technische universität
dortmund

Bachelor thesis

Multi-modal Route Planning with Dynamic Public Transport Delays

Sebastian Peter

February, 2017

Reviewers:
Prof. Dr. Katharina Morik
Dr. Thomas Liebig

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS VIII)
http://www-ai.cs.uni-dortmund.de/

Contents

Chapter 1

Introduction

1.1 Background and motivation

The trend of increased congestion and traffic jams in urban areas currently leads to rising needs for situational aware routing software [12]. Common journey planning systems already consider realtime delay information of public transfer services. This plays an important role in improving the quality of computed journeys for the end user, as delays in public transportation may frustrate journeys that have been generated statically. Models for predicting future traffic situations have been proposed many times, such as using Spatio-Temporal Random Fields [12].

In conventional dynamic routing systems, delay information not only help improving accuracy of regular journeys, but furthermore also allow for alternative journeys to emerge, which would have not been computed neglecting delay information. This phenomenon is, among others, caused by regular journeys being made infeasible by delays exceeding the waiting time between two trips, thus making the imaginable user miss a specific transfer. The regular journey being blocked makes space for alternative journeys, which are only dominating given the specific delay scenario.

Regarding routing algorithms in public transport, transfer patterns represent a data structure and algorithm improving performance by several orders of magnitude compared to many available algorithms. Transfer patterns are based on the assumption that many optimal journeys share the same sequence of transfer stops in the course of a day. The data structure around transfer patterns thus only includes stops representing origin, target and eventual transfer stops of the journey, omitting all intermediate stops. During query time, the complete route including intermediate stops and concrete departure and arrival times at each stop are reconstructed using direct connection tables [4].

Combining both fast routing queries and updated traffic information describes an ongoing challenge of research in this area and is thus topic of this thesis. Bast et al. propose including delay information at query time by associating the delay data with corresponding trips in the direct connection table. This simple approach alone already yields good results for many routes compared to other path finding algorithms [2]. The approach of this thesis goes in a different direction: during precomputation, trips of each transfer pattern are artificially delayed in order to make space for alternative transfer patterns. These new patterns are then classified in regard to the circumstances that led to their creation, specifically the selection of delayed trips and the amount of delay of each.

This thesis describes an implementation of dynamic transfer patterns in Open Trip Planner, an Open Source client-server routing software. While OTP already calculates routes considering multiple criteria like accessibility and mode of transfer, its internal Dijkstra algorithm is not optimal in terms of speed. Furthermore, a routing service for the city of Warsaw handles over ten thousand queries per day and would thus benefit by improved algorithm speeds.

1.2 Structure

The next chapter deals with essential concepts of routing in public transportation and road network graphs, including graph models, basic algorithms and specifications of realtime formats. Next, the concept of transfer patterns with delay classification is outlined, followed by details about its usage and implementation in OTP. In the end we will present test results, followed by general conclusions and inspirations for future work.

Chapter 2

Preliminaries

2.1 Routing in road networks

In order to do routing on the road or public transportation, we need to define a directed graph $G = (V, A)$ first with a set of vertices V and a set of arcs A. There is a weight assigned to each arc, denoted by $\mathrm{l}(u, v)$ for the arc between nodes u and v. Here we are considering the point-to-point shortest path problem: given a graph G, a source station $s \in V$ and a target station $t \in V$, a sequence of arcs leading from s to t with the lowest possible weight needs to be found [5].

Dijkstra A standard solution to this problem is using Dijkstra's algorithm, published by Edsger W. Dijkstra in 1959. Given the graph $G = (V, A)$ and $s, t \in V$, it initializes a queue of nodes $Q = V$ and a distance function dist with $\mathrm{dist}(s, s) = 0$ and $\mathrm{dist}(s, v) = \infty$ for all $v \neq s, v \in V$. Then, until the queue is empty, the main loop is executed: The node u with the smallest distance $\mathrm{dist}(s, u)$ is picked and removed from Q. For each neighboring node v of u, the label of v is updated if the former distance $\mathrm{dist}(s, v)$ is greater than the new path way $\mathrm{dist}(s, u) + \mathrm{l}(u, v)$ using node u [9].

Dijkstra's algorithm can be extended to return shortest paths by extending each node's label with the arc leading the node. By walking through the nodes in reverse after the algorithm has finished, the shortest sequence of arcs from source to target node can be reconstructed.

The algorithm can be sped up by running two instances simultaneously from both s and t until a common node u is hit. This bidirectional search provably finds the shortest path from s to t over u and uses considerably less search space [14].

A* While Dijkstra picks the node with the smallest distance from the current node, goal-directed approaches like A* tend to decrease the bound on running time by using a heuristic $\pi : V \to \mathbb{R}$. π is a lower bound on the remaining distance to the target $\mathrm{dist}(u, t)$.

Dijkstra's algorithm is then adapted to pick the node with smallest $\text{dist}(s, u) + \pi(u)$ with each loop iteration. This way, nodes that are closer to t get picked earlier during execution of the algorithm [4].

Contraction Hierarchies In order to speed up the query time, which is the time needed to solve a particular point-to-point shortest path problem, several techniques involving precomputation have been invented. One of these are Contraction Hierarchies, which use shortcuts to bypass unimportant nodes in order to reduce the overall number of nodes visited during query time.

The importance here is a measure used during precomputation, with the overall goal of adding the fewest new arcs possible. After all nodes have been sorted by importance, the algorithm repeatedly picks the least important node and connects its neighboring nodes with new arcs. During query time, CH uses a bidirectional search that only visits nodes with increasing importance. The algorithm stops when a most important common node u^* is met by both searches [10].

2.2 Routing with public transport

Widespread concepts of graphs modeling public transportation timetables incorporate a temporal layer besides spatial information, which represent the only component in graphs describing road networks. The temporal information here model an inherent restriction of public transportation: vehicles can only be boarded at specific times.

A trip T serves a sequence of stops $\mathit{stops}(T) = (s_1, ..., s_n), s_i \in S$. Thus T connects two stops s_a and s_b if and only if $\mathit{stop}(T, s_a) < \mathit{stop}(T, s_b)$. Multiple trips form a line if they contain the exact same sequence of stops. Timetables are valid for specific traffic days, which repeat themselves throughout the year [2].

Routing queries used for road transportation need some adjustments in order to be usable for public transport. The source and target nodes are now stops s_s and s_t, and routes have a departure time τ to consider, in short $s_s@\tau \rightarrow s_t$. If we want to specify the trips taken in such a route, we denote $s_s \rightharpoonup s_u \rightharpoonup s_t$ for a two-trip route from s_s to s_t with a transfer at s_u. With a multi-modal approach, trips using different modes of transportation (such as walking or biking) are available as well, and thus routes can have source and target nodes on street level.

When computing routes in public transportation, a number of requirements play a role in deciding for an optimal algorithm. Preferably small computation time and data size of precomputed data, short query times, consideration of real time data and multi criteria optimization (including criteria like travel time, number of transfers, fares etc.) are of importance [1].

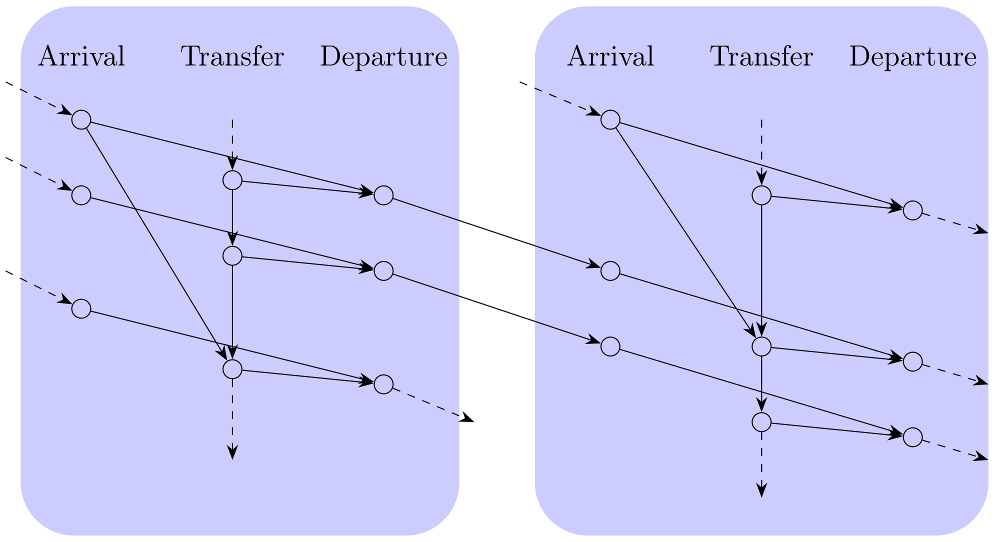


Figure 2.1: An example of modeling timetable data with a realistic time expanded model

Over time there have been developed two standard approaches describing a public transport network: the time expanded and time dependent model.

