# Multi Objective Optimization of Multimodal Two-Way Roundtrip Journeys*

Felix Gündling [1], Pablo Hoch [1], Karsten Weihe [1]
[1] Department of Computer Science, Technische Universität Darmstadt
Darmstadt 64289, Hochschulstraße 10, Germany
E-mail: {guendling, hoch, weihe}@cs.tu-darmstadt.de, Phone: +49 (0) 6151 16-20874

**Abstract**
Multi modal journeys often involve two trips: one outgoing and one return trip, as in many cases, the traveller would like to return to her starting point. If a car or bike was used in combination with public transportation (i.e. park & ride), this introduces a dependency between outward and return trip: both must include the same parking place. Optimizing both trips independently may yield suboptimal results. We consider the multi modal two-way roundtrip problem and propose several algorithms. All proposed algorithms compute journeys that are optimal regarding multiple criteria. We present a variant that supports price optimization (including driving and parking costs) as a Pareto criterion in addition to travel time and the number of transfers. Our study with realistic scenarios based on real data shows promising results.

**Keywords**
2-way round trip, multi modal, multi criteria optimization, park and ride, routing

## 1 Introduction

Many journeys do not consist of one-way trips. On the contrary, in many cases travelers return to the starting point (home for private, office for business trips) (Baumann et al., 2004). We consider a very common practical use case of multi modal routing: optimizing outward and return trip of a journey involving both private (e.g. a private car or bike) and public transportation (e.g. busses, trains, etc.). Planning a journey with commonly available online systems, that calculate optimal one-way trips, becomes quite cumbersome: finding the optimal P&R parking place (or an optimal place to park the bike) is not trivial. A parking place that was optimal for the outward trip might yield a suboptimal or infeasible return trip and vice versa. Considering multiple optimization criteria such as the number of transfers and travel time (accumulated for both trips), there might even be multiple optimal solutions. The reason for this is the time-dependent and directed nature of public transportation: an optimal route on the return trip does not necessarily include the parking place used in the outward trip. Combining independently optimized journeys may thus yield suboptimal or infeasible journeys. Consequently, optimizing both trips in a combined manner is required to compute optimal journeys for this use case. The fastest journey may not always be the most attractive one for everyone: in addition to a short travel time, some users prefer a cheap and/or convenient travel that minimizes the number of transfers. Since priorities of those

*This work was partially supported by Deutsche Bahn.

optimization targets differ from traveler to traveler, our approaches compute a complete Pareto set considering convenience (number of transfers) and travel time as criteria. One variant additionally considers the price of the journey as optimization criterion: the total price is the sum of the costs of private transportation including time dependent costs for parking as well as the public transport ticket price. For the public transport ticket price, our price model assigns a separate milage price per means of transportation (high speed trains are more expensive than local public transport). Parking prices are based on the parking duration. Our evaluation is based on real data: the public transport timetable is provided by Deutsche Bahn and covers Germany including trains (long distance as well as local), metro, busses and streetcar services. Street routing is based on Project OSRM (Luxen and Vetter, 2011) using an OpenStreetMap dataset covering the same geographic area as the public transport timetable. To the best of our knowledge there are neither scientific publications nor commercial systems offering this functionality. The presented algorithms are suitable for use in online routers and mobile routing applications.

The paper is organized as follows. Section 2 discusses previous work in the area. Section 3 outlines our contribution to the topic. Section 4 introduces basic notation and the formal problem definition. Section 5 describes the different approaches. Section 6 shows how to extend these approaches to optimize price as an additional Pareto criterion. Section 7 presents the results of our experiments and Section 8 concludes our results and gives an outlook to future research directions.

## 2 Related Work

To the best of our knowledge, there are no publications that solve the described problem in a Pareto-optimal way optimizing multiple criteria. The first solution solving the problem (Bousquet et al., 2009) considers a single optimization criterion: travel time. An improved bi-directional shortest path algorithm to solve the problem is described in (Huguet et al., 2013). Another approach based on access node routing (Delling et al., 2009) is presented in (Spinatelli, 2015). All three publications optimize travel time as single criterion and apply their algorithm to datasets covering a single city: Paris including its suburbs (Huguet et al., 2013), the rural area around Lyon (Bousquet et al., 2009), and Milano (Spinatelli, 2015). Recent advances in public transport routing and multi modal routing such as RAPTOR/MCR[1] (Delling et al., 2012, 2013), CSA (Dibbelt et al., 2013), TripBased (Witt, 2015) were not extended to compute Pareto-optimal journeys for the multi modal park and ride two-way roundtrip problem.

## 3 Contribution

In this paper, we present various algorithms to solve the two-way park and ride roundtrip problem optimizing multiple criteria in a Pareto-optimal way. We compare different solutions based on a time-dependent graph model (Disser et al., 2008) with an algorithm based on connection scanning (Dibbelt et al., 2013) and another algorithm which is based on TripBased routing (Witt, 2015). All approaches optimize travel time as well as the number of transfers. Furthermore, we propose a variant that additionally optimizes prices.

We evaluate all algorithms on a realistic nationwide network: a complete public trans-

[1] Round bAsed Public Transit Optimized Router, Multimodal Multi Criteria RAPTOR

port schedule for all of Germany including all modes of public transportation (e.g. busses, street cars, all kinds of trains) kindly provided by Deutsche Bahn. Our computational study shows that our algorithms are suitable to be deployed in online or mobile multimodal routing systems.

# 4 Preliminaries

This section describes the problem definition, static and dynamic/user inputs, and how they are preprocessed to be used as input for our core routing algorithms.

## 4.1 Problem Definition

We consider computing Pareto optimal solutions to the problem
$\alpha \xrightarrow{t_{\text{out}}} \omega@[t_1, t_2] \xrightarrow{t_{\text{ret}}} \alpha$ where we call the outward trip $t_{\text{out}}$ and the return trip $t_{\text{ret}}$. $\alpha$ and $\omega$ are locations (addresses / geographic coordinates). $\alpha$ might be the user's home address and $\omega$ the office address. The time interval $[t_1, t_2]$ is the minimal time range to stay at $\omega$ (e.g. office hours). Thus, our journeys have one of the following two structures:

$$\alpha \xrightarrow{\text{car}_1} p \xrightarrow{\text{walk}_1} s_w \xrightarrow{\text{pt}_1} s_x \xrightarrow{\text{walk}_2} \omega@t_1 \quad \dots \quad \omega@t_2 \xrightarrow{\text{walk}_3} s_y \xrightarrow{\text{pt}_2} s_z \xrightarrow{\text{walk}_4} p \xrightarrow{\text{car}_2} \alpha \quad (1)$$

$$\alpha \xrightarrow{\text{walk}_1} s_w \xrightarrow{\text{pt}_1} s_x \xrightarrow{\text{walk}_2} \omega@t_1 \quad \dots \quad \omega@t_2 \xrightarrow{\text{walk}_3} s_y \xrightarrow{\text{pt}_2} s_z \xrightarrow{\text{walk}_4} \alpha \quad (2)$$

