common means of transport (buses, trains) is working quite fine, in air transport it's still in the beginning.

The aim of this project is to create a route planning service which can handle user requests for pairs of destinations across the whole world in real time. Kiwi.com, which backs up this project has its own solution, which relies heavily on preprocessing. The problem of preprocessing and air transport is that flights and their fares can change quickly (unlike trains), so an older result can become invalid in time. The main objective of this service is to reduce the preprocessing part as much as possible (into the units of minutes) while wisely choosing the compromises which will have to be made. It should be able to produce a valid answer in at most a matter of seconds without an extensive preprocessing.

At the first sight, the problem seems like nothing more than a shortest path problem. However, the major difference is the time aspect of the problem, which makes the connection graph dynamic. The input information is not only a source destination, but also a time range interval for when a person wants to start their trip. Second difference is the fact that edges don't have a single value. People planning their trip might prefer

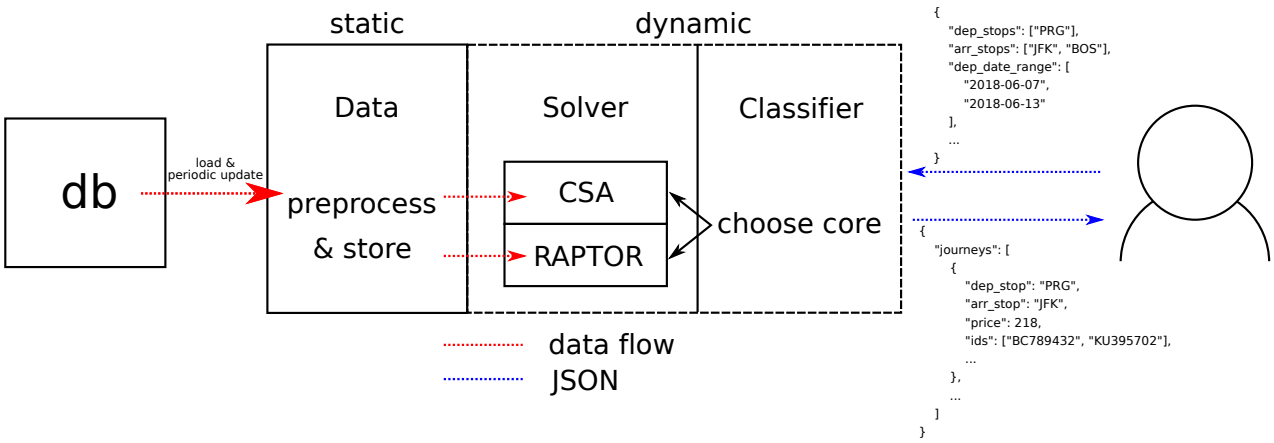


Figure 1. The scheme of the proposed service.

shorter journeys to cheaper or vice versa. That's why it's becoming a problem of multicriteria optimisation. We dig deeper into current approaches for tackling this problem in Section 2.

Our service starts with loading the data from a given source, scans it, filters the flights for duplicates and stores the flights in custom structs. Then the preprocessing starts, which consists of building a static graph which is used for A*-like cutoffs of distant destinations. After analysis of the flight graph and current user requests, we decided to try two algorithms for the core of the service. Those being Connection Scan Algorithm (CSA) and Round-bAsed Public Transit Optimized Router (RAPTOR). So when a user request with various filters comes, the solver core is queried with it and produces a JSON response with an array of connected journeys within seconds. The whole process can be seen in Figure 1.

We've performed a set of experiments to compare these two methods and determine the effect of proposed optimisations on the performance. The results show that CSA is much less influenced by the complexity of the request as opposed to RAPTOR. The latter shows extremely fast responses with more straightforward (and also more common) requests, but is much slower at handling more complicated ones. Our proposed system utilizes these facts and after classifying the request it chooses the right solver.

2. Related Work

The problem of route planning is commonly divided into three variations: earliest arrival problem, range problem and multicriteria problem. The first, given a source and a timestamp, finds only the time of the earliest possible arrival to the target node. The second extends the input timestamp into a time interval (for each timestamp from the interval it solves the first problem). Finally, the last changes the single criterion (earliest arrival to a destination) into a set of criteria. Since the last is the most complicated, includes the complexity of the previous and finally is the crux of this work, we shall describe only those methods that are able to handle this one.