Time Expanded Model In the time expanded model, the graph is constructed by creating a node for each distinct event that occurs in public transport. The arcs represent either an elementary connection between two stops or waiting at a stop. In this way, the spatial graph of stations is duplicated for each distinct trip and interconnected by arcs whenever a transfer is possible. Since timetables repeat itself in a known time frame, the last nodes are linked to the first ones again by arcs.

An extended version of this model splits stop nodes into arrival, transfer and departure nodes in order to provide a data structure modeling individual transfer times between trips. Here, arrival nodes are directly linked to departure nodes if it is possible to stay on the vehicle at the station. Arrival nodes also have an arc to the next transfer node that the vehicle can be descended at. Considering transfer time, each transfer node links to the next possible departure node, making it possible to ascend the corresponding trip. Another arc leads to the next transfer node in time, modeling waiting at the stop.

Time Dependent Model The time dependent model allows for smaller graph size by representing each stop with one node only. An arc connects two stops in a given direction if and only if there exists at least one trip connecting the stops in that direction. A travel time function is assigned to each arc, returning the weight and the travel time of the trip for any given point in time.

This model has been extended as well to include transfer times in a more realistic way. The stop nodes of the simple approach here link to a route node for each line that stops

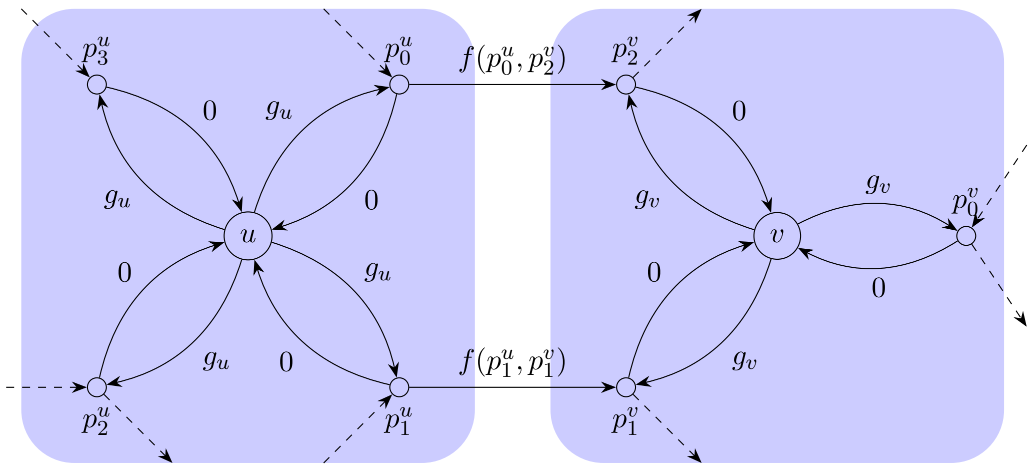


Figure 2.2: An example of modeling timetable data with a realistic time dependent model and constant transfer time per station

there. The (constant) transfer time is then assigned to the ascending arc, meaning the arc that directs from the stop node to the route node [15].

Frequency Based Model The frequency based model differs from the time dependent model only in its approach of representing the travel time function. Because in reality trips usually depart in specific frequencies for different intervals each day, using an algorithm to find the longest arithmetic progressions, the data can be easily compressed as the time of the first departing trip τ_{dep}, a time interval Δ, and a frequency f. All departure times can then be computed with $\tau_{dep} + fi$ for all $i \in \mathbb{N}$ that satisfy $fi < \Delta$ [3].

In order to solve multi-modal problems, graphs of road networks and of public transport are combined by linking stops to their respective nodes at street level.

Pareto optimality Routing in public transportation offers a handful of criteria to consider when choosing optimal routes, predominantly earliest arrival, number of transfers and fare costs. Each criteria is measured as a component of total costs, for which Pareto optimality can be used for comparison. A route a is Pareto optimal (or dominating) if and only if all cost components of a are never beaten by corresponding components of all other routes. A Pareto set thus describes a set of routes which draw a tie against each other (neither $a < b$ nor $b < a$ holds true) [5].

Dijkstra variants An obvious approach to compute for a query $s_s@\tau \to s_t$ on a time expanded graph model is to use Time Expanded Dijkstra (TED): a modification of Dijkstra starts at the first node of s_s that departs after τ and stops once an arrival node of s_t is reached [17]. Time Dependent Dijkstra (TDD) is the name of a modification running on time dependent graphs. It traverses the graph in a straight-forward way while keeping track of travel time in order to retrieve correct costs from travel time functions of arcs [5].

For Dijkstra to handle multiple criteria, a list of labels is attached to each node, containing costs for each criterion to be considered. Each potentially faster path to a node s_i is only inserted if not dominated by any existing set of labels of s_i. This approach is called Multi-Criteria Label Setting (MLS) [13].

Connection Scan Algorithm CSA is a recent development that omits all efforts of building a timetable graph, but instead collects all trip departures in a single array, sorted by departure time. When answering a query $s_s@\tau \to s_t$, the algorithm starts at the array element of s_s at time τ and works its way through the array. If a consecutive connection can be reached by any preceding trip, its label is updated, until the element of s_t is reached [8].

RAPTOR Round-Based Public Transit Routing is an algorithm proposed by Delling et al. [6]. In a set number of rounds K, the algorithm scans public transport lines and updates the shortest path to each stop accordingly.

At the beginning, K labels $\tau_0(s), \tau_1(s), \ldots, \tau_K(s)$ are attached to each stop s, each set to ∞. At each round $k \in [1 \ldots K]$, the values of $\tau_k(s)$ are set to $\tau_{k-1}(s)$ for each stop s, functioning as an upper limit for arrival time. Then, each route r with a stop s that improved arrival time $\tau_{k-1}(s)$ in the last round is traversed. Throughout traversal, at each stop s, the next available trip t is searched, which is the one with departure time right after $\tau_{k-1}(s)$. Once a suitable trip is found, it is "ascended", meaning the next stops in line are updated with new arrival times of t until an earlier suitable trip is found, which is then used instead. At the end of each round, precomputed footpaths between stops are incorporated by updating $\tau_k(s)$ if a faster connection is found through walking from any station s' to s.

RAPTOR can be sped up with parallelization by traversing mutually exclusive subsets of routes only. Race conditions can be avoided here by precomputing a conflict graph of routes, making edits on stops of a route only possible once all dependent routes are processed. In order to solve the multi-criteria problem, McRAPTOR extends the algorithm implementing multiple labels per stop, each representing a criterion. The range problem is solved with the extension rRAPTOR, executing the algorithm for each departure time of the source stop within the requested range.

Transfer patterns The data structure and algorithm named transfer patterns has been proposed by Bast et al. and is considered state-of-the-art. Transfer patterns are based on the assumption that in one day, there are only a few optimal routes from s_s to s_t that solely differ in specific trips they use.

During preprocessing phase, Pareto sets of optimal routes between all stops are computed using a variant of Dijkstra. For each route, a transfer pattern is then created by

$$\begin{tikzcd}
 & & & s_u \arrow[d] & \\
s_s & s_y \arrow[l] & s_w \arrow[l] & s_x \arrow[l] & s_t \arrow[l] \arrow[lll, bend left] \arrow[llll, bend left=40]
\end{tikzcd}$$

Figure 2.3: Example data structure of transfer patterns

line L17	s_a	s_s	s_b	s_y	...
trip 1	8:15	8:22 8:23	8:27 8:29	8:38 8:39	...
trip 2	9:14	9:21 9:22	9:28 9:28	9:37 9:38	...
⋮	⋮	⋮	⋮	⋮	⋱

Figure 2.4: Direct connection line table

stripping away any temporal components as well as information about intermediate stations, leaving only a sequence of transfer stations. For each source station s_s, this data is stored in directed acyclic graphs ending in s_s [4].

Furthermore, for each line all trips are stored in a table containing arrival and departure time at each stop of the line. Also for each stop a list of lines that serve the stop is created, including its position in each line.