The first one is most interesting to us. However, enabling the approach to find journeys with the second structure is necessary to avoid presenting unreasonable journeys to the user: it is not reasonable to use the car[2] if the trip between $\alpha$ and $s_w$ over $p$ ($\alpha \longleftrightarrow p \longleftrightarrow s_w$) takes longer than walking directly between $\alpha$ and $s_w$ ($\alpha \longleftrightarrow s_w$). By allowing both structures, the journey involving the unnecessary car leg (structure 1) will be superseded by the walking journey (structure 2).

We minimize the combined travel time sum of $t_{\text{out}}$ and $t_{\text{ret}}$ as one Pareto criterion and the combined number of transfers of $t_{\text{out}}$ and $t_{\text{ret}}$ as another. The travel time includes the time from the start with the car at $\alpha$ until $t_1$ for the outward trip and the time from $t_2$ until $\alpha$ is reached again for the return trip. This includes waiting times at $\omega$.

Note that the stations $s_y$ and $s_z$ as well as the stations $s_w$ and $s_w$ do not need to match but the parking place $p$ is required to be the same for outward trip $t_{\text{out}}$ and return trip $t_{\text{ret}}$. The user specifies $\alpha$, $\omega$, $t_1$, $t_2$, maximum driving distance $d_{\text{max}}$ and maximum walking distance $w_{\text{max}}$. This naturally limits the number of parking places (candidates for $p$) and stations (for journey structure 2) reachable from $\alpha$ ($\text{car}_1$ and $\text{car}_2$ / $\text{walk}_1$ and $\text{walk}_4$), the number of candidate stations for $s_w$ and $s_z$ reachable from a parking place ($\text{walk}_1$ and $\text{walk}_4$), and the number of candidate stations for $s_x$ and $s_y$ reachable from $\omega$ ($\text{walk}_2$ and $\text{walk}_3$).

## 4.2 Inputs

Basically, an algorithm to solve the problem described above requires information about the road network, the locations of suitable parking places $P$, and the public transport timetable.

[2]Bad weather or mobility impairments could be reasons to use the car regardless of longer travel time. However, weather dependent routing and routing for handicapped persons is not addressed in this paper.

The road network as well as the locations of parking places are extracted from OpenStreetMap. The public transport timetable consists of a set of stations $S$ where each is associated with a geographic coordinate and a transfer time, *trips* (a vehicle visiting a stop sequence with associated departure and arrival times) and a set of *footpaths* that connect stations which are in close proximity so that walking between them is feasible. Furthermore, the timetable data contains information such as track names, service head signs, train category and service attributes like wireless internet availability or bicycle carriage. All presented algorithms require the trips to be grouped into *routes*: all trips in a route share the same sequence of stations. Additionally, trips in a route are not allowed to overtake each other. Otherwise, the route needs to be split into two separate routes. Grouping into routes is done as a preparation step.

### 4.3 Preprocessing

Since driving and walking is only available for the first and the last leg of both trips $t_{\text{out}}$ and $t_{\text{ret}}$, we do not need to integrate both networks (timetable and road network). This allows us to use specialized models and algorithms for each network: contraction hierarchies for the road network and Time Dependent/CSA/TripBased routing for public transport (cf. Section 5). Consequently, we can split the procedure to compute optimal roundtrip journeys into two parts without losing optimality: the preprocessing step computes all possibilities for the first and last leg of $t_{\text{out}}$ and $t_{\text{ret}}$. This is the input for the actual core routing algorithm described in Section 5.

Procedure `preprocess_roundtrip()` shown in Listing 1 computes three sets $W$, $C$, and $D$: these enumerate all possibilities to reach a public transport station from $\alpha$ (sets $W$ and $C$) and $\omega$ (set $D$) respecting the journey structure and user supplied driving and walking limits $d_{\text{max}}$ and $w_{\text{max}}$. The preprocessing makes use of the following data structures and procedures:

- The procedures `car_route` and `foot_route` compute shortest paths (optimizing travel time) on the car/foot street network. They return the required time. Our `car_route` routine makes use of (Luxen and Vetter, 2011). The `foot_route` routine is a specialized implementation based on OpenStreetMap data. Routes by foot are computed by a specialized algorithm that considers stairs, crossing roads, elevators, and many more elements.

- The table `dist` contains precomputed foot path durations between parking places and nearby stations. `dist[`$p$`][`$s$`]` is the time it takes to walk from parking place $p$ to station $s$ (and vice versa).

- `get_stations` and `get_parkings` are geographic lookup functions taking a coordinate and a radius. They return all stations/parkings where the distance to the given coordinate is less than the provided radius. The functions can be efficiently implemented using a spacial data structure such as an R-tree or a quadtree.

$C$ contains all possibilities to get to a public transport station from $\alpha$ and vice versa (required to find journeys with Structure 1). Note that the entries store also the parking location. This is important because the core routing algorithm needs to match parking locations from $t_{\text{out}}$ and $t_{\text{ret}}$. $W$ contains all possibilities to walk between $\alpha$ and nearby public transport stations within $w_{\text{max}}$ distance. $W$ is required to find journeys with Structure 2. $D$

Listing 1: Preprocessing Procedure: computes edge sets $W$, $C$, and $D$ to connect $\alpha$ and $\omega$ with the public transport network.

```
dist[p ∈ P][s ∈ S]

fn car_route(from, to) do ... return driving_time done
fn foot_route(from, to) do ... return walking_time done
fn get_parkings(coordinate, radius) do ... return parking_set done
fn get_stations(coordinate, radius) do ... return station_set done

fn preprocess_roundtrip(α, ω, d_max, w_max) do
  W := ∅ // possibilities for α ⟷ s ∈ S via foot
  walking_candidates := get_stations(α, w_max)
  foreach s ∈ walking_candidates do
    walking_time = foot_route(α, s)
    W := W ⋃ {(α → s, walking_time), (s → α, walking_time)}
  done

  C := ∅ // possibilities for α → p ∈ P → s ∈ S and s ∈ S → p ∈ P → α
  Π := get_parkings(α, d_max) // parking candidates
  foreach p ∈ Π do
    c_out := car_route(α, p) // driving time outward
    c_ret := car_route(p, α) // driving time back
    station_candidates := get_stations(p, w_max)
    foreach s ∈ station_candidates do
      w := dist[p][s] // walking time between parking and station
      C := C ⋃ {(α → p → s, c_out + w), (s → p → α, c_ret + w}
    done
  done

  D := ∅ // set of possibilities ω ⟷ s ∈ S via foot
  destination_station_candidates = get_stations(ω, w_max)
  foreach s in destination_station_candidates do
    walking_time := foot_route(s, ω)
    D := D ⋃ {(s → ω, walking_time), (ω → s, walking_time)}
  done

  return (W, C, D)
done
```

contains all possibilities to walk between $\omega$ and nearby public transport stations within $w_{\max}$ distance.