The problem of route planning in dynamic networks is widely connected to the shortest path problem in a static graph. This is backed by the fact that it can be solved by a modified version of the generally known Dijkstra's algorithm for finding the shortest path in a graph. But first, the dynamic graph has to be transformed in order to be applicable for this static algorithm.

There are two main approaches for modeling the graph: time-expanded [1] and time-dependent [2]. The first models each time event (arrival at stop, transfer) as a node with a timestamp and connects those that are chronological by an edge. The edge linking might incorporate various constrains, such as minimal and maximal transfer time. This, however, results in a large graph. The second approach creates for each stop a set of nodes representing the outbound routes. Then those nodes that are reachable by at least a single connection are connected by an edge. The cost of every edge is determined by a travel-time function, which reflects not only the travel time itself, but also the waiting time.

This modeling allows the use of several graph algorithms. One being the modification of the algorithm mentioned before, Layered Dijkstra [2], which can only be used when the second criterion is discrete. At each node instead of a single value, it stores a time function which maps the departure times from source node to the travel time to this node. Every time an edge (u, v) is scanned, it first merges the time function at u to the edge. Then it tries to merge the result to the node v. Every time a better arrival time is added, the node is to be scanned again. To incorporate the second criterion, the graph is copied into K layers (K

being e.g. the maximum transfer count) and each edge is rewired into the higher level.

The graph-based methods, however, were proven not to fulfill the needs of larger networks [3, 4, 5]. The first response to this need was RAPTOR [4], which does not use an underlying graph, but rather just stores whole routes by the stops through which they go. At each stop it stores a set of non-dominating labels (consisting of criteria) and each time a connection to this node is scanned, it tries to merge the new label to the set. The algorithm works in rounds - round k computes journeys with $k-1$ transfers. At each round it scans the active nodes that have improved in the previous round (in the first round the only active is the source).

Another method is CSA [3], which is similar to RAPTOR, but works with basic connections. It orders the whole timetable by the departure time and for each query it scans every connection at most once. This results in excellent memory locality for the price of the possibility of scanning unwanted connections. Same as RAPTOR, it also stores labels at each stop so that it can handle multiple criteria.

A very intriguing approach shows out to be Transfer Patterns [6]. It is based on the idea that between every two stops there are only certain possible ways. This massively reduces the scanned space and eliminates the problem of finding out the right transit stops. However, the cost of the precomputation is extremely high, making it unsuitable for our cause.

All authors of the methods mentioned above have experimented on train or bus routes, but none have tried it on a world flight network.

3. Air Transport Router

Our proposed service is able to respond to user requests, which contain the following: departure airports, arrival airports, departure dates, number of results, ordering and limits for price, transfer count, transfer times and total duration. The result journeys can be ordered by price, by duration or by quality (the quality function was supplied by Kiwi.com).

The service can be divided into three main parts (as seen in Figure 1):

Data Manager

The first part is about data loading from given source and its preprocessing. It transforms the local times and stop ids into more convenient form. During the load it saves the data needed for building three static graphs. The following procedure showed on price is analogous also for segment count and duration. While loading connections, it saves the cheapest price between every two stops (if a connection exists) across the whole time range. Then, it builds a static graph using these values as edges. Next, it computes the shortest paths between every two stops using plain Dijkstra and stores it for further use (described in Section 3.1). The data manager also handles the periodic update of the data (as well as the static graphs) so that it's as fresh as possible.

Solver core

The solver contains the implementation of two algorithms - CSA and RAPTOR. For the sake of brevity we shall not describe the algorithms in detail, but we encourage the reader to have a look at them (see [3, 4]). Before the request runs one of the cores, it gets the allowed routing stops from the preprocessed data. The solver is not parallel (even though at least RAPTOR could be), due to the fact that we want to serve many requests at a time and parallelism would not, in fact, help anything.

Both algorithms store a profile at each stop which consists of bags of Pareto-optimal labels. The criteria are price, transfer count and departure and arrival times. One label can dominate other iff it is better in all criteria (cheaper, fewer transfers, departs later and arrives sooner). The profile is split into the bags by the arrival time so that when a connection is scanned, it finds the appropriate bag, extends all labels and tries to merge the new labels (using domination) into the corresponding bag at arrival stop's profile.

Classifier

After the experiments showed that certain requests suit one of the methods better, we decided we should exploit this fact and create a new layer before the actual solver - a classifier. Its purpose is to decide which algorithm should be run, based on the information in request. However, this part is not implemented yet, but we plan to train a simple decision tree using real user requests that we have at hand.