A query s_s@9:10$\rightarrow s_t$ is solved by retrieving the transfer pattern graph for source station s_s and building a query graph that only contains transfer patterns ending in target station s_t. Taking the DAG in figure 2.3 as an example, the pattern $s_s \rightharpoonup s_y \rightharpoonup s_w \rightharpoonup s_x \rightharpoonup s_t$ represents one possible route of this TP. From there, a route is reconstructed from the pattern. For each pair of stops a line is searched by intersecting the rows associated with both stops. Taking the trip $s_s \rightharpoonup s_y$ as an example, L17 is the only line connecting both stops (cf. figure 2.5). Now the trip table of L17 is queried, retrieving the first trip that departs after τ or after arrival of the previous trip. In our example trip 2 is the first option, departing from s_s at 9:22 and arriving at s_y at 9:37 (cf. figure 2.4).

s_s:	(L8, 4)	(L17, 2)	(L34, 5)	(L87, 17)	...
s_y:	(L9, 1)	(L13, 5)	(L17, 4)	(L55, 16)	...
⋮	⋮	⋮	⋮	⋮	⋱

Figure 2.5: Direct connection stop table

For multi-modal problems, queries can start or end with nodes on street level. Here, the closest stations to source or target node are searched and combined as route request using the Cartesian product.

Delay information A widely established format for public transport timetable information is called General Transit Feed Specification (GTFS). It encodes relevant information such as transportation line geographies, departure and arrival times and information about stops. Carriers can publish their timetable data as GTFS files, which then can be worked with by developers and researchers. In order to provide updated service information like delayed departure and arrival times, GTFS can be enhanced with a realtime extension. Through GTFS Realtime, trip updates concerning the timetable, service alerts on parts of the transport network and vehicle positions can be shared [7].

Chapter 3

Dynamic transfer patterns

Due to the static nature of transfer patterns, the inclusion of delay information is not trivial. Recent work of Bast has shown that updating the arrival and departure times in the direct connection table alone yields optimal results for a large set of routes [2] (see Figure 3.1 for an example). For a small subset of source and target pairs and certain delay scenarios though, transfer patterns do not replicate optimal routes. This is because optimality of these routes stems from non-optimality or infeasibility of the regular routes due to changed departure and arrival times.

Delays in public transportation has been classified before in terms of its variability [11]. Here, effects of delay scenarios on Pareto optimality are distinguished by investigating the relationship between delays and optimality: a non-dominating route can become optimal through delays inside the same route or inside other routes.

Delays inside a route, making it Pareto optimal, need to happen for a sequence of trips at the beginning (or at the end, if negative delays are considered). An example of a route becoming Pareto optimal could be illustrated as follows. Let there be a route $s_s \rightharpoonup s_u \rightharpoonup s_t$ with a waiting time of Δ_0 at s_u. It is also given that there exists another arbitrary route which is dominating due to shorter travel time. If the travel time of the former route is now altered by a delay of $\Delta_0 - \epsilon$ of the first trip, it may now become the optimal route. In praxis, these kind of cases are not worthy to consider though, as they are very unreliable: a vehicle that experiences a delay is likely to increase or decrease its delay again before arriving at the stop that is relevant to the user.

line L17	s_a	s_s		s_b		s_y		…
trip 1	8:15	8:22	8:23	8:27	8:29	8:38	8:39	…
trip 2	9:14 +3	9:21 +3	9:22 +3	9:28 +4	9:28 +4	9:37 +5	9:38 +5	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Figure 3.1: Updated direct connection line table

For this reason, the focus of this thesis is set on routes becoming optimal because of different (and formerly optimal) routes experiencing delays. Delayed routes then become either non-optimal or even infeasible if a delay is greater than the waiting time at the next transfer.

As a simple example of such a scenario, consider a route $s_s \rightharpoonup s_y \rightharpoonup s_t$ being made infeasible with a sufficient delay of the first trip. An alternative route $s_s \rightharpoonup s_v \rightharpoonup s_t$, which would have been dominated by the first pattern, now becomes favorable in the Pareto sense. Since regular transfer patterns are merely computed on a static graph and the advantage of the second pattern depends on a certain scenario of realtime delays, it could not be replicated using static transfer patterns. The new approach is to include alternative routes like above in the data structure by simulating delays during precomputation. In this thesis, these alternative routes are called *dynamic (transfer) patterns*.

3.1 Precomputation

In contrast to static transfer patterns, which can be computed in a straight-forward fashion, dynamic patterns require information about lines to be artificially delayed during precomputation. In order to achieve this, each static transfer pattern computation originating in s_s keeps a set D that is composed as follows. During computation of a route $s_s \rightarrow s_t$, each line l of a trip $s_i \rightharpoonup s_{i+1}$ with $s_{i+1} \neq s_t$ is saved with the corresponding transfer time θ as a tuple (l, θ) in D. If the same line already exists in D, the tuple with lower θ is eliminated. Once static transfer patterns are computed, dynamic patterns for s_s are searched.

All tuples in D are combined into a number of delay scenarios to be used in the next precomputation step. There are many imaginable combinations that tuples can be fused into. The most comprehensive way is to consider the power set $P(D)$ of all n tuples, which results in 2^n scenarios. This entails that all possible dynamic routes are computed, but also implies enormous computational costs.

Multiple ways of combining tuples were thus introduced, trying to cover many dynamic routes while keeping the overall number of scenarios as low as possible. A trivial approach is to incorporate delays of only a single line at a time. This method may already significantly increase computation costs for large graphs compared to merely computing static patterns. For this reason, picking a limited number of random tuples per transfer pattern subgraph was introduced as another approach. Lastly, in pursuance of computing the most useful dynamic routes, past data of lines with a high likelihood of delay can be utilized. This means picking often-delayed lines or combinations thereof more frequently when constructing a limited amount of delay scenarios.

Each delay scenario of the previous step is then applied to the graph and the graph search is repeated. A tuple (l, θ) of a delay scenario is applied to the graph by artificially increasing arrival and departure times of all trips following line l by $\theta + 1$. This way, the

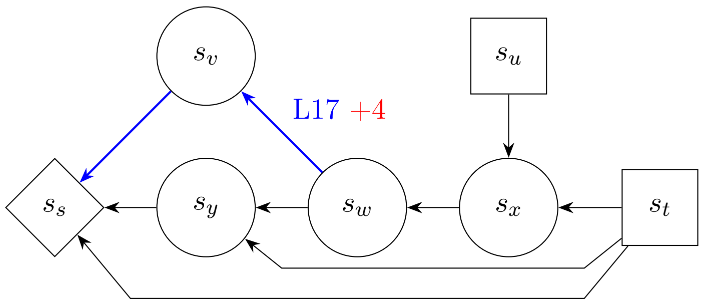


Figure 3.2: Example for transfer patterns with dynamic trips in blue

transfers after using a trip of line l in the static routes become infeasible, making room for dynamic patterns considering the delay situation.

All routes computed in the last step are merged into the static transfer pattern graph. The merge process inserts a new transfer $s_a \rightharpoonup s_b$ only if it doesn't already exist, otherwise it is discarded (see Figure 3.2). In case the transfer is inserted, it is classified in terms of the delay scenario that led to its existence.

3.2 Query time

When answering a routing query $s_s@\tau \rightarrow s_t$, the query graph for $s_s \rightarrow s_t$ is fetched in the same way regular transfer patterns are handled. All routes from s_s to s_t are reconstructed by traversing the query graph for s_s, starting at the node of target station s_t.

When coming across transfers that are attached with delay classifications, they are only considered if the classifications match the actual traffic situation at query time. A tuple (l, θ) matches realtime delay data if and only if there exists a trip of line l of which any arrival time is delayed by an amount θ_a with $\theta_a \geq \theta$. This constraint is rather hard and neglects cases in which the dynamic pattern could dominate despite the maximum delay θ not being reached. Thus, softer constraints such as $\theta_a \geq c\theta$ for a constant $c < 1$ were taken into consideration.

All resulting routes are compared regarding their transfers as well as arrival and departure times. Routes with less transfers, earlier arrival and later departure are always favorable. Since real time delay information has been applied to departure and arrival times, it is possible that some patterns need a much longer travel time or are completely infeasible and thus no longer interesting to the user. These patterns are discarded either when direct connections are fetched for all trips or when they are dominated by other patterns in the Pareto sense. Finally, the Pareto set of routes is returned, representing the result for the query.

Chapter 4

Implementation

4.1 Open Trip Planner

The exemplary implementation of dynamic transfer patterns in this thesis is built upon the Open Source software Open Trip Planner, which is a platform for multi-modal journey planning using the client-server model. OTP offers a web interface for handling user requests and displaying time-specific route information on an interactive map. The server framework is written in Java and works with OpenStreetMap map data and GTFS public transport tables[1].