The code in Listing 1 can be improved by computing the routes to all targets in one step. Since Dijkstra-like algorithms (like contraction hierarchies employed in `car_routing`) are inherently "multi target"-algorithms, we calculate the walking/driving times to all candidates in one single step instead of running one one-to-one query for each target in a loop. Thus, we change the interface of `route_car` and `route_foot` to take a single location and a set of targets as input and return the travel time to each target as result.

# 5 Approaches

This section describes the different approaches for the core routing procedure. Each algorithm takes the same input computed in the preprocessing phase (in addition to the public transport timetable): the sets $W$ and $C$ which connect $\alpha$ with the public transport network through walking/driving, and $D$ which connects $\omega$ with the public transport network through walking.

## 5.1 Time Dependent Graph

One established way to compute multi criteria shortest paths on public transport timetable networks are label correcting algorithms on graph data structures representing the timetable (i.e. time expanded and time dependent graphs). The time dependent graph is more compact (as compared to the time expanded graph) and therefore better suited to cope with large timetables containing not only trains but also streetcars and busses. So our first approach is based on the time dependent graph as described by Disser et al. (2008).

The model presented in (Disser et al., 2008) does not support backward search (latest departure problem) because it is not consistent, meaning that the path lengths $\ell(u, v)$ and $\ell(v, u)$ in forward and reversed graph differ for at least one optimization criterion. An example and the new graph layout can be found in Appendix A. So from now on, finding the journey with the latest departure (starting with a fixed arrival time) is analogous to finding the journey with the earliest arrival (starting with a fixed departure time) with reversed edges. This does also apply to the multi criteria case.

**Baseline**
In this section, we will describe an algorithm that is purely based on an unchanged base algorithm: the time dependent earliest arrival problem. We extend the graph model so that it fits the problem. Basically, the sets $W$, $C$, and $D$ can be seen as edges which extend the time dependent graph. Consequently, we need to add nodes to represent $\alpha$ and $\omega$. In the following, $\alpha$ and $\omega$ refer to those additional nodes if they are used in the graph context. Routing $t_{\text{out}}$ and $t_{\text{ret}}$ independently with all additional edges at once could yield suboptimal or unfeasible journeys due to non-matching parking places.

Since the edges from the set $D$ (connecting $\omega$ with public transport stations and vice versa) do not introduce any dependencies, they are added for every search. It is sufficient to add those that match the search direction ($\omega \to s \in S$ for $t_{\text{ret}}$ and $s \in S \to \omega$ for $t_{\text{out}}$). However, to prevent the interference between parking places, we conduct one multi-criteria search for each parking place separately for $t_{\text{out}}$ and $t_{\text{ret}}$: the graph gets extended by all edges (one for each station that is reachable from $p$) that lead over the selected parking. These are earliest arrival problems $\omega @ t_2 \to p_i$ in case of $t_{\text{ret}}$ and latest departure problems $p_i \leftarrow \omega @ t_1$ for $t_{\text{out}}$ (for every parking $i$). This generates all optimal trips $T_{\text{out}}^{p_i}$ for $t_{\text{out}}$ and $T_{\text{ret}}^{p_i}$ for $t_{\text{ret}}$ for every potential parking place $p_i$. Waiting time (arriving earlier than $t_1$ or departing later than $t_2$ at $\omega$) is considered travel time and is therefore minimized as described in Section 4.1. Note that every overall optimal roundtrip needs to be a combination of optimal trips $t_{\text{out}}$ and $t_{\text{ret}}$ for one of those potential parking places. Otherwise (if an optimal roundtrip would not be a combination of optimal individual trips), it could obviously be improved by an optimal one. Consequently, the combination of all computed trips $\bigcup_{p \in P} T_{\text{out}}^{p} \times T_{\text{ret}}^{p}$ contains all optimal roundtrips. Removing all roundtrips that are superseded by others (including

duplicate ones) yields the final set of optimal roundtrips.

Edges from the set $W$ representing all options to walk from $\alpha$ to a public transport station (for $t_{\text{out}}$) and vice versa (for $t_{\text{ret}}$) are added in a separate search (to enable the system to find journeys of Structure 2). Since there are no constraints to use the same parking place in $t_{\text{out}}$ and $t_{\text{ret}}$, they can all be used in one search.

This approach requires $2(|\Pi| + 1)$ invocations of the basic time dependent routing routine (earliest arrival / latest departure) where $\Pi$ is the set of all considered parking places: for each direction one invocation for every parking place candidate and one with all edges from $W$. This is certainly not optimal regarding computational effort (compared to the approaches presented later on). However, this approach is still useful for the practical verification of other approaches.

**Parallelization**

Since all searches are independent, they can be trivially parallelized. In theory, if the number of parallel processors is equals to two times the number of parkings, this can reduce the overall calculation time to the time it takes to respond to one routing query. However, since most systems (besides super compute clusters) cannot provide this level of parallelism, this is not a feasible approach, either.

**Combined Search for $t_{\text{out}}$ and $t_{\text{ret}}$**

The basic Dijkstra algorithm computes shortest paths not only to the target node but to all nodes in the graph. Since the basic algorithm presented by Disser et al. (2008) makes use of goal direction and domination by terminal labels[3], this property does not apply anymore: when the algorithm terminates, only the labels for one destination node will be correct (in the sense that they necessarily represent the non-extensible Pareto set).

