Arc-Flags Meet Trip-Based Public Transit Routing

Ernestine Großmann
Universität Heidelberg, Germany

Jonas Sauer
Karlsruhe Institute of Technology, Germany

Christian Schulz
Universität Heidelberg, Germany

Patrick Steil
Universität Heidelberg, Germany

Abstract

We present Arc-Flag TB, a journey planning algorithm for public transit networks which combines Trip-Based Public Transit Routing (TB) with the Arc-Flags speedup technique. Compared to previous attempts to apply Arc-Flags to public transit networks, which saw limited success, our approach uses stronger pruning rules to reduce the search space. Our experiments show that Arc-Flag TB achieves a speedup of up to two orders of magnitude over TB, offering query times of less than a millisecond even on large countrywide networks. Compared to the state-of-the-art speedup technique Trip-Based Public Transit Routing Using Condensed Search Trees (TB-CST), our algorithm achieves similar query times but requires significantly less additional memory. Other state-of-the-art algorithms which achieve even faster query times, e.g., Public Transit Labeling, require enormous memory usage. In contrast, Arc-Flag TB offers a tradeoff between query performance and memory usage due to the fact that the number of regions in the network partition required by our algorithm is a configurable parameter. We also identify a previously undiscovered issue in the transfer precomputation of TB, which causes both TB-CST and Arc-Flag TB to answer some queries incorrectly. We provide discussion on how to resolve this issue in the future. Currently, Arc-Flag TB answers 1–6% of queries incorrectly, compared to over 20% for TB-CST on some networks.

2012 ACM Subject Classification Theory of computation → Shortest paths; Mathematics of computing → Graph algorithms; Applied computing → Transportation

Keywords and phrases Public transit routing, graph algorithms, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.16

Supplementary Material *Software (Code)*: `https://github.com/TransitRouting/Arc-FlagTB` archived at `swh:1:dir:6788b9abdd6ad1056ecdec1365b88effd6057558`

Funding We acknowledge support by DFG grants SCHU 2567/3-1 and WA 654/23-2.

Acknowledgements We want to thank Dr. Patrick Brosi for providing us with the Germany dataset and Sascha Witt for providing us with the source code for TB-CST.

1 Introduction

Interactive journey planning applications which provide routing information in real time have become a part of our everyday lives. While Dijkstra's Algorithm [17] solves the shortest path problem in quasi-linear time, it still takes several seconds on continental-sized networks, which is too slow for interactive use. Practical applications therefore rely on *speedup techniques*, which compute auxiliary data in a preprocessing phase and then use this data to speed up the query phase. Recent decades have seen the development of many successful speedup techniques for route planning on road networks [4]. These achieve query times of less than a millisecond with only moderate preprocessing time and space consumption.

For public transit networks, the state of the art is not as satisfactory. In order to achieve query times below a millisecond on large country-sized networks, existing techniques must precompute data in the tens to hundreds of gigabytes. This discrepancy has been explained by the fact that road networks exhibit beneficial structural properties that are not as pronounced in public transit networks [2]. An additional challenge is that passengers in public transportation systems typically consider more criteria than just the travel time when evaluating journeys. Most recent algorithms in the literature Pareto-optimize at least two criteria: arrival time and the number of used trips. For these reasons, a speedup technique which achieves very low query times with only a moderate amount of precomputed data remains elusive.

State of the Art. For this work, we consider algorithms for journey planning in public transit networks which Pareto-optimize arrival time and number of trips. For a more general overview of journey planning algorithms, we refer to [4]. The classical approach is to model the public transit timetable as a graph and then apply a multicriteria variant of Dijkstra's Algorithm [19, 22, 18]. The time-dependent and time-expanded approaches are the two most prominent ways of modeling the timetable. In the *time-dependent* model [11, 23], stops in the network are represented by nodes in the graph and connections between them as edges with a time-dependent, piecewise linear travel time function. This yields a compact graph but requires a time-dependent version of Dijkstra's Algorithm. By contrast, the *time-expanded* model [22, 23] introduces a node for each event in the timetable (e.g., a vehicle arriving or departing from a stop). Edges connect consecutive events of the same trip and events between which a transfer is possible. The resulting graph is significantly larger but has scalar edge weights, allowing Dijkstra's Algorithm to be applied without modification.

Using a graph-based model has the advantage that speedup techniques for Dijkstra's Algorithm can be applied. However, the achieved speedups are much smaller than on road networks [2, 7]. A notable technique which has been applied to bicriteria optimization in public transit networks is Arc-Flags [21]. Its basic idea is to partition the graph into regions and to compute a *flag* for each combination of edge and region, which indicates whether the edge is required to reach the region. Dijkstra's Algorithm can then be sped up by ignoring unflagged edges. Arc-Flags has been applied to both time-dependent [10] and time-expanded [15] graphs, although only arrival time was optimized in the latter case. This yielded speedups of 3 and 4, respectively, whereas Arc-Flags on road networks achieves speedups of over 500 [21].