OTP uses a time-dependent graph as its internal data structure, which can be constructed from any GTFS feed in combination with respective map data. A variety of `Vertex` subclasses model junctions of the road network as well as stops and stations on the public transport level. Derivatives of the `Edge` class represent the links between vertices, modeling differences in properties and behavior by overwriting abstract methods. One of these methods is the `traverse()` function, adding costs and elapsed time of an edge when visited by A*. On public transport level, stops are modeled by `TransitStop` objects, which itself are linked to `TransitStopArrive` and `TransitStopDepart` vertices, representing route nodes of the traditional time-dependent model. From there, vehicles can be alighted or boarded, respectively, using `TransitBoardAlight` edges. `TransitBoardAlight` edges handle temporal restrictions of boarding or alighting from a vehicle and increase travel costs accordingly. Traveling itself is modeled by `PatternHop` and `PatternDwell` edges, representing a move from stop to stop and staying inside a vehicle at a stop. As a preparation to answering multi-modal routing queries, the public transport graph is connected to the street network at every stop that allows transferring from or onto public transportation. Benefiting quick transfers between platforms or nearby stops, `SimpleTransfer` edges allow transfers within one single arc.

[1]http://www.opentripplanner.org

From the OTP client interface the user can pick source and target location, the date and time that he wishes the journeys to arrive or depart at and the modes of transfer to be used. Additional options include restricting the routes to be accessible or to declare certain lines as preferred or to not be used by the routing algorithm. After the user has submitted and the server has received the request, a number of fastest routes that arrive before or depart after given point in time are calculated on server-side using an implementation of A*. The A* algorithm works with different heuristics depending on the requested modes of transfer. If public transport is enabled in query parameters, a bidirectional heuristic is used.

OTP is shipped with an implementation of RAPTOR as well, which is typically used for analysis purposes only. It can handle only one-to-all queries and thus takes more query time despite of RAPTOR's inherent advantages in algorithmic complexity.

4.2 Precomputation

As transfer patterns are a novel approach to routing, it was not realized in OTP yet. In order to provide an alternative algorithm for computing multi-modal routes, algorithms and data structures of TP were implemented and integrated into OTP in the course of this thesis. In the following section I will explain the steps required to compute transfer patterns, as well as the underlying data structure and implemented algorithms.

Setup A couple of preparation steps need to be executed before transfer patterns can be used. After being exported as a `.jar` file with Maven, an OTP graph object of given area needs to be built using parameter `--build` *`graphDir`*, where *graphDir* is a directory containing map and GTFS files. Each instance of OTP needs an indication of a working directory using `--basePath` *`dir`*, with *`dir`*`/graphs` being the future directory for complete graph objects. Since compilation of graphs and TP can require large amounts of memory, the Java Virtual Machine needs to be configured to start with a bigger heap using `-Xmx`y`G`, where y is size of the heap in GiB.

Transfer patterns can then be constructed using `--buildTp`. Base path needs to be setup pointing at the graph directory, and the standard graph needs to be loaded using `--router` ''. `--autoScan` can be used as well in this context but leads to problems if the graph directory contains mulptiple graph objects. In order to enable parallelized precomputation, see the according paragraph below.

Algorithm First and foremost, a direct connection table in form of stops referencing all lines serving the stop is precomputed. With each line, the stop's index in the sequence of the line is saved. This way a route can be directly boarded during query time without needs to traverse it.

For precomputation of transfer patterns, the internal routing algorithm of OTP is used. Instead of one-to-one computation like in regular server mode, the precomputation algorithm uses a one-to-all routing approach. For each source station, the departure times of all trips leaving the station are compiled. Then, for each departure time a one-to-all routing search is started, from which a TP graph is built later on.

Each invocation of the A* routing algorithm returns a search tree structure containing shortest paths from s_s to every other stop. The search tree is then walked through in-order from target nodes to s_s, wrapping each stop with a `TPNode` object (cf. Figure A.1). Each TP node references a list of `TPTravel` objects, of which each encodes a journey to another stop and thus contains an (empty) delay classification. `TPTravel` objects can represent a transfer or footpath, which is encoded by a boolean variable. Sternisko's approach of avoiding duplicate TP nodes is also considered here: all intermediate nodes of a TP graph are stored in a map for later reuse [18]. On completion of precomputation, all transfer patterns are referenced as a field of the regular graph class and then exported by serializing the graph object.

Constraining complexity The precomputation of transfer patterns, especially including artificial delays, requires high computational effort. Since running the algorithm without constraints can take weeks, a number of measures were taken in order to finish computation in time for this thesis. Maximum distance of footpaths within a route had to be reduced to 500m and, similar to the implementation of Bast et al. [4], routes were limited to two transfers. Since the computation of fastest routes is repeated for each departing trip of the source station and every delay scenario, a lot of redundant routes are calculated and discarded in the process of forming transfer pattern graphs. The number of graph searches was thus limited to only run searches for departure times that were at least half an hour apart.

Dynamic patterns During the creation of the static transfer pattern graph, tuples of non-final trips and their respective waiting times to the next transfer are recorded. After static patterns are computed for a given stop s_s, delay scenarios are generated using these tuples and a specific delay builder. Delay builders are implemented as subclasses of `TransferPatternDelayBuilder` and dictate how delay scenarios are created: given all tuples of lines and their respective maximal transfer time, delay scenarios with arbitrary amounts of delayed lines are formed and returned as a set (cf. Figure 4.1).

With delay scenarios at hand a new routing search is started for each scenario and departure time. Each search returns a separate search tree, which is then integrated into the existing transfer pattern graph of s_s. In the course of this, new `TPTravel` arcs with their specific delay classifications are only added if such arcs do not exist as static patterns yet.

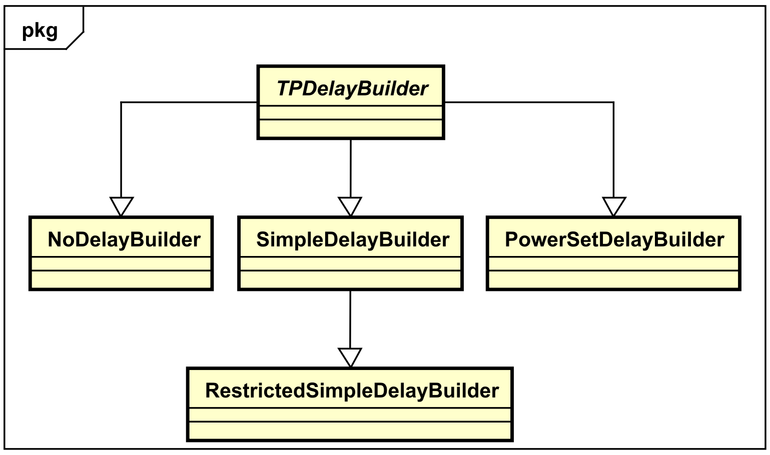


Figure 4.1: Class digram of delay builders

Parallelized precomputation Precomputing regular transfer patterns can be easily sped up using parallelization [4]. This is why splitting the computation task in smaller sub-problems of arbitrary number was worthwhile to consider. By starting OTP with option `--chunk` n/m (with $n, m \in \mathbb{N}$ and $0 < n \leq m$), precomputation can be configured to solely compute the nth chunk with m^{-1} times of full size TP. All chunks are solely computing patterns that start with mutually exclusive subsets of source stops. This is achieved by sorting all stops by index, establishing an order that is equal among all instances of a graph. For a graph with N stops, each chunk is restricted to stops with position $\left\lfloor \frac{N(n-1)}{m} \right\rfloor \leq i < \left\lfloor \frac{Nn}{m} \right\rfloor$.

Since dynamic patterns require alteration of the graph, parallelized computation with just one graph object in memory is not a viable option. In the end it was decided to start computation of each chunk in its own JVM. Once computation of a chunk is finished, its graph is serialized and saved to disk. In order to merge all graph chunks into one, OTP needs to be started with parameter `--mergeTp` m. All available graph chunks are then loaded and their transfer patterns are combined into a single graph. Loading all m graph chunks at once can lead to astronomic memory requirements of up to 300GiB when precomputing TP of Berlin. For this reason, the merging process was changed to release each chunk from memory after having been merged successfully.

Another obstacle that arose when merging transfer pattern graphs was that each TP chunk references objects in its own respective OTP graph. Merging two TP graphs thus resulted in duplicate OTP vertices and edges, making routing impossible if source and target stop were located in different chunks. During the merge process, only the OTP graph of the first TP graph is thus taken as a reference point. Whenever a subsequent graph is merged into the first one, all references to OTP objects are replaced by references to equivalent objects of the first graph.