The first step of the combined search approach is to compute the shortest paths from/to every single parking place like in the baseline approach described in Section 5.1 for one direction $t_{\text{out}}$ or $t_{\text{ret}}$. For the opposite direction, we now can make one combined search: instead of adding just the edges for only one parking, we add all parking edges but combine the edge cost with the criteria computed for the opposite direction in the first step. Since the first routing can yield more than one optimal trip for one parking, we have one additional edge for each optimal trip. Assuming we chose $t_{\text{out}}$ in the first step, we add one edge from $s \in S \rightarrow \alpha$ for each optimal $t_{\text{out}}$ journey using parking $p_i \in P$ for each station reachable from $p_i$. The edges carry the following costs: $(\text{dist}[p][s] + \text{car\_route}(p, \alpha) + \text{tt}_i, \text{ic}_i)$ where $\text{ic}_i$ is the number of transfers and $\text{tt}_i$ is the travel time for journey $i$ in $t_{\text{out}}$. If $t_{\text{ret}}$ was chosen for the first step, the approach works analogously.

This approach allows us to reduce the complexity of the baseline approach from $2(|\Pi| + 1)$ invocations of the time dependent routing routine to $|\Pi| + 1$ invocations: in one direction (outward or return) we need to route to/from every parking. In the return direction, only one query is required. The invocation with all edges from $W$ in the first step stays the same. The search in the opposite direction is conducted with all edges in $W$.

As with the baseline approach, this approach can also be parallelized. However, this approach has one constraint on the ordering: the invocation for the opposite direction requires all results from the first step.

[3] labels are partial journeys that are used in the routing algorithm

**No Terminal Domination and Worst Bounds**

Applying domination by terminal labels (in combination with lower bounds and goal direction) in the time dependent graph routing is a very effective speedup technique for queries to a single target. However, in this setting (preventing interference of labels that use different parking places), domination by terminal labels as implemented in our basic time dependent routing algorithm demands a high number of invocations as we have seen in the previous three sections. Now, we want to further reduce the number of invocations by computing all optimal journeys over all parkings in one run of the algorithm (as opposed to $|\Pi|$ invocations in the first step of Section 5.1). Simply disabling the domination by terminal labels and routing with all additional edges ($W$, $C$, and $D$) would be one option.

However, domination by terminal labels can be replaced by a different technique that still allows us to discard labels early in the search process: those that are worse or equal to the combined worst (i.e. numerically greatest assuming optimization criteria are minimized) value of each optimization criterion over all parking places ("worst bound") cannot contribute a new optimum. Consequently, this requires at least one terminal label for each parking place. Until this precondition is met, we cannot discard any label.

To implement this, we have a list of parkings that were not yet reached. This list is initialized with all parkings (identified by a unique index) reachable from $\alpha$. Every time a label reaches the target node, the used parking is removed from this list if it is the first to use this parking. If the list is empty (i.e. every parking was reached), this means that domination by worst bounds can be applied. To track the worst bounds, one variable per optimization criterion is introduced and updated every time a label is created on the target node. If every parking was reached, every newly (through edge extension) created label is compared to the stored combined worst bounds and discarded if its criteria values are equal or greater. The same check will be applied upon queue extraction since the worst bounds can change between queue insertion and extraction. Labels that were created through expansion of edges carrying different parking indices are deemed incomparable to prevent domination of options that may be part of an optimal round trip but are not optimal for this search direction. Walking options from $W$ are always added. They can be implemented as "virtual" parking directly at $\alpha$. Thus, no driving is required.

The routing for the opposite direction can be implemented as described in Section 5.1 and therefore benefit from unconditional dominance by terminal labels. This approach cannot be parallelized. Altogether this approach further reduces the number of invocations to two, albeit more complex calls: one for each direction $t_{\text{out}}$ and $t_{\text{ret}}$.

**Concurrent**

In this section, we present an algorithm that handles the search in both directions (for $t_{\text{out}}$ and $t_{\text{ret}}$) in an interleaved manner. Basically, we still have two multi criteria Dijkstra algorithms with the addition that they exchange information at runtime. Thus, every data structure (such as the priority queue, lower bounds, etc.) is redundant: one for each search direction.

Instead of the standard domination by terminal labels, we maintain a list of complete roundtrips: every time a new label has reached the target node in one direction, it is combined with each terminal label of the opposite direction that has a matching parking place. The resulting valid round trips are then added to the list of complete round trips if they are Pareto optimal. Previously added roundtrips that are worse than the newly added roundtrip are removed. These complete round trips can then be used to dominate labels in both search directions at the creation of new labels and after queue extraction: if a partial roundtrip is

already worse (in the Pareto sense) than a complete roundtrip, it can be discarded. Instead of comparing the label values (here: travel time and the number of transfers) directly with those of the terminal label, we can employ lower bounds to discard suboptimal labels as early as possible: a label with travel time $t$ in the $t_{\text{out}}$ routing takes at least $t + lb_{t_{\text{out}}}[n] + lb_{t_{\text{ret}}}[\omega]$ minutes for the complete roundtrip where $lb_{t_{\text{out}}}$ and $lb_{t_{\text{ret}}}$ are precomputed lower bounds for every node in both search directions. This is analogous for the $t_{\text{ret}}$ search: $t + lb_{t_{\text{ret}}}[n] + lb_{t_{\text{out}}}[\omega]$.

All in all, we reduced the number of invocations from $2(|\Pi| + 1)$ for the baseline approach to just one. This comes with an increased complexity of the queries. However, the combination of information (instead of separate invocations of the basic time dependent routing procedure) as described here reduces the total number of steps required to compute all optimal round trips.

### 5.2 Connection Scan

In this section, we present an algorithm that is based on the Connection Scanning Algorithm (CSA) by Dibbelt et al. (2013). As opposed to the time dependent routing algorithm, it does not require a graph to represent the timetable, neither does it depend on a priority queue. The timetable model is a simple array of all elemental connections (departure and arrival of a trip with no intermediate stops in between) of the timetable sorted by departure time. The algorithm iterates through the array and updates earliest arrival times at the stations visited by the iterated connections accordingly. The algorithm also handles footpaths between stations and transfer times between transport services.

As the basic variant of CSA just iterates "through time" (sorted connections) it is not directed towards a specific target station. Therefore, it is well suited to be adapted as a multi target algorithm without a performance penalty. This can be utilized: in the first step, we ignore the actual driving and walking times from $D$ and $W$ that connect $\alpha$ with the public transport timetable. Instead, we search from all stations in $D$ to all stations in $W$ and $C$.

The original publication does not describe a multi-criteria version of the earliest arrival problem or journey reconstruction for this type of search nor does it describe the latest departure problem or multi source and multi destination routing. Consequently, we need a specialized version of the CSA algorithm for our use case:

- *Multiple Start Stations*: In the basic version, only one station is initialized with the desired start time. In our use case, every station in $D$ is initialized with the walking time (between $\omega$ and $s \in S$) as offset that is added to $t_2$ (for $t_{\text{ret}}$) and subtracted from $t_1$ (for $t_{\text{out}}$).