More recent algorithms do not model the timetable as a graph but employ more cache-efficient data structures to achieve faster query times. Notable examples are RAPTOR [16] and Trip-Based Public Transit Routing (TB) [25]. The latter employs a lightweight preprocessing phase which precomputes relevant transfers between individual trips. This yields query times in the tens of milliseconds, even on large networks, significantly improving upon graph-based techniques. HypRAPTOR [14] achieves a speedup of 2 over RAPTOR by using hypergraph partitioning to group the vehicle routes into cells and precomputing a set of *fill-in routes* which are required to cross cell boundaries. Applying the same approach to TB has only yielded a speedup of 20–40% [1].

Algorithms which reduce query times to the sub-millisecond range do so by precomputing auxiliary data whose size is quadratic in the size of the network. Public Transit Labeling (PTL) [13] adapts the ideas of Hub Labeling [12] to time-expanded graphs. While this yields query times of a few microseconds, it requires tens of gigabytes of space on metropolitan networks. Moreover, this does not include the additional overhead required for *journey unpacking*, i.e., retrieving descriptions of the optimal journeys, which would increase the size of the auxiliary data into the hundreds of gigabytes. Transfer Patterns (TP) [3] employs a preprocessing phase which essentially answers every possible query in advance. Since storing a full description of every optimal journey would require too much space, TP condenses this information into a generalized search graph for each possible source stop, which is then explored during the query phase. On the network of Germany, TP answers queries in less than a millisecond but requires hundreds of hours of preprocessing time and over 100 GB of space. Scalable Transfer Patterns [5] reduces the preprocessing effort with a clustering-based approach, but the resulting query times are only barely competitive with TB. Trip-Based Routing Using Condensed Search Trees (TB-CST) [26] re-engineers the ideas of TP with a faster, TB-based preprocessing algorithm and by splitting the computed search graphs in order to save space.

Contribution. We revisit the concept of Arc-Flags for public transit journey planning. In contrast to previous approaches, we use modern TB-based algorithms in preprocessing and query phases. The high cache efficiency and stronger pruning rules of these algorithms drastically reduce the search space and running times. The resulting algorithm, Arc-Flag TB, matches or exceeds the performance of TB-CST with a similar precomputation time and significantly lower space consumption. Compared to TB, it achieves a speedup of one order of magnitude on metropolitan networks and two orders of magnitude on country networks. Since the number of regions in the underlying network partition is a configurable parameter, Arc-Flag TB additionally offers a tradeoff between query performance and the size of the precomputed data.

We identify an issue in the transfer precomputation of TB, which both TB-CST and Arc-Flag TB rely on. As a result, both algorithms answer some queries incorrectly. We discuss how this issue can be resolved in the future. In its current configuration, Arc-Flag TB answers 1–6% of queries incorrectly, depending on the network, compared to over 20% for TB-CST on some networks. Altogether, we show that Arc-Flags for public transit networks has more potential than previously thought.

2 Preliminaries

2.1 Basic Concepts

Public Transit Network. A *public transit network* is a 4-tuple $(\mathcal{S}, \mathcal{L}, \mathcal{T}, \mathcal{F})$ consisting of a set of stops $\mathcal{S}$, a set of lines $\mathcal{L}$, a set of trips $\mathcal{T}$, and a set of footpaths $\mathcal{F} \subseteq \mathcal{S} \times \mathcal{S}$. A *stop* $p \in \mathcal{S}$ is a location where a vehicle stops and passengers can enter or exit the vehicle. A *trip* is a sequence $T = \langle T[0], T[1], \dots \rangle$ of *stop events*, where each stop event $T[i]$ has an associated arrival time $\tau_{\mathrm{arr}}(T[i])$, departure time $\tau_{\mathrm{dep}}(T[i])$ and stop $p(T[i]) \in \mathcal{S}$. We denote the number of stop events in T as $|T|$. Trips with the same stop sequence that do not overtake each other are grouped into *lines*. A trip $T_a \in \mathcal{T}$ overtakes another trip $T_b \in \mathcal{T}$ if there are stops $p, q \in \mathcal{S}$ such that T_b arrives (or departs) later at p than T_a, but T_a arrives (or departs) earlier than T_b at q. The set of all trips belonging to a line L is denoted as $\mathcal{T}(L)$. Since trips $T_a, T_b \in \mathcal{T}(L)$ cannot overtake each other, we can define a total ordering

$$\begin{aligned} T_a \preceq T_b &\iff \forall i \in [0, |T_a|) : \tau_{\mathrm{arr}}(T_a[i]) \le \tau_{\mathrm{arr}}(T_b[i]) \\ T_a \prec T_b &\iff T_a \preceq T_b \wedge \exists i \in [0, |T_a|) : \tau_{\mathrm{arr}}(T_a[i]) < \tau_{\mathrm{arr}}(T_b[i]). \end{aligned}$$