Removing cycles During testing the computed transfer patterns in query time, it became apparent that in some cases the transfer pattern graph contained cycles, violating the transfer patterns' acyclic criterion and causing endless loops during query time. The most likely explanation here seems to be that OTP's implementation of A* might have an inclination towards computing routes that visit stops in disparate sequences at different starting times. Since transfer patterns are stripped of any temporal information, cycles might appear when one TP contains a trip from s_a to s_b and another a trip from s_b to s_a.

In order to cope with this phenomenon, a recursive graph cleaning algorithm `TransferPatternGraphCleaner` was introduced (cf. Figure A.3), removing all cyclic arcs of the graph. For each recursive call of the function, it is checked if the current node has been visited before. If so, the arc leading to this node is removed and the function returns to the previous level of recursion. If not, the current node is added to the set of visited nodes and the function is called again for all child nodes.

4.3 Query time

At query time, the following procedure is followed in order to construct routes for the given query. Minimum requirements for computing routes are a source and a target node, as well as a departure or arrival time. See Figure A.1 for a sequence diagram.

If source and target vertex do not represent a stop of the public transport layer, the closest stops surrounding the respective vertex are searched. For all combinations of source and target stations then transfer patterns are retrieved. In order to extract a single transfer pattern from source to target, the directed acyclic graph of given source stop is obtained from the TP graph. Then, a subgraph of the target node is returned in form of a `TPNode` representing the target. In a next step, the pattern's tree structure is unfolded into separate linear paths. Since the in this way reconstructed patterns might lack walking paths at the beginning or end, appropriate partial patterns might be added.

In the next step, concrete trips or walking paths are searched for each segment of the pattern. Trips are obtained from precomputed direct connection tables and walking paths are calculated using the internal A* algorithm. Since using Dijkstra for creating walking paths represents a bottleneck of this implementation of TP, footpaths are cached using `WalkingPathBucket`. This optimization helped cut query time significantly.

If no trip can be obtained for any segment of a given pattern, the pattern is discarded. This happens predominantly when no trip is active for the given time period, most of all during the night. Finally, a Pareto set of all routes is searched and returned as the query result. The Pareto set is built by adding the first route to a preliminary set, then comparing each further route to the all routes of the set. If a new route beats any of the former routes in set, the former route is replaced with the new route. In case a new route is Pareto-equal to all routes in set, it is added to the set (cf. Figure A.4).

In an earlier version of this implementation, routes were restricted to starting and ending with transfers and not footpaths. This was achieved by implementing an A* heuristic which disallowed walking in the beginning, as well as skipping all search graph paths which ended in a walking path. While this restriction was supposed to reduce the size of transfer patterns data, query times improved significantly when discarding it.

Once computation of all routes is complete they need to be converted into a `TripPlan`, displayable on OTP client side. For this purpose, a `TransferPatternToTripPlanConverter` was implemented, analogous to `GraphPathToTripPlanConverter` for routes computed by A*. Various details of each route, including times of departure and arrival, geographic details of trips and walking paths as well as other trip details are collected here and send back to client.

4.4 Missing features

In order to construct an OTP server with transfer patterns and similar capabilities as the original version, a few features are yet to be implemented. As of now, routes can only be searched *departing* at and not *arriving* before a certain time. As TP are time-independent, solely the function filling in concrete trips needs to be extended for this. Furthermore, routes are currently only searched for a single departure time τ, while OTP always offers a variety of Pareto optimal routes departing in a time window after τ.

Advanced features like Interlining or considering wheelchair accessibility during query time are not implemented yet, either. Lastly, corresponding timetables for different service days are not considered yet during precomputation. Timetable data of a full Monday is currently used, assuming that patterns do not fundamentally differ on other days. During query time, actual requested date and time are already used to retrieve direct connections, thus delivering accurate but possibly incomplete information.

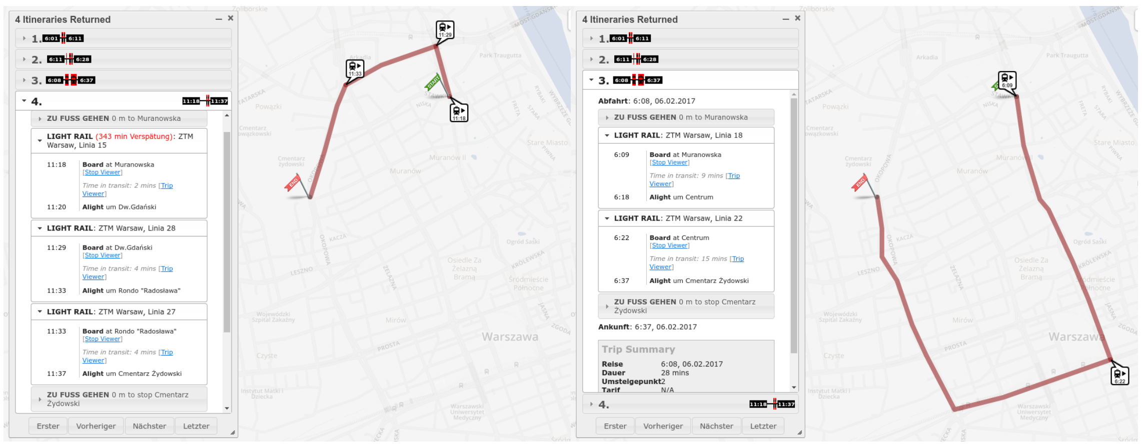
Chapter 5

Results

Dynamic transfer patterns for various cities were built on a server cluster running a basic Docker image with Ubuntu 14.04 and Java 1.8. In order to build transfer patterns on a server with a command line interface, the OTP project was exported as a jar file.

TP for the tram network of Warsaw with 236 stops were computed on 5 cores, using a 30 minutes departure filter and simple one-fold combinations of delays. With this configuration, precomputation took 2.5 hours with additional 5 minutes for merging graph chunks. The resulting TP graph consisted of over 500.000 arcs, of which 39.314 were walking arcs and 22.712 were dynamic arcs.

An example of a dynamic transfer pattern in Warsaw can be found when considering the query *Muranowska*@11.00 $\rightarrow$ *Cmentarz Żydowski*. If line *15 to P+R Al.Krakowska* is delayed, the regular route *Muranowska* $\rightharpoonup$ *Dw.Gdański* $\rightharpoonup$ *Rondo "Radosława"* $\rightharpoonup$ *Cmentarz Żydowski* (cf. Figure 5.1a) is not optimal anymore. Instead, dynamic pattern *Muranowska* $\rightharpoonup$ *Centrum* $\rightharpoonup$ *Cmentarz Żydowski* (cf. Figure 5.1b) represents a better choice now.



4 Itineraries Returned

1. 6:01 6:11
2. 6:11 6:28
3. 6:08 6:37
4. 11:18 11:37

ZU FUSS GEHEN 0 m to Muranowska

LIGHT RAIL (343 min Verspätung): ZTM Warsaw, Linia 15

11:18 Board at Muranowska [Stop Viewer]
Time in transit: 2 mins [Trip Viewer]
11:20 Alight um Dw.Gdański

LIGHT RAIL: ZTM Warsaw, Linia 28

11:29 Board at Dw.Gdański [Stop Viewer]
Time in transit: 4 mins [Trip Viewer]
11:33 Alight um Rondo "Radosława"

LIGHT RAIL: ZTM Warsaw, Linia 27

11:33 Board at Rondo "Radosława" [Stop Viewer]
Time in transit: 4 mins [Trip Viewer]
11:37 Alight um Cmentarz Żydowski

ZU FUSS GEHEN 0 m to stop Cmentarz Żydowski

4 Itineraries Returned

1. 6:01 6:11
2. 6:11 6:28
3. 6:08 6:37

Abfahrt: 6:08, 06.02.2017

ZU FUSS GEHEN 0 m to Muranowska

LIGHT RAIL: ZTM Warsaw, Linia 18

6:09 Board at Muranowska [Stop Viewer]
Time in transit: 9 mins [Trip Viewer]
6:18 Alight um Centrum

LIGHT RAIL: ZTM Warsaw, Linia 22

6:22 Board at Centrum [Stop Viewer]
Time in transit: 15 mins [Trip Viewer]
6:37 Alight um Cmentarz Żydowski

ZU FUSS GEHEN 0 m to stop Cmentarz Żydowski

Ankunft: 6:37, 06.02.2017

Trip Summary
Reise 6:08, 06.02.2017
Dauer 28 mins
Umsteigepunkt 2
Tarif N/A

4. 11:18 11:37

(a) Original route with delay of first trip **(b)** Alternative route, available iff line 15 is delayed