- *Multiple Destination Stations*: Basically, this is what the algorithm does anyway if we omit the early termination mechanism which stops when the departure time of the currently iterated connection exceeds the earliest arrival at the destination.

- *Latest Departure Problem*: For $t_{\text{out}}$, we need to solve the problem $(s \in W \cup C) \leftarrow \omega @ t_1$. This can be done analogously to the forward search. For example, the connection array is sorted by descending arrival time and footpath walk times are subtracted instead of added.

- *Multi Criteria*: To support the optimization of the number of transfers as additional Pareto criterion, we do not only store a single earliest arrival time for each station but instead one for each number of transfers. The same applies to the array $T$ which

indicates whether a trip can be reached or not: instead of single reachable bit, one bit per number of transfers is stored. The $n^{\text{th}}$ bit indicates whether the trip can be reached with $n$ transfers.

- *Reconstruction*: Since additional journey pointers (which would need to be maintained for every number of transfers) as described in (Dibbelt et al., 2013) slow down the search (scan running time), we chose to adapt the version that works without them. As our implementation of the algorithm supports the optimization of the number of transfers as Pareto criterion, we need to reconstruct one journey for each optimal number of transfers. The recursive call with $n$ transfers at the next interchange stop continues with $n - 1$ transfers. Similarly, the trip reachable array needs to be looked up at the bit referring to the current number of transfers. Not knowing where the journey may have started imposes additional complexity: we need to iterate every possible station and check whether the travel time matches the walk ($\omega \leftrightarrow s \in D$) for this station.

Now that we have a variant that handles multiple departure stations, multiple destinations and multiple criteria in both search directions (earliest arrival / latest departure), we can utilize it to find optimal round trips: For each direction $t_{\text{out}}$ and $t_{\text{ret}}$, we execute one search. Both searches are independent and can therefore be executed in parallel. We execute one latest departure query (starting at $t_1 @ \omega$) for $t_{\text{out}}$ and one earliest arrival query (starting at $t_2 @ \omega$) for $t_{\text{ret}}$. The results of those queries are then merged to complete roundtrips by iterating every parking place and combining all journeys from $t_{\text{out}}$ and $t_{\text{ret}}$. Since not every roundtrip is necessarily optimal, we remove all that are not Pareto optimal. This yields the full set of optimal roundtrip journeys.

### 5.3 TripBased

As with the Connection Scanning Algorithm, TripBased routing as presented by Witt (2015) is inherently a multi target routing algorithm: it can be seen as a breadth first search on a graph-like data structure consisting of trip sections and transfers between those trip sections. Similarly to the RAPTOR algorithm (Delling et al., 2012), it operates in iterations/"rounds" where the $n^{\text{th}}$ iteration computes all optimal connections with $n$ transfers. Each round updates the trip sections that are reachable through one additional transfer from previously reachable trip sections.

We adapt the algorithm to be able to compute optimal journeys to multiple targets. Therefore, we need to keep one result set $J$ for each target station. Additionally, the earliest arrival time $\tau_{\text{min}}$ needs to be kept separately for each target station to check whether a trip reaching the target is optimal. A new trip segment needs to be added to the queue only if its arrival time does not exceed the maximum earliest arrival time $\tau_{\text{min}}$ over all target stations. Otherwise, it can be discarded because it cannot be optimal for any target station anymore: every slower connection with less transfers was already discovered in a previous iteration.

The additional footpaths between $\omega$ and nearby public transport stations can be handled analogously to those already contained in the basic static timetable.

In addition to the changes required to compute optimal journeys to multiple targets, the basic TripBased algorithm needs to be adjusted to compute connections for the latest departure problem, not just the earliest arrival problem. Since the preprocessed transfers (transfer reduction step) differ for the forward (earliest arrival) and reverse (latest departure)

direction, we need to have one transfer set $T$ for each search direction. This doubles the preprocessing workload. Otherwise, the latest departure computation is analogous to the earliest arrival computation described in (Witt, 2015).

As we now have an algorithm with properties similar to the adapted CSA algorithm (multi criteria, multi source, multi target, earliest departure, earliest arrival), we can use it to compute optimal roundtrip journeys as described in Section 5.2.

# 6 Price as an Additional Optimization Criterion

In this section, we present a version that optimizes not just travel time and the number of transfers but also the price. The price of the complete roundtrip is comprised of the costs for parking at $p$ (depending on the parking duration), the driving costs (of $\text{car}_1$ and $\text{car}_2$), the public transport ticket price (of $\text{pt}_1$ and $\text{pt}_2$) and an hourly wage to eliminate cheap but exceedingly long journeys that are unattractive from a practical perspective.

Since public transport pricing models are very complex, constantly changing and different for every area, we decided to use two artificial pricing models. Both are milage and vehicle class based: a high speed train (such as a German ICE or French TGV) costs \$0.22 per kilometer, a local train costs \$0.18 per kilometer and short distance transports such as busses and trams cost \$0.15 per kilometer. Additionally, we introduce an hourly wage of \$4.80 (converted to the atomic timetable time unit, minutes). The first model computes just the sum of those costs. The second, more advanced model, introduces a special ticket that allows the passenger to use arbitrary local transports (local trains, busses, trams) for a flat price (\$42.00 here). All mentioned values are freely configurable.

All algorithms need an updated route definition which takes the vehicle class into account because otherwise later departures (which will not be considered by the algorithms) may yield a cheaper connection. In the following, we describe the extensions to the approaches presented in Section 5 that enable price optimization for the two price models described above.

## 6.1 Graph Based

Extending the graph based approaches (Baseline, Parallelized Baseline, Worst Bound and Concurrent) to support price as additional Pareto criterion is mostly straightforward: the edge weight vector as well as the individual labels carry the price as additional entry. However, we need to also adjust the label comparison. Before, a label $a$ dominated label $b$ if and only if every criterion value of $a$ was less than or equal to the corresponding criterion value in $b$. The criteria were $(\text{arr}_i, -\text{dep}_i, \text{transfers}_i)$ where $\text{arr}_i$ and $\text{dep}_i$ are the arrival and the departure time of label $i$.