A *footpath* $(p, q) \in \mathcal{F}$ allows passengers to transfer between stops p and q with the *transfer time* $\Delta\tau_{\mathrm{fp}}(p, q)$. If no footpath between p and q exists, we define $\Delta\tau_{\mathrm{fp}}(p, q) = \infty$. If $p = q$, then $\Delta\tau_{\mathrm{fp}}(p, q) = 0$. We require that the set of footpaths is transitively closed and fulfills the triangle inequality, i.e., if there are stops $p, q, r \in \mathcal{S}$ with $(p, q) \in \mathcal{F}$ and $(q, r) \in \mathcal{F}$, then there must be a footpath $(p, r) \in \mathcal{F}$ with $\Delta\tau_{\mathrm{fp}}(p, r) \le \Delta\tau_{\mathrm{fp}}(p, q) + \Delta\tau_{\mathrm{fp}}(q, r)$.

A *trip segment* $T[i, j]$ ($0 \leq i < j < |T|$) is the subsequence of trip T between the two stop events $T[i]$ and $T[j]$. A *transfer* $T_a[i] \to T_b[j]$ represents a passenger transferring from T_a to T_b at the corresponding stop events. Note that this requires $\tau_{\mathrm{arr}}(T_a[i]) + \Delta\tau_{\mathrm{fp}}(p(T_a[i]), p(T_b[j])) \leq \tau_{\mathrm{dep}}(T_b[j])$. A *journey* J from a source stop p_s to a target stop p_t is an sequence of trip segments such that every pair of consecutive trip segments is connected by a transfer. In addition, a journey contains an *initial* and *final footpath*, where the initial footpath connects p_s to the first stop event, and the final footpath connects the last stop event to p_t.

Problem Statement. A journey J from p_s to p_t is evaluated according to two criteria: its arrival time at p_t, and the number of trips used by J. We say that J *weakly dominates* another journey J' if J is not worse than J' in either of the two criteria. Moreover, J *strongly dominates* J' if J weakly dominates J' and J is strictly better in at least one criterion. Given source and target stops $p_s, p_t \in \mathcal{S}$ and an earliest departure time τ_{dep} at p_s, a journey J from p_s to p_t is *feasible* if its departure time at p_s is not earlier than τ_{dep}. A *Pareto set* $\mathcal{P}$ is a set of journeys such that $\mathcal{P}$ has minimal size and every feasible journey is weakly dominated by a journey in $\mathcal{P}$. Given source and target stops $p_s, p_t \in \mathcal{S}$ and a departure time τ_{dep}, the *fixed departure time problem* asks for a Pareto set with respect to the two criteria arrival time and number of trips. For the *profile problem*, we are given an interval $[\tau_1, \tau_2]$ of possible departure times in addition to p_s and p_t. Here, the objective is to find the union of the Pareto sets for each distinct departure time $\tau \in [\tau_1, \tau_2]$. In the *full-range profile problem*, the departure time interval spans the entire service duration of the network.

Graph. A directed, weighted graph $G = (V, E, c)$ is a triple consisting of a set of nodes V, a set of edges $E \subseteq V \times V$, and an edge weight function $c : E \to \mathbb{R}$. We denote by $n = |V|$ the number of nodes and $m = |E|$ the number of edges. A *path* $P = \langle v_1, v_2, \ldots, v_k \rangle$ is a sequence of nodes between v_1 and v_k such that an edge connects each pair of consecutive nodes. The weight of a path is the sum of the weights of all edges in the path. A path $P = \langle v_s, \ldots, v_t \rangle$ between a source node v_s and a target node v_t is called the *shortest path* if there is no other path between v_s and v_t with a smaller weight.

Given a value $k \in \mathbb{N}$ and a graph $G = (V, E, c)$, a (k-way) *partition* of G is a function $r : V \to \{1, \ldots, k\}$ which partitions the node set V into k *cells*. The set of nodes in cell i is denoted as $V_i := r^{-1}(i)$. An edge (u, v) is called a *cut edge* if its endpoints u and v belong to different cells. A node is called a *boundary node* if it is incident to a cut edge. The partition is called *balanced* for an imbalance parameter $\varepsilon > 0$ if the size of each cell V_i is bounded by $|V_i| \leq (1 + \varepsilon) \left\lceil \frac{|V|}{k} \right\rceil$. The *graph partitioning problem* asks for a balanced partition that minimizes the weighted sum of all cut edges.