Figure 5.1: Example of a delayed route dominated by a dynamic route in Warsaw

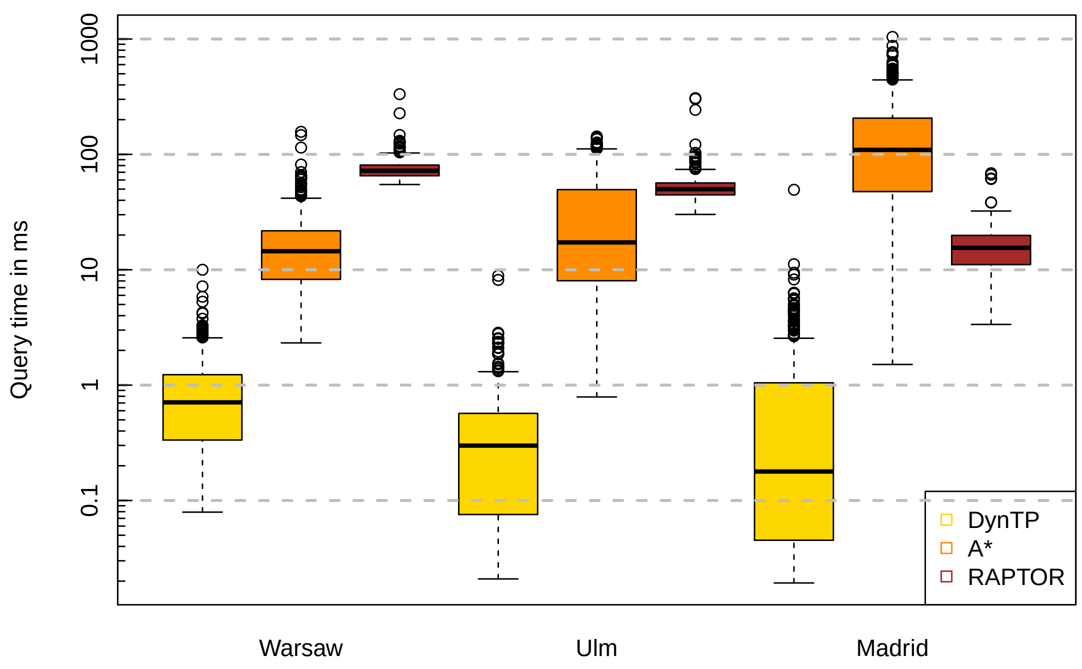


Figure 5.2: Performance of algorithms in milliseconds for 1.000 randomly chosen source-target pairs in Warsaw, Ulm and Madrid

Computation of patterns for over 28.000 stops of Berlin on 20 cores took around a day, choosing only departures every 2 hours, and utilizing maximal five random delay scenarios per source stop. Merging all 20 chunks was canceled after hours of runtime, when the maximal size of serialized objects in Java was reached. Transfer patterns of Berlin take up around 20GiB of space and thus require a different approach to storing graph data on disk in the future.

Transfer pattern graphs were furthermore computed for Ulm, Germany (505 stops, 20 routes)[1] and Madrid, Spain (4.679 stops, 213 routes)[2]. Computation time was relatively faster for Madrid here (15 mins, 15 cores) compared to Ulm (3 hours, 5 hours) because dynamic patterns and departure times for Madrid were restricted in the same fashion as for Berlin. The number of computed dynamic arcs varies widely from graph to graph. For Ulm, $0,0075\%$ of all arcs were dynamic, in Warsaw and Madrid the proportion was several orders of magnitude higher with $4,54\%$ and $0,74\%$, respectively.

[1]http://transitfeeds.com/p/swu-verkehr-gmbh/512
[2]http://transitfeeds.com/p/emt-madrid/212

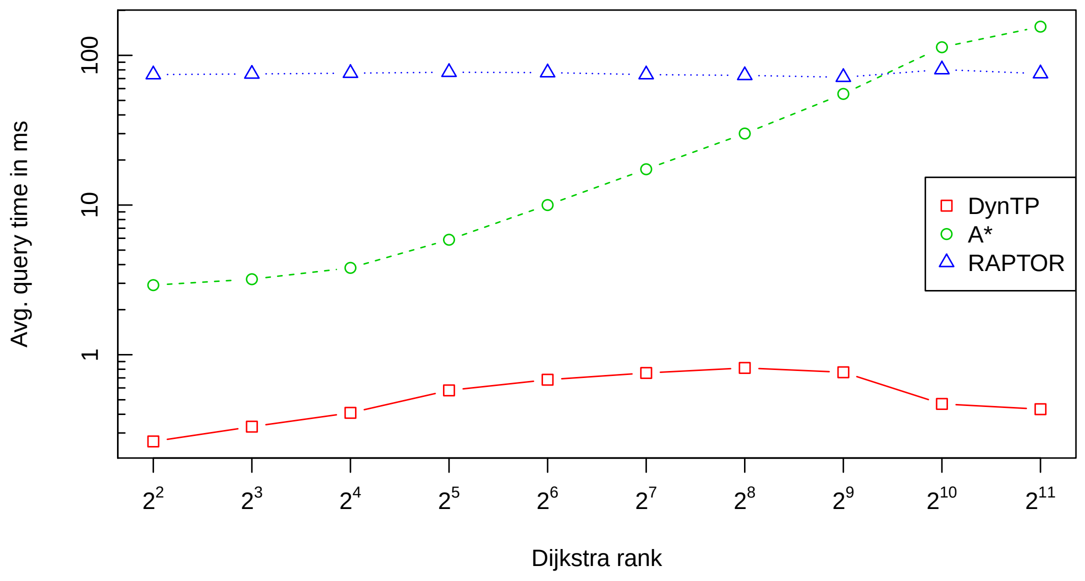


Figure 5.3: Performance of algorithms in milliseconds for various Dijkstra ranks in Warsaw

Query performance In order to compare implementations of built-in A* and RAPTOR with the implementation of dynamic transfer patterns, an algorithm computing routes with all three algorithms was introduced. Queries were built using random source and target stop pairs. The query departure time was set to 12 o'clock on an arbitrary but fixed day, since trips usually have a high density mid-day. A comparison with 1.000 random queries (see Figure 5.2) shows that dynamic transfer patterns beat OTP's A* algorithm by an order of magnitude. The dependence of graph algorithms like A* on graph size can be observed when considering its slow query times on larger graphs like Ulm and Madrid. RAPTOR's overall poor performance can be explained by the fact that its implementation in OTP merely computes one-to-all *profile searches* for analysis purposes. The average performance of RAPTOR and dynamic TP improve in Ulm and Madrid. Higher density of stops in these cities, resulting in shorter footpaths between nearby stops, could be an explanation for this phenomenon.

Computation speed of routing algorithms varies for different path lengths between source and target stop. Picking random source and target stops might over-represent long range queries and thus not reflect the reality of users of public transport. This problem tends to be more significant in larger cities, as random stops are more likely to be far apart. Sanders et al. suggest itemizing computation times by Dijkstra rank, which is defined as $r_s(n) = i$ if n is the ith node visited by Dijkstra [16]. Finally, query time was measured for all source-target pair combinations of Warsaw and split into bins of Dijkstra rank $(2^j, 2^{j+1}]$ with $j \in [2, 11]$, similarly to testing technique of Bast et al. (see Figure 5.3) [5].

Chapter 6

Discussion and future work

6.1 Scaling transfer patterns

Since precomputation of transfer patterns in itself is costly in regards to memory and time, several mechanisms that reduce the cost of precomputation have been invented. When instead of city-sized graphs the routing algorithm needs to traverse graphs of country or even continent size, transfer patterns in the original version are not suitable to precompute the required data structure. With dynamic transfer patterns, runtime increases by another factor, making the following measures even more relevant.

An early method suggested by Bast makes use of so-called hub stations, reducing the required amount of transfer patterns at query time. Instead of selecting all possible pairs of stations as input data, this approach limits transfer pattern precomputation by declaring a small subset of stations *hubs*. Hub stations are chosen by comparing the extent of their usage in routes of the network. During precomputation, transfer patterns generation is limited exclusively to paths from hub stations to all remaining stations, and from non-hubs to the closest hub [4].

Clustering, another recently developed method, is capable of reducing the precomputation time even further. Clusters are formed by choosing parts of the graph that have as few connections to outside stops as possible, a moderate size and include a stop connecting to long distance trains. In a next step, TP without hubs are calculated for each cluster. Lastly, patterns for long distance trains are computed in a different layer, as well as border patterns between clusters. During query time, a query graph is constructed from local, border and long-distance TP and traversed with Dijkstra [1].