Instead of just adding the price to this comparison, the hourly wage requires special treatment to retain correctness of the search. Assume we compare two labels $a$ and $b$ where $a$ has a higher ticket price than $b$ but a lower total price because it arrived earlier and did accumulate less costs due to the hourly wage. Consequently, from a Pareto perspective $a$ dominates $b$ (lower price, earlier arrival with same departure time). Since $b$ arrived later, it now has to wait less for the next departure. Due to the hourly wage, the edge costs less for $b$ than it does for $a$. So after edge expansion, $a$ does not dominate $b$ anymore which implies that we lost an optimal connection. To prevent this, we need to add the hourly wage price of the travel time difference to the price of $a$ when comparing $a$ with $b$. This way, the waiting

time disparity is compensated.

The additional edges derived from the set $C$ (connecting $\alpha$ with public transport stations through a parking) now carry the according kilometer based price for the car route. Additionally, the parking itself can be modeled as a time dependent edge: coming back later to the parking increases the costs by \$2.00 per hour (staircase function).

### 6.2 Connection Scanning

**Data Structures** Extending the Connection Scanning algorithm to support price as an additional optimization criterion in the Pareto sense requires more effort than for the baseline approach because the data structures were designed with only travel time and number of transfers in mind. Before, the data structures holding the earliest arrival time for each station ($S$ - note that we use the nomenclature of the CSA publication in this section) and the trip reachable bits for each trip $T$ were both two-dimensional arrays with one entry for each number of transfers. This was sufficient for two criteria (number of transfers and travel time) because for each number of transfers only the fastest journey was relevant. Now, when additionally optimizing prices, there can be an arbitrary number of optimal journeys for each number of transfers (all optimal trade-offs between travel time and price). Thus, each entry of $S[\text{station}][\text{transfers}]$ now maintains an array with all Pareto optimal travel time / price tuples for this station instead of just the minimal travel time for this number of transfers.

The array $T$ holds a bit (for each number of transfers $n$) that indicates whether a trip is reachable with $n$ transfers. However, this is not sufficient because it is not known at which cost the trip can be reached. Note that the price to reach the trip is not the same for each section of the trip. Consequently, we need to maintain the cheapest price for each trip section for each trip for each number of transfers. This is necessary to compute the correct journey price when iterating the connections array in the main loop of the Connection Scanning Algorithm.

**Algorithm** When initializing $S$ with the offsets from $D$ (foot routes between $\omega$ and nearby public transport stations) the price (incurred by the hourly wage) needs to be initialized, too. Furthermore, the main loop of the algorithm needs to be adjusted: if a connection is reachable through the station and the trip reachable flag is set (i.e. it has a price entry for the corresponding trip section), the cheaper solution is selected. If entering the trip at this station is the cheaper solution, the price of the following trip segments in the $T$ array needs to be updated with the cheaper price including the hourly wage. Only Pareto optimal entries (travel time / price tuples) are added to $S[\text{station}][\text{transfers}]$ removing superseded ones. Footpaths also incur costs due to the hourly wage.

**Reconstruction** Naturally, the journey reconstruction step also needs to be adapted to the new data structures: when looking up $S$ and $T$ entries, not only the travel time and the number of transfers but also the price of the entry needs to match.

### 6.3 Trip-Based

**Preprocessing** The preprocessing to eliminate unnecessary transfers was aimed at transfers and travel time as optimization criteria. Thus, transfers that lead to cheap connections may be discarded. To prevent this, the transfer reduction step is omitted. Even transfers to

later trip sections of the same trip (in case the trip visits a station two or more times) and other trips of the same line can save money. U-turn transfers are still being removed.

**Data Structures** To track the price of each journey, queue entries now also carry the current journey price (in addition to the trip segment and the number of transfers). Similarly to the CSA extension, the cheapest price to reach each trip segment is maintained: the data structure $R(t)$ which previously maintained for each trip the first reachable stop now holds the cheapest price to reach each stop of the trip with the corresponding trip.

**Algorithm** The algorithm now tracks the price of each queue entry. Updating trip $t$ entails maintaining the cheapest prices in $R(t)$ as well as the cheapest prices of all trips of the same route with later departure times.

**Pruning** We extend the implementation to track the latest arrival time and most expensive price over each target station. Journeys exceeding these limits are discarded and therefore not added to the queue to be processed in the next iteration.

## 7 Computational Study

Our C++ implementation (compiler: LLVM/Clang 6 with "-O2" optimizations) of the presented algorithms was evaluated on a computer with an Intel Core i7 6850K (6x 3.6GHz) CPU and 64GB main memory. The public transport timetable was provided by Deutsche Bahn and covers all services (busses, trams, trains, etc.) operated in Germany. For foot and car routing the complete OpenStreetMap dataset of Germany was loaded.

The timetable spans the 27th and 28th of November 2018. It contains approximately 30M departure and arrival events (60M events total) that take place in 1.7M trips on 224,832 routes.

Queries are generated by choosing a random $t_1$ and a random $t_2$ 30min to 4h after $t_1$. To generate coordinates that yield a high chance of non-empty result sets, we randomly select a public transport station that has at least one arrival event in the time interval $[t_1 - 60\text{min}, t_1]$ and at least one departure event in the time interval $[t_2, t_2 + 60\text{min}]$. Then, a random coordinate in a radius of $w_{\max}$ around this station is selected as $\omega$. $\alpha$ is a random coordinate located in a 200km radius around destination. Both coordinates need to be within Germany which is checked with the help of a polygon that resembles Germanys borders.

### 7.1 Preprocessing

Extracting all parking places and calculating optimal foot paths between public transport stations and nearby parking places takes 41 minutes and 44 seconds. However, this needs to be done only once for every dataset. At runtime, a fast lookup table with the precomputed foot path times is used. Our OpenStreetMap dataset contains 319,361 parking places. On average, 5.18 stations are reachable from a parking place (median 4, 99% quantile 23).

The execution of the preprocessing step described in Section 4.3 takes place at query runtime. Street routing between $\alpha$ and all parking places in the selected radius takes 383ms. The lookup times for stations/parkings in a specified radius around a coordinate are negligible (below 1ms). Lookup of precomputed foot routes between parking places and nearby

public transport stations takes 3.68ms. Since $\omega$ is a user input, foot paths between $\omega$ and nearby public transport stations (set $D$) cannot be precomputed. Computing $W$ takes 27.4ms at runtime. The sets $W$, $D$ and $C$ can be computed in parallel. So in total, 387ms of the runtime are due to preprocessing. The next sections report runtimes including preprocessing times. Therefore, to obtain the total core routing runtime, approximately 0.4s need to be subtracted from the runtimes reported below.