2.2 Related Work

Trip-Based Public Transit Routing. The Trip-Based Public Transit Routing (TB) algorithm [25] solves the fixed departure time problem on a public transit network. It employs a precomputation phase, which first generates all possible transfers between stop events. Then, using a set of pruning rules, transfers that are not required to answer queries are discarded. We denote the remaining set of transfers as $\mathfrak{T}$. Note that $\mathfrak{T}$ may still contain transfers which do not occur in any Pareto-optimal journey.

The TB query algorithm is a modified breadth-first search on the set of trips and the precomputed transfers. The algorithm tracks which parts of the network have already been explored by maintaining a *reached index* $R(T)$ for each trip T. This is the index of the first reached stop event of T, or $|T|$ if none have been reached yet. The TB query operates in *rounds*, where round i finds Pareto-optimal journeys which use i trips. Each round maintains a FIFO (first-in-first-out) queue of newly reached trip segments; these are then scanned during the round. A trip segment $T_a[i, j]$ is scanned by iterating over the stop events $T_a[k]$ with $i \leq k \leq j$ and relaxing all outgoing transfers $(T_a[k], T_b[\ell]) \in \mathfrak{T}$. If $\ell < R(T_b)$, then the trip segment $T_b[\ell, R(T_b) - 1]$ is added to the queue for the next round. Additionally, for every succeeding trip T'_b of the same line with $T_b \preceq T'_b$, the reached index $R(T'_b)$ is set to $\min(R(T'_b), \ell)$. This ensures that the search only enters the earliest reachable trip of each line, a principle we call *line pruning*.

Profile-TB is an extension of TB for solving the profile problem. It exploits the observation that journeys with a later departure time weakly dominate journeys with an earlier arrival time if they are equivalent or better in the other criteria. Therefore, it collects all possible departure times at p_s within the departure time interval $[\tau_1, \tau_2]$ and processes them in descending order. For each departure time, a run of the TB query algorithm is performed. All data structures, including reached indices, are not reset between runs. This allows results from earlier runs to prune suboptimal results in the current run, a principle called *self-pruning*. In order to obtain correct results, the definition of reached indices must be modified slightly. For each trip T and each possible number of trips i, the algorithm now maintains a reached index $R_i(T)$, which is the index of the first stop event in T which was reached with i or fewer trips. Whenever $R_i(T)$ is updated to $\min(R_i(T), k)$ for some value k, the same is done for the reached indices $R_j(T)$ with $j \geq i$.

Condensed Search Trees. Trip-Based Routing Using Condensed Search Trees (TB-CST) [26] employs Profile-TB to precompute search graphs which allow for extremely fast queries. The preprocessing phase solves the full-range profile problem for every possible pair of source and target stops by running a modified *one-to-all* version of Profile-TB from every stop. Consider the Profile-TB search for a source stop p_s. After each TB run, all newly found Pareto-optimal journeys are unpacked. This yields a breadth-first search tree with p_s as the root, trip segments as inner nodes, the reached target stops as leaves, and footpaths and transfers as edges. The search trees of all runs are merged into the *prefix tree* of p_s. Here, each trip segment $T[i, j]$ is replaced with a tuple (L, i) consisting of the line L with $T \in \mathcal{T}(L)$ and the stop index i where the line is entered.

To answer one-to-all queries, Profile-TB additionally maintains an earliest arrival time $\tau_{\text{arr}}(p, n)$ for each stop p and number of trips n. Like the reached indices, these arrival times are not reset between runs. When scanning a stop event $T[k]$ in round n, the algorithm iterates over all stops p with $\Delta\tau_{\text{fp}}(p(T[k]), p) < \infty$ and computes $\overline{\tau_{\text{arr}}} = \tau_{\text{arr}}(T[k]) + \Delta\tau_{\text{fp}}(p(T[k]), p)$. If $\overline{\tau_{\text{arr}}} < \tau_{\text{arr}}(p, n)$, then the best known journey to p with n trips was improved, so $\tau_{\text{arr}}(p, m)$ is set to $\min(\overline{\tau_{\text{arr}}}, \tau_{\text{arr}}(p, m))$ for all $m \geq n$.

To answer a query between source stop p_s and target stop p_t, TB-CST constructs a *query graph* from the prefix tree of p_s by extracting all paths which lead to a leaf representing p_t. Then a variant of Dijkstra's Algorithm is run on the query graph. Since the prefix tree only provides information about lines but not specific trips, these must be reconstructed during the query. When relaxing an edge from p_s to the first used line, the earliest reachable trip is identified based on the departure time at p_s. When relaxing an edge between lines L_1 and L_2, the used trip T_1 of L_1 is already known, so the algorithm explores the outgoing transfers of T_1 in $\mathfrak{T}$ to find the earliest reachable trip T_2 of L_2.