6.2 Precomputation

Profile searches are accelerated one-to-one or one-to-all routing searches covering a larger window of departure times. Exact journeys with individual trips are not of interest here, but rather abstract information about which connections can be used throughout the requested time range. OTP features an implementation of profile routing that clusters stops by geographical distance, which could run faster than the regular routing algorithm. Other than that, there is various other ways of improving the performance of the routing algorithm, such as precomputing walking paths by adding appropriate arcs to the graph [4].

This thesis exclusively considers positive public transport delays. Negative delays (vehicles arriving too early) and their effects on emergence of dynamic patterns is a topic that could be investigated in future research.

Parallelizing the precomputation of transfer patterns sped up the process enormously but at the same time increased memory consumption significantly. This is because several Java Virtual Machines were started computing different chunks of the network, each taking roughly 20GiB of memory per instance. A solution would be to share the same graph object among multiple threads in a single JVM. As mentioned earlier, for dynamic transfer patterns this is harder to realize, since the simulation of delays alters arrival and departure times for every running instance. An abstraction of applying realtime data to the graph would thus be necessary per thread.

The exemplary GTFS data of the public transport system in Warsaw contains a stop entity per physical platform or lane instead of combining them in one stop. As the size of transfer pattern graphs scales quadratically, they are especially sensitive to such redundancy of data. A probable solution here could be to compress all entities of a stop into one by identifying them by name similarity or geographic proximity.

Another way to save disk space would be separating the TP graph from the OTP graph. Native objects of onebusaway, which is used by OTP, could be referenced by transfers patterns instead. This proposition would be particularly useful for parallelized computing of TP, as currently each graph chunk is saved with a reference to its own instance of OTP graph.

Appendix A

Algorithms and diagrams

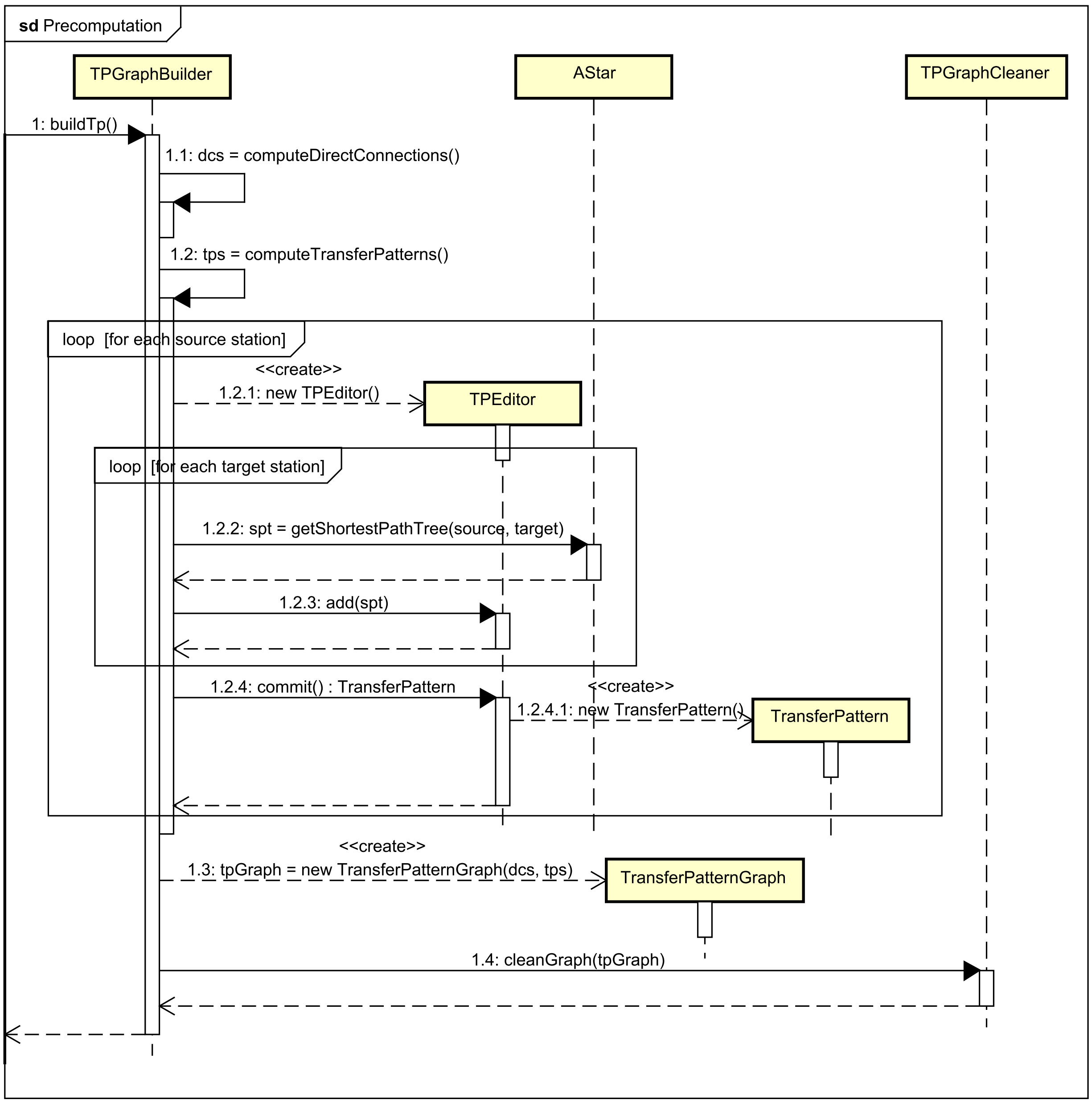

sd Precomputation
TPGraphBuilder
AStar
TPGraphCleaner
1: buildTp()
1.1: dcs = computeDirectConnections()
1.2: tps = computeTransferPatterns()
loop [for each source station]
<<create>>
1.2.1: new TPEditor()
TPEditor
loop [for each target station]
1.2.2: spt = getShortestPathTree(source, target)
1.2.3: add(spt)
1.2.4: commit() : TransferPattern
<<create>>
1.2.4.1: new TransferPattern()
TransferPattern
<<create>>
1.3: tpGraph = new TransferPatternGraph(dcs, tps)
TransferPatternGraph
1.4: cleanGraph(tpGraph)