### 7.2 Baseline Algorithms

Table 1: Runtimes of Baseline Algorithms without Price Optimization in Milliseconds

| | avg | Q(99) | Q(90) | Q(80) | Q(50) |
|---|---|---|---|---|---|
| Baseline | 659 700 | 2 732 816 | 1 676 236 | 924 110 | 398 985 |
| Combined | 399 353 | 1 455 144 | 868 975 | 630 930 | 258 519 |
| Parallel | 276 666 | 761 288 | 561 843 | 444 319 | 215 487 |
| Comb. Par. | 193 793 | 624 920 | 402 261 | 301 287 | 159 545 |

As depicted in Table 7.2, parallel execution of the baseline approach yields a reasonable 2.4x speedup on average. The "trick" of an integrated optimization for one of the two directions (including parallel execution for the non-integrated search direction) yields another 2x speedup on average. Nonetheless, the baseline approach and its variations described in Section 5.1 (parallel implementation) and Section 5.1 (combined search) are not really of any practical use because they require many invocations of the time dependent routing routine. Users of online services are not eager to wait more than three minutes for their routing result. However, due to their simplicity those approaches are useful for validation of the other implementations.

### 7.3 Advanced Algorithms

Table 2: Runtimes of Advanced Algorithms in Milliseconds

| | avg | Q(99) | Q(90) | Q(80) | Q(50) |
|---|---|---|---|---|---|
| No Terminal Dominance | 4800 | 11 430 | 7969 | 6615 | 4403 |
| Worst Bound | 4762 | 11 268 | 8042 | 6564 | 4363 |
| Concurrent | 3573 | 10 305 | 6439 | 5000 | 3156 |
| CSA | 1697 | 3384 | 2322 | 1999 | 1577 |
| CSA SIMD | 908 | 2453 | 1353 | 1113 | 806 |
| TripBased | 816 | 2644 | 1302 | 975 | 689 |

In this section, we present the results of the advanced algorithms: different Dijkstra-based algorithms (with worst bounds, without terminal dominance and the interleaved / concurrent approach) on the time dependent graph model (introduced in Section 5.1), Connection Scanning (Section 5.2), and TripBased routing (Section 5.3). Additionally, we have implemented a CSA version that makes use of SIMD instructions.

As we can see in Table 7.3, the concurrent (interleaved) search in both directions (outward and return trip) brings the runtimes on the time dependent graph down from more than three minutes to 3.5 seconds. However, one percent of the queries take more than 10 seconds to answer. All non-graph-based approaches (CSA and TripBased) yield better runtime performance: the CSA SIMD variant as well as the trip based routing have average runtimes under one second. The data-parallel SIMD implementation of the CSA algorithm yields nearly a 2x speedup compared to the basic CSA version. Note that the CSA SIMD version is even faster than the TripBased approach when it comes to the 99% quantile (2.45 seconds vs 2.64 seconds) indicating that it has a more predictable performance profile.

**Parking Radius** In this section, we analyze the relation of the runtime of the approaches presented in this paper with the $d_{\max}$ parameter which essentially determines the number of parkings to consider. We analyze this relation for two and three optimization criteria.

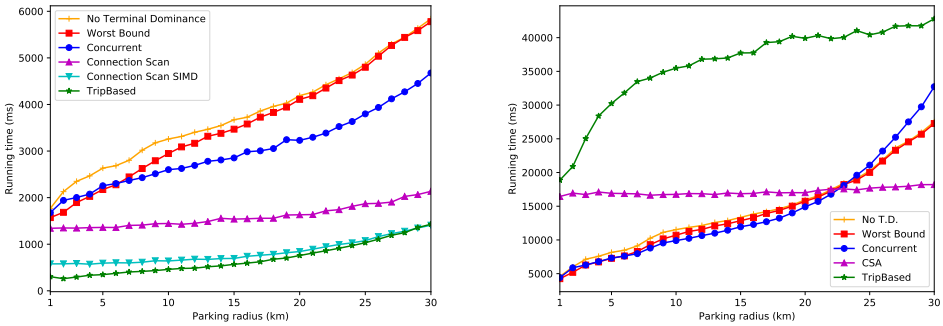



Figure 1: Runtime Subject to Parking Radius Distance $d_{\max}$: the left figure shows the basic optimization with travel time and number of transfers. The right figure shows the runtime for optimization with all three criteria: travel time, number of transfers and price.

As we can see in Figure 1, the runtime scales mostly linearly with the parking radius for all approaches regardless of the optimization criteria. However, the increase in runtime is different for the presented approaches without price optimization: while the Connection Scan SIMD and TripBased runtimes rise minimally with an increased parking radius and stay below one second, Concurrent and Worst Bound runtimes have a steep increase with a growing parking radius.

Note the different ordinate scale of the right graph of Figure 1: price optimization imposes a heavy toll on query runtime. When optimizing prices, the runtimes of the graph based approaches (Worst Bound and Concurrent) for short distances are better than those of CSA and TripBased. However, this changes for $d_{\max}$ values greater than 23km where CSA delivers the fastest (almost constant) runtimes. A mixed approach could pick a graph based algorithm for smaller radii and switch to CSA for larger radii. As the TripBased algorithm is tailored to two optimization criteria (travel time and number of transfers), the runtimes with three optimization criteria lack behind the other approaches.

**Distance Analysis** In Table 7.3 we see that the runtimes of the CSA and TripBased approaches are insensitive to changing distances between $\alpha$ and $\omega$. All runtimes of graph

Table 3: Runtimes for Different $\alpha/\omega$ Distances in Milliseconds

| | 50km | 200km | 900km |
|---|---|---|---|
| No Terminal Dominance | 2236 | 4800 | 5308 |
| Worst Bound | 2234 | 4762 | 5263 |
| Concurrent | 1544 | 3573 | 4876 |
| CSA | 1778 | 1697 | 1656 |
| CSA SIMD | 952 | 908 | 853 |
| TripBased | 878 | 816 | 730 |

based approaches grow with larger distances. Note that for short distances, the average runtimes of the Concurrent approach are lower than those of the basic CSA approach. This is not the case anymore for higher distances.

### 7.4 Price Optimization

Table 4: base scenario, no price vs. simple price vs. regional price

| | no price | simple price | regional price |
|---|---|---|---|
| No Terminal Dominance | 4800 | 19 505 | 17 492 |
| Worst Bound | 4762 | 19 249 | 17 316 |
| Concurrent | 3573 | 17 718 | 15 141 |
| CSA | 1697 | 18 314 | 23 245 |
| TripBased | 816 | 40 670 | 39 542 |

In this section, we analyze the impact price optimization has on the runtimes of the different approaches. We evaluated both public transport price models introduced in Section 6: one is based only on distance and vehicle class (called "simple" in this section), the other model adds a special regional ticket with a flat price (called "regional price"). The changed route definition (described in Section 6) leads to 0.61% more routes. As we can see in Table 4, this additional search criterion increases the runtimes of all algorithms between 4x and 50x compared to the two criteria implementations (regardless of the concrete pricing model).