The space required to store all prefix trees can be reduced by extracting *postfix trees*. Consider the prefix tree for a source stop p_s. For each path from the root to a leaf representing a target stop p_t, a *cut node* is chosen. The subpath from the cut node to the leaf is then removed from the prefix tree of p_s and added to the postfix tree of p_t. Since many of these extracted subpaths occur in multiple prefix trees, moving them into a shared postfix tree considerably reduces memory consumption. To construct the query graph for a source stop p_s and target stop p_t, the prefix tree p_s and the postfix tree of p_t are spliced back together at the cut nodes.

Arc-Flags. Arc-Flags is a speedup technique for Dijkstra's Algorithm. Its basic idea is to precompute *flags* for each edge, which indicate whether the edge is necessary to reach a particular region of the graph. This allows Dijkstra's Algorithm to reduce the search space during a query by ignoring edges which are not flagged for the target region.

Given a weighted graph $G = (V, E, c)$, the preprocessing phase of Arc-Flags performs two steps: First, it computes a partition $r : V \to \{1, \dots, k\}$ of the node set into k cells, where k is a freely chosen parameter. Then, a *flags function* $b : E \times \{1, \dots, k\} \to \{0, 1\}$ is computed. Each individual value $b(e, i)$ for an edge e and a cell i is called a *flag*, hence the name Arc-Flags. The flags function must have the following property: for each pair of source node v_s and target node v_t, there is at least one shortest path P from v_s to v_t such that $b(e, r(v_t)) = 1$ for every edge e in P. With this precomputed information, a shortest path query between v_s and v_t can be answered by running Dijkstra's Algorithm but only relaxing edges e for which $b(e, r(v_t)) = 1$. The parameter k imposes a tradeoff between query speed and memory consumption. The space required to store the flags is in $\Theta(km)$, which is manageable for $k \ll n$. On the other hand, the search space of the query decreases for larger values of k since fewer flags will be set to 1 if the target cell is smaller.

Flags can be computed naively by solving the *all-pairs shortest path* problem, i.e., computing the shortest path between every pair of nodes. However, this requires $\Omega(n^2)$ precomputation time. The precomputation can be sped up by exploiting the observation that every shortest path that leads into a cell must pass through a boundary node. Thus, it is sufficient to compute backward shortest-path trees from all boundary nodes. For more details, see [20].

Arc-Flags for Public Transit Networks. Berger et al. [10] applied Arc-Flags to a time-dependent graph model in a problem setting which asks for *all* Pareto-optimal paths, including duplicates (i.e., Pareto-optimal paths which are equivalent in both criteria). They observed that for nearly every combination of edge e and cell i, there is at least one point in time during which e occurs on a Pareto-optimal path to a node in cell i. To solve this problem, the authors divided the service period of the network into two-hour intervals and computed a flag for each combination of edge, cell and time interval. However, this approach merely achieved a speedup of ≈ 3 over Dijkstra's Algorithm.

Time resolution is not an issue in time-expanded graphs, where each node is associated with a specific point in time. However, Delling et al. [15] observed a different problem when applying Arc-Flags to a time-expanded graph, even when optimizing only arrival time. Since the arrival time of a path depends only on its target node, all valid paths are optimal. Delling et al. therefore evaluated various tiebreaking strategies to decide which optimal paths should be flagged. The most successful strategy only achieved a speedup of ≈ 4 over Dijkstra's Algorithm. In the same paper, Delling et al. proposed a pruning technique called *Node-Blocking*, which applies the principle of line pruning to Dijkstra's Algorithm in time-expanded graphs. The authors observed that Node-Blocking conflicts with their tiebreaking choices for Arc-Flags, leading to incorrectly answered queries. Therefore, they only evaluated Arc-Flags without Node-Blocking.

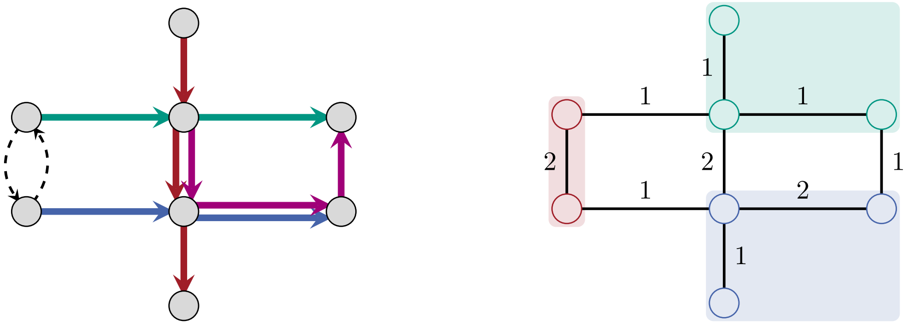

Figure 1 *Left:* An example network with stops as nodes, trips as colored solid edges and transfers as dashed edges. *Right:* The corresponding layout graph with edges weighted by the number of corresponding connections. Node groupings indicate a possible 3-way partition of the graph.