3.1 Optimisations

We incorporate an A*-like optimisation in order to determine the space where it is still relevant to route. We do so using two heuristic functions at two places. Every time it's used three times - for price, duration and transfer count (again, it's analogous, we'll describe it only once). The array of shortest paths (using the least possible value across the whole time interval) between every pair was built during the preprocessing phase. First function is constant for every pair and returns the limit for given metric (ineq. 1). This cuts off the places from where it would be impossible to get to the

target destination. Second heuristic function returns twice the lowest distance (ineq. 2). This tries to cut off the parts that could produce results not appealing for the user. The algorithm does not find the next connection for extension this way as this optimisation only temporarily removes the distant nodes from the graph. Possible space reduction can be seen in Figure 2.

$$dist(s,x) + dist(x,t) \leq limit \quad (1)$$

$$dist(s,x) + dist(x,t) \leq 2 * dist(s,t) \quad (2)$$

The first time the cutoffs are realized is once a request comes. It takes the source node s and target node t and then for all the remaining nodes x it tries if the equations hold. All nodes have to satisfy all six equations (for all metrics) in order to be routable through. The second time it's used is before a possible label extension. The value of $dist(s,x)$ is computed as follows: it extends the current cost with the connection's cost. The other value is computed as in the first case (x is the connection's destination node). If at least one equation does not hold, label can't be extended with this connection.

Second optimisation is a simple target pruning. However, the impact is affected by the *limit* parameter, which decides the number of final journeys. At the grouped arrival stops we keep a heap of Pareto-optimal labels ordered by the input *order* parameter. Every time a label is to be inserted into this heap, it must be undominated and better than the last.

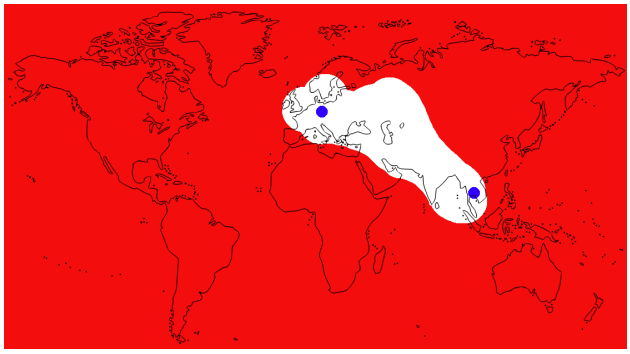


Figure 2. The impact of space reduction on route Prague - Bangkok. Stops in the red area won't be taken into account at all.

4. Implementation Details

Our service was written in languages C++ and Python3. The server, network communication and request distribution were written in Python3 using library asyncio[1], which allows proper asynchronous code execution without having to use threads. The computantionally intensive parts were written in C++ to ensure high performance. We used lightweight library pybind11[2] as the connection layer between these two languages.

5. Experiments on Flight Graph

We ran our experiments on Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz with 12 CPUs, 256GB RAM running Ubuntu 16.04. The dataset we used contained 1.57 billion connections among 3100 airports in the date interval 2018-04-01 – 2018-05-20. The data was supplied by Kiwi.com.

The main point of the experimenting was to compare the two methods and chose (if possible) the better for production usage. However, results have shown that neither of them is better than the other in all cases. We took 1000 various real user requests, sent them to both cores and tracked the response times. In Table 5 we can see the results. We can see that RAPTOR prevails in the total number of faster requests, however, in the cases RAPTOR is slower, the response times are much higher. This shows CSA is much more suitable for complicated requests, which take longer time in general, whereas RAPTOR is perfect for the simple (and also more common) ones.

	% of faster responses	Avg abs. speedup	Avg rel. speedup
CSA	33%	911ms	3x
RAPTOR	67%	82ms	19x

Table 1. Results showing comparison between CSA and RAPTOR response times.

As we were proven that certain request suit one of the cores better, we wanted to be able to decide this fact before actually processing the request. This is best done by some form of machine learning (decision trees), but for the sake of illustration, we present an experiment which shows how both algorithms react to extension of the *dep_date_range* parameter. We ran this experiment under similar conditions as the one before, but scaled departure interval from one day up to one week.