Since the price optimizing implementation of TripBased disabled most of the speedup it gained through the preprocessing (transfers reduction), it switched from being the fastest implementation to being the slowest implementation. Note that while Worst Bound and Concurrent could gain more than 10% speedup through the regional price model, CSA was slowed down by it (by more than 25%).

## 8 Conclusion

We presented several novel approaches to compute Pareto optimal solutions to the 2-way park and ride roundtrip problem. In addition to two criteria optimization (travel time and number of transfers), we introduce variants of the approaches which additionally optimize

the journey price. Since many journeys follow this pattern (e.g. for commuters), the developed algorithms are useful in practice. The approaches are based on state-of-the art algorithms for public transport routing such as TD (Disser et al., 2008), CSA (Dibbelt et al., 2013) and TripBased routing (Witt, 2015). Our evaluation on a dataset covering all of Germany shows that the approaches offer query runtimes below 2 seconds which makes them suitable for use in online or mobile app information systems.

## References


Baumann, D., Torday, A., and Dumont, A.-G. (2004). The importance of computing intermodal roundtrips in multimodal guidance systems. In *Proceedings of the 4th STRC Swiss Transport Research Conference*, number LAVOC-CONF-2008-029.

Bousquet, A., Constans, S., and El Faouzi, N.-E. (2009). On the adaptation of a label-setting shortest path algorithm for one-way and two-way routing in multimodal urban transport networks. In *International Network Optimization Conference*.

Delling, D., Dibbelt, J., Pajor, T., Wagner, D., and Werneck, R. F. (2013). Computing multimodal journeys in practice. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 260–271. Springer.

Delling, D., Pajor, T., and Wagner, D. (2009). Accelerating multi-modal route planning by access-nodes. In *European Symposium on Algorithms*, pages 587–598. Springer.

Delling, D., Pajor, T., and Werneck, R. F. (2012). Round-based public transit routing. In *2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 130–140. SIAM.

Dibbelt, J., Pajor, T., Strasser, B., and Wagner, D. (2013). Intriguingly simple and fast transit routing. In *International Symposium on Experimental Algorithms*, pages 43–54. Springer.

Disser, Y., Müller-Hannemann, M., and Schnee, M. (2008). Multi-criteria shortest paths in time-dependent train networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 347–361. Springer.

Huguet, M.-J., Kirchler, D., Parent, P., and Calvo, R. W. (2013). Efficient algorithms for the 2-way multi modal shortest path problem. *Electronic Notes in Discrete Mathematics*, 41:431–437.

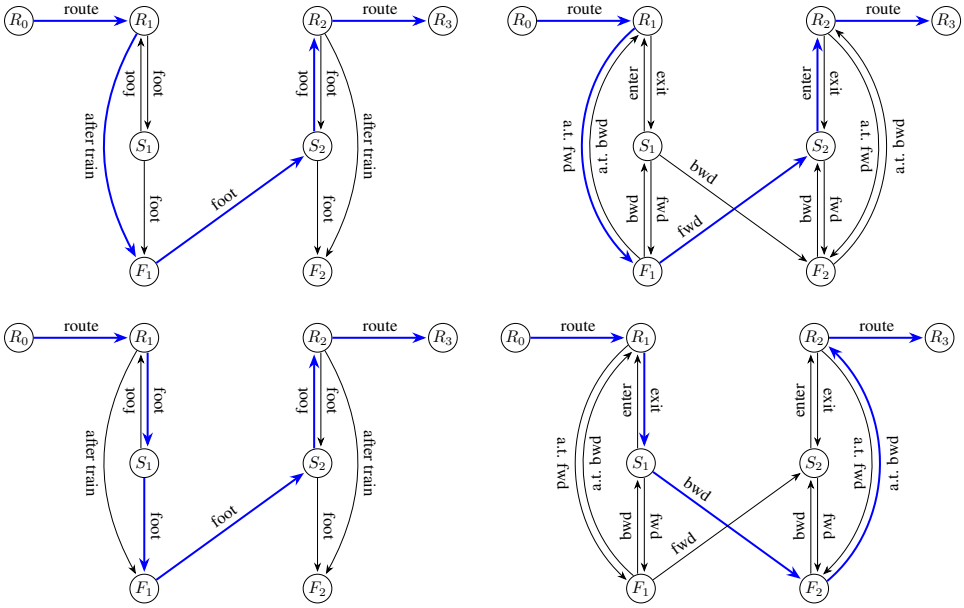
Luxen, D. and Vetter, C. (2011). Real-time routing with openstreetmap data. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11, pages 513–516, New York, NY, USA. ACM.

Spinatelli, S. (2015). Minimal effective time two-way park and ride problem. Master's thesis.

Witt, S. (2015). Trip-based public transit routing. In *Algorithms-ESA 2015*, pages 1025–1036. Springer.

# A Changes to the Time Dependent Graph Model by Disser et al. (2008) to Support Latest Departure Queries

Figure 2: Changes to the Time Dependent Graph Model to Support Backward Search: The new model (right side) fixes the inconsistency (different costs for forward and backward search) of the old model (left side).




As we can see in Figure 2 (edge costs are listed in Table 5), the basic time dependent model presented in (Disser et al., 2008) is not consistent (i.e. equal graph costs for the same journey in forward and backward search) for routes containing walks between nearby stations: in the backward search the path includes the transfer costs of $S_2$ while in the forward search no transfer costs are included (which is the desired behavior). In the fixed model, a walk between two stations has the same costs in forward and backward search direction.

Table 5: Edge Type Costs for Forward and Backward Search in the Time Dependent Graph as (Travel Time, Transfer Count) tuples: costs marked with a star "*" are not feasible at edge expansion if the corresponding label did not use a route edge before. The symbol $\oslash$ indicates that the edge is not feasible in this search direction. $\text{ic}_s$ is the transfer time for interchanges at station $s \in S$.

| Edge Type | Forward Search | Backward Search |
|---|---|---|
| enter | $(0, 0)$ | ($\text{ic}_s$, 1)* |
| exit | ($\text{ic}_s$, 1)* | $(0, 0)$ |
| after train forward | $(0, 1)$* | $\oslash$ |
| after train backward | $\oslash$ | $(0, 1)$* |
| fwd | $(x, y)$ | $\oslash$ |
| bwd | $\oslash$ | $(x, y)$ |