3 Arc-Flag TB

We now present the core ideas of our new algorithm Arc-Flag TB, which applies the main idea of Arc-Flags to TB. We first explain the general idea and then discuss details and optimizations. Finally, we compare our approach to similar algorithms.

The Arc-Flag TB precomputation performs two tasks: First it partitions the set $\mathcal{S}$ of stops into k cells, which yields a partition function $r : \mathcal{S} \to \{1, \dots, k\}$. Then it computes a flag for each transfer $\mathfrak{t} \in \mathfrak{T}$ and cell i which indicates whether $\mathfrak{t}$ is required to reach any target stops in cell i. Formally, this yields a flags function $b : \mathfrak{T} \times \{1, \dots, k\} \to \{0, 1\}$ with the following property: for each query with source stop p_s, target stop p_t and departure time τ_{dep}, there is a Pareto set $\mathcal{P}$ such that $b(\mathfrak{t}, r(p_t)) = 1$ for every transfer $\mathfrak{t} = T_a[i] \to T_b[j]$ that occurs in a journey $J \in \mathcal{P}$. A query between source stop p_s and target stop p_t is answered by running the TB query algorithm with one modification: a transfer $\mathfrak{t} \in \mathfrak{T}$ is only explored if the flag for the target cell is set to 1, i.e., $b(\mathfrak{t}, r(p_t)) = 1$.

3.1 Partitioning

To represent the topology of the public transit network without its time dependency, we define the *layout graph* G_L. The set of *connections* between a pair p, q of stops is given by

$$X(p, q) := \{T[i, i+1] \mid T \in \mathcal{T}, p(T[i]) = p, p(T[i+1]) = q\} \cup \{(p, q) \mid (p, q) \in \mathcal{F}\}\,.$$

Thus, a connection is either a trip segment between two consecutive stops or a footpath. Then the layout graph is defined as $G_L = (\mathcal{S}, E_L, c_L)$, with the set of edges $E_L \subseteq \mathcal{S} \times \mathcal{S}$ and edge weight function $c_L : E_L \to \mathbb{N}$ defined by

$$\begin{aligned} E_L &:= \{(p, q) \mid X(p, q) \neq \emptyset\}\,, \\ c_L\left((p, q)\right) &:= |X(p, q)|\,. \end{aligned}$$

An illustration of a layout graph is given in Figure 1. The stop partition r is obtained by generating the layout graph and running a graph partitioning algorithm of choice. Due to the weight function, the partitioning algorithm will attempt to avoid separating stops which have many connections between them.

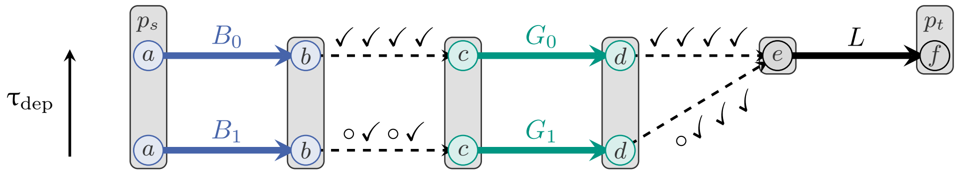

Figure 2 An example network illustrating the need for departure time fixing. Grey boxes represent stops. Nodes within the boxes represent stop events and are labeled with their indices along the respective trip. Within a stop, events are depicted in increasing order of time from bottom to top. Colored edges represent trips, with trips of the same line using the same color. Dashed edges with arrows represent transfers. Assume that p_t is the only stop in cell i of the stop partition r. For each transfer $\mathfrak{t}$, checkmarks indicate whether the flag of $\mathfrak{t}$ for cell i is set to 1. From left to right, these represent various configurations of the flag computation algorithm: unmodified Profile-TB, departure time buffering, flag augmentation, buffering + augmentation.

3.2 Flag Computation

To compute the flags, the full-range profile problem is solved for all pairs of source and target stops. As with TB-CST, this is done by running one-to-all Profile-TB search for every possible source stop. After each TB run of the Profile-TB search, all newly found journeys are unpacked. For a journey J to a target stop p_t and each transfer $\mathfrak{t}$ in J, the flag $b(\mathfrak{t}, r(p_t))$ is set to 1. Once all flags have been computed, transfers for which no flags are set to 1 can be removed from $\mathfrak{T}$.