In Figure 3 we can see that CSA handles longer departure interval much worse than RAPTOR, which makes the latter a perfect choice for requests with wider departure interval. This inequality is caused by CSA not grouping connections. While RAPTOR scans connections departing from only certain stops, CSA scans everything. And if we assume the connection count grows linearly with time, the difference in scanned connection count is increasing rapidly.

[1] https://docs.python.org/3/library/asyncio.html

[2] https://github.com/pybind/pybind11

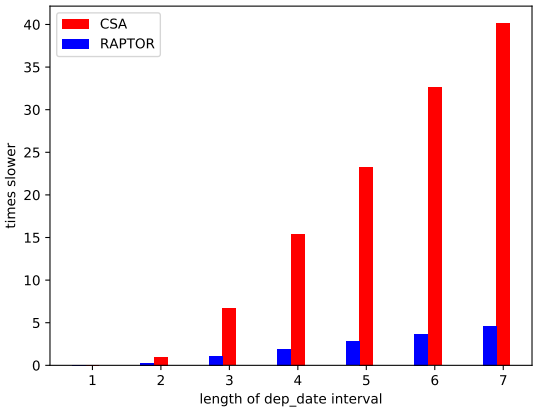


Figure 3. The impact *dep_date_range* extension on performance. Values are relative to response time for $dep\_date\_range = 1$.

As we stated, we also wanted to prove the optimisations improve the overall performance. First experiment was supposed to show how much does pruning improve the response time and how is it affected by the *limit* parameter of request. We took 100 real requests and sent them to both algorithm cores each time with changed *limit* parameter, which scaled from 2^1 to 2^{11}. We ran the experiment 10 times in total, computed the average values and plotted the increase of response times. In Figure 4 we can see the result.

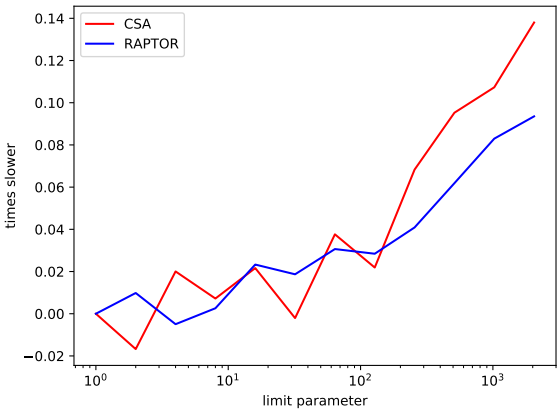


Figure 4. The impact of limit parameter on performance. Values are relative to response time for $limit = 1$.

For small values of *limit* the increase is imperceptible, but for higher values the slowdown is significant. We can think of disabled pruning as infinitely high *limit*, which is, of course, undesirable.

6. Conclusions

We presented a complex system for planning routes in air transport. For the solver core we chose two algorithms, CSA and RAPTOR, which we compared against each other using real world data and real user requests. We also included two optimisations, target pruning and A*-like space reduction.

Results have shown that common requests could be divided into two groups by ratio 1:2 where one algorithm outperforms the other in the same group. We show that CSA is much better when handling complex request and RAPTOR is extremely fast when answering simple ones.

However, there is no single criterion which would divide the requests, so we propose a classifier that would solve this task based on learned knowledge. The classifier was not implemented yet, but during the next weeks it will be, thus making the system complete.

Although a correct approach is crucial, precise work with memory and correct choice of data structs can improve the performance a lot when working with big data. In the following months we're going to focus on this field in order to make the system competitive among other proprietary systems.

Acknowledgements

I would like to thank my supervisor Ing. Zbyněk Křivka, PhD. for guidance and Mgr. Jan Plhák for an introduction to this topic.

References

[1] Stefano Pallottino and Maria Grazia Scutella. Shortest path algorithms in transportation models: classical and innovative aspects. In *Equilibrium and advanced transportation modelling*, pages 245–281. Springer, 1998.

[2] Gerth Stølting Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries. *Electronic Notes in Theoretical Computer Science*, 92:3–15, 2004.

[3] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly simple and fast transit routing. In *International Symposium on Experimental Algorithms*, pages 43–54. Springer, 2013.

[4] Daniel Delling, Thomas Pajor, and Renato F Werneck. Round-based public transit routing. *Transportation Science*, 49(3):591–604, 2014.

[5] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. Route planning in transportation networks. In *Algorithm Engineering*, pages 19–80. Springer, 2016.

[6] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *European Symposium on Algorithms*, pages 290–301. Springer, 2010.