Figure A.1: Sequence diagram of precomputation process

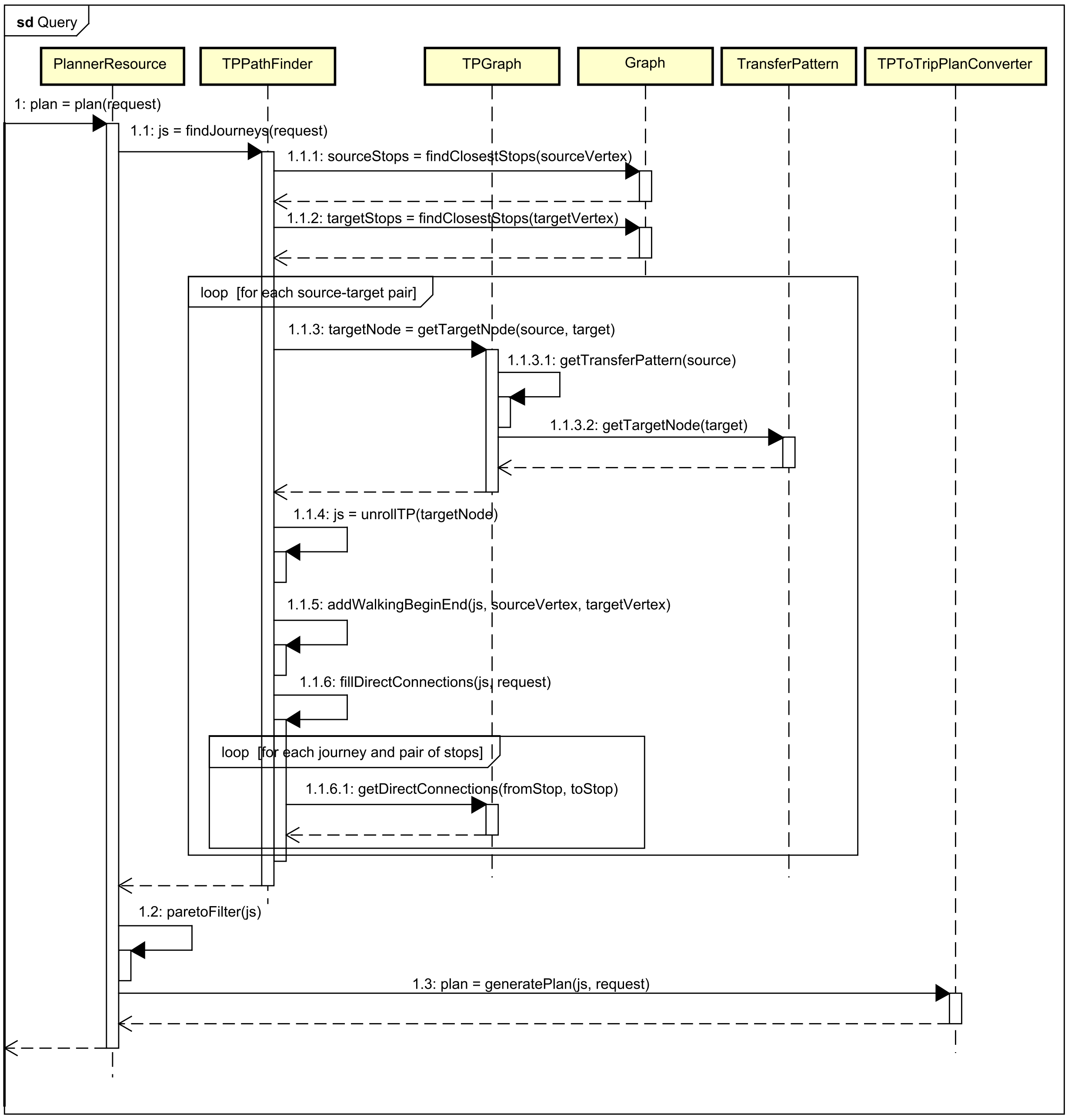

sd Query
PlannerResource
TPPathFinder
TPGraph
Graph
TransferPattern
TPToTripPlanConverter
1: plan = plan(request)
1.1: js = findJourneys(request)
1.1.1: sourceStops = findClosestStops(sourceVertex)
1.1.2: targetStops = findClosestStops(targetVertex)
loop [for each source-target pair]
1.1.3: targetNode = getTargetNode(source, target)
1.1.3.1: getTransferPattern(source)
1.1.3.2: getTargetNode(target)
1.1.4: js = unrollTP(targetNode)
1.1.5: addWalkingBeginEnd(js, sourceVertex, targetVertex)
1.1.6: fillDirectConnections(js, request)
loop [for each journey and pair of stops]
1.1.6.1: getDirectConnections(fromStop, toStop)
1.2: paretoFilter(js)
1.3: plan = generatePlan(js, request)

Figure A.2: Sequence diagram of routing query

Figure A.3: Graph cleaning algorithm

```java
void clean(Set<TransitStationStop> allStops, TransferPatternGraph tpGraph) {
  for (TransferPattern tp : tpGraph.getTransferPatterns().values()) {
    for (TransitStationStop target : allStops) {
      TPNode targetNode = tp.getTransferPattern(target);
      if (targetNode == null)
        continue;

      visit(targetNode, null, new HashSet<>());
    }
  }
}

void visit(TPNode node, TPNode lastNode, Set<TPNode> visited) {
  if (visited.contains(node)) {
    // cycle detected, remove it
    Iterator<TPTravel> itTravel = lastNode.listIterator();

    while (itTravel.hasNext()) {
      TPNode current = itTravel.next().getNode();
      if (current == node)
        itTravel.remove();
    }
  } else {
    visited.add(node);

    // clone predecessors here, since list might be changed in subsequent recursive calls
    List<TPTravel> clonedPreds = new ArrayList<>(node.getPredecessors());

    for (TPTravel pred : clonedPreds) {
      // clone here, since nodes can be reached through more than one path, just not cyclic
      HashSet<TPNode> clonedVisited = new HashSet<>(visited);
      visit(pred.getNode(), node, clonedVisited);
    }
  }
}
```

Figure A.4: Pareto comparator using departure time, arrival time and number of transfers

```java
int compare(TPJourney o1, TPJourney o2) {
  if (!o1.hasLegs())
    return !o2.hasLegs() ? 0 : -1;
  if (!o2.hasLegs())
    return 1;

  // compute differences in arrival time, departure time and number
     of transfers and transform into usable values
  int valueArr = cut(o2.getArrival() - o1.getArrival());
  int valueDep = cut(o1.getDeparture() - o2.getDeparture());
  int valueTransfer = cut(o2.legsCount() - o1.legsCount());

  // create set of compare values and sum of all values
  Set<Integer> values = Sets.newHashSet(valueArr, valueDep,
    valueTransfer);
  final int sum = valueArr + valueDep + valueTransfer;

  // determine Pareto answer
  if (sum > 0) {
    if (values.contains(-1))
      return 0;
    return 1;
  } else if (sum < 0) {
    if (values.contains(1))
      return 0;
    return -1;
  }
  return 0;
}

int cut(long difference) {
  if (difference > 0)
    return 1;
  if (difference < 0)
    return -1;
  return 0;
}
```

Bibliography

[1] Hannah Bast, Matthias Hertel, and Sabine Storandt. “Scalable Transfer Patterns”. In: *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*. 2016, pp. 15–29.

[2] Hannah Bast, Jonas Sternisko, and Sabine Storandt. “Delay-Robustness of Transfer Patterns in Public Transportation Route Planning”. In: *13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2013, September 5, 2013, Sophia Antipolis, France*. Ed. by Daniele Frigioni and Sebastian Stiller. Vol. 33. OASICS. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013, pp. 42–54. ISBN: 978-3-939897-58-3.

[3] Hannah Bast and Sabine Storandt. “Frequency-based search for public transit”. In: *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Dallas/Fort Worth, TX, USA, November 4-7, 2014*. 2014, pp. 13–22.

[4] Hannah Bast et al. “Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns”. In: *Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I*. 2010, pp. 290–301.

[5] Hannah Bast et al. “Route Planning in Transportation Networks”. In: *Computing Research Repository* abs/1504.05140 (2015).

[6] Daniel Delling, Thomas Pajor, and Renato F. Werneck. “Round-Based Public Transit Routing”. In: *Transportation Science* 49.3 (2015), pp. 591–604.

[7] Google Developers. *GTFS Static Overview | Static Transit | Google Developers*. [Online; accessed 6-February-2017]. 2016.

[8] Julian Dibbelt et al. “Intriguingly Simple and Fast Transit Routing”. In: *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*. 2013, pp. 43–54.

[9] Edsger Wybe Dijkstra. “A Note on Two Problems in Connexion with Graphs”. In: *Numerische Mathematik* 1 (1959), pp. 269–271.

[10] Robert Geisberger et al. "Exact Routing in Large Road Networks Using Contraction Hierarchies". In: *Transportation Science* 46.3 (2012), pp. 388–404.

[11] Le Minh Kieu, Ashish Bhaskar, and Edward Chung. "Empirical evaluation of public transport travel time variability". In: *Australasian Transport Research Forum 2013*. Queensland University of Technology, Brisbane, QLD, Oct. 2013.

[12] Thomas Liebig et al. "Dynamic Route Planning with Real-Time Traffic Predictions". In: *Information Systems* 64 (2017), pp. 258–265. ISSN: 0306-4379.

[13] Matthias M. "Finding All Attractive Train Connections by Multi-Criteria Pareto Search". In: *Proceedings of the 4th Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS 2004)*. Vol. 4359. Lecture Notes in Computer Science. Bergen, Norway: Springer, 2007, pp. 246–263.

[14] Ira S. Pohl. "Bi-directional search". In: *Machine Intelligence* 6 (1971), pp. 127–140.

[15] Evangelia Pyrga et al. "Efficient models for timetable information in public transportation systems". In: *ACM Journal of Experimental Algorithmics* 12 (2007), 2.4:1–2.4:39.

[16] Peter Sanders and Dominik Schultes. "Highway Hierarchies Hasten Exact Shortest Path Queries". In: *Proceedings of the 13th Annual European Conference on Algorithms*. ESA'05. Palma de Mallorca, Spain: Springer-Verlag, 2005, pp. 568–579. ISBN: 3-540-29118-0, 978-3-540-29118-3.

[17] Frank Schulz, Dorothea Wagner, and Karsten Weihe. "Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport". In: *Algorithm Engineering, 3rd International Workshop, WAE '99, London, UK, July 19-21, 1999, Proceedings*. 1999, pp. 110–123.

[18] Jonas Sternisko. "On Compact Representation and Robustness of Transfer Patterns in Public Transportation Routing". MA thesis. Albert-Ludwigs-Universität Freiburg im Breisgau, 2013.