Flag Compression. For an edge e, we call the set of flags $b(e, i)$ for $1 \leq i \leq k$ its *flag pattern*. Bauer et al. [6] observed for Arc-Flags on road networks that many edges in the graph share the same flag pattern. They exploit this with the following compression technique: All flag patterns which occur in the graph are stored in a global array A. For each edge e, the algorithm does not store the flag pattern of e directly, but rather the index i for which $A[i]$ holds the flag pattern of e. This significantly reduces memory consumption at the cost of an additional pointer access whenever an edge is relaxed. We also apply this compression technique in Arc-Flag TB and sort the flag pattern array in decreasing order of occurrence. This ensures that the most commonly accessed flag patterns are stored close together in memory, which increases the likelihood of cache hits.

3.3 Resolving Issues with Correctness

Departure Time Buffering. Due to line pruning, a TB query always enters the earliest reachable trip of a line; later trips of the same line are not explored. However, because Profile-TB processes departure times in decreasing order and applies self-pruning, it returns journeys which depart as late as possible. These two pruning rules conflict, leading to situations where Arc-Flag TB fails to find a Pareto-optimal journey. An example of this is shown in Figure 2. An unmodified Profile-TB search from p_s will find the journey $J_0 :=$ $\langle B_0[a, b], G_0[c, d], L[e, f]\rangle$ and flag it for cell i. However, it will not flag the journey $J_1 :=$ $\langle B_1[a, b], G_1[c, d], L[e, f]\rangle$, which has an earlier departure time and is therefore processed in a later run, but has the same arrival time and number of trips. An Arc-Flag TB query from p_s to p_t with departure time $\tau_{\mathrm{dep}}(B_1[a])$ will enter B_1 but not relax the unflagged transfer $B_1[b] \to G_1[c]$. While $B_0[b] \to G_0[c]$ is flagged, the query will not enter B_0 due to line pruning and therefore not relax this transfer either.

To solve this issue, we introduce the notion of the itinerary. An *itinerary* is a generalized description of a journey which specifies the lines used and the stop indices where they are entered and exited, but not the trips used. Corresponding to a stop event $T[i]$ is the *line event* $L[i]$ where $T \in \mathcal{T}(L)$. The stop visited by $L[i]$ is denoted as $p(L[i])$. A trip segment $T[i,j]$ corresponds to the *line segment* $L[i,j]$. An itinerary is therefore a sequence of line segments. The itinerary describing a journey $J = \langle T_1[b_1,e_1],\dots,T_k[b_k,e_k]\rangle$ is given by $\mathcal{I}(J) = \langle L_1[b_1,e_1],\dots,L_k[b_k,e_k]\rangle$, where $T_i \in \mathcal{T}(L_i)$ for $1 \le i \le k$. For a line L, an index i and a departure time τ_{dep}, let $T_{\min}(L,i,\tau_{\text{dep}})$ denote the earliest trip of L which departs at $p(L[i])$ no earlier than τ_{dep}. For an itinerary $\mathcal{I} = \langle L_1[b_1,e_1],\dots,L_k[b_k,e_k]\rangle$, the journey $J_{\min}(\mathcal{I},\tau_{\text{dep}})$ is the journey with itinerary $\mathcal{I}$ which takes the earliest reachable trip of every line when starting with departure time τ_{dep}. Formally, $J_{\min}(\mathcal{I},\tau_{\text{dep}}) = \langle T_1[b_1,e_1],\dots,T_k[b_k,e_k]\rangle$ with $T_i = T_{\min}(L_i,b_i,\tau^i_{\text{dep}})$ and

$$\tau^i_{\text{dep}} = \begin{cases} \tau_{\text{dep}} + \Delta\tau_{\text{fp}}(p_s, p(L_1[b_1])) & \text{if } i = 1,\\ \tau_{\text{arr}}(T_{i-1}[e_{i-1}]) + \Delta\tau_{\text{fp}}(p(L_{i-1}[e_{i-1}]), p(L_i[b_i])) & \text{otherwise.}\end{cases}$$

In Figure 2, J_0 and J_1 have the same itinerary $\mathcal{I}$. To ensure that the query from p_s to p_t with departure time $\tau_{\text{dep}}(B_1[a])$ is answered correctly by Arc-Flag TB, $J_{\min}(\mathcal{I},\tau_{\text{dep}}(B_1[a])) = J_1$ must be flagged as well. In general, consider the one-to-all Profile-TB search from a source stop p_s. For a TB run with departure time τ_{dep} and a target stop p_t, let $\mathcal{P}$ be the found Pareto set. We define the *buffered Pareto set*

$$\mathcal{P}_{\text{buf}}(\mathcal{P},\tau_{\text{dep}}) \coloneqq \{J_{\min}(\mathcal{I}(J),\tau_{\text{dep}}) \mid J \in \mathcal{P}\}.$$

Since $\mathcal{P}$ is a Pareto set, we know that every journey $J \in \mathcal{P}$ has the same arrival time as $J_{\min}(J,\tau_{\text{dep}})$. Hence, the last trip segment of $J_{\min}(\mathcal{I}(J),\tau_{\text{dep}})$ and J is always identical. However, for the other trip segments, $J_{\min}(\mathcal{I}(J),\tau_{\text{dep}})$ may use earlier trips than J. We modify the Profile-TB search to flag all transfers in $\mathcal{P}_{\text{buf}}(\mathcal{P},\tau_{\text{dep}})$. To do this efficiently, we employ an approach which we call *departure time buffering*. An itinerary $\mathcal{I}$ beginning with the line segment $L_1[b_1,e_1]$ is *unpacked* within the interval $(\tau_1,\tau_2]$ as follows: For a trip $T_1 \in \mathcal{T}(L_1)$, let $\tau_{\text{dep}}(\mathcal{I},T_1) \coloneqq \tau_{\text{dep}}(T_1[b_1]) - \Delta\tau_{\text{fp}}(p_s, p(T_1[b_1]))$ be the departure time of a journey with the itinerary $\mathcal{I}$ that uses T_1 as the first trip. For each trip $T_1 \in \mathcal{T}(L_1)$ with $\tau_{\text{dep}}(\mathcal{I},T_1) \in (\tau_1,\tau_2]$, the journey $J_{\min}(\mathcal{I},\tau_{\text{dep}}(\mathcal{I},T_1))$ is constructed and its transfers are flagged.

For each stop p and round n, the algorithm maintains not only the earliest arrival time $\tau_{\text{arr}}(p,n)$ but a buffered itinerary $\mathcal{I}(p,n)$, which represents the journey associated with $\tau_{\text{arr}}(p,n)$, as well as the departure time $\tau_{\text{dep}}(p,n)$ of the run in which $\tau_{\text{arr}}(p,n)$ was last changed. If $\tau_{\text{arr}}(p,n)$ is improved during a run with departure time τ_{dep}, then the journey corresponding to this arrival time is not flagged right away. Instead, after the end of the run, the algorithm unpacks the buffered itinerary $\mathcal{I}(p,n)$ within the interval $(\tau_{\text{dep}},\tau_{\text{dep}}(p,n))$ (unless $\mathcal{I}(p,n)$ has not been set before). Afterwards, the buffered itinerary $\mathcal{I}(p,n)$ is updated by unpacking the journey corresponding to the new value of $\tau_{\text{arr}}(p,n)$. After the last TB run of the profile search, the remaining buffered itineraries are processed. For every stop p and round n such that $\tau_{\text{arr}}(p,n) < \infty$, the itinerary $\mathcal{I}(p,n)$ is unpacked within the interval $(-\infty,\tau_{\text{dep}}(p,n))$.

Flag Augmentation. Departure time buffering does not fix all issues caused by the incompatibility between line pruning and self-pruning. Consider the example shown in Figure 3. Once again, an unmodified Profile-TB search from p_s will find the journey $J_0 \coloneqq \langle B[a,b], G_0[c,d], L[e,f]\rangle$ and flag it for cell i, whereas the equivalent

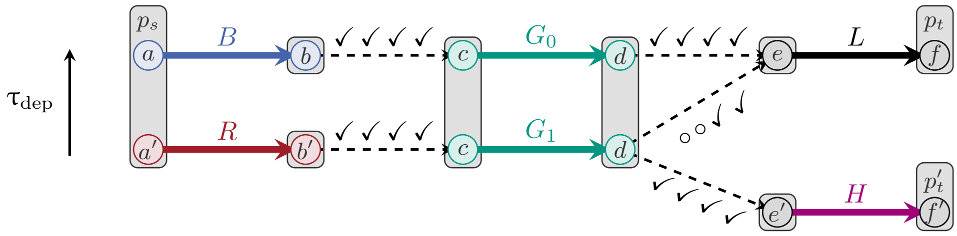

Figure 3 An example network illustrating the need for flag augmentation. Nodes, edges and checkmarks have the same meaning as in Figure 2. Assume that the stops p_t and p'_t are the only stops in cell i of the stop partition r.

journey $J_1 := \langle R[a', b'], G_1[c, d], L[e, f]\rangle$ is discarded. In this case, however, departure time buffering will not cause J_1 to be flagged either because it starts with a different line than J_0. Once again, consider an Arc-Flag TB query from p_s to p_t with departure time $\tau_{dep}(R[a'])$. If the transfer $R[b'] \to G_1[c]$ is not flagged, then the algorithm will enter G_0 and find J_0. However, in the example network, this transfer is flagged due to another journey $J'_1 := \langle R[a', b'], G_1[c, d], H[e', f']\rangle$, which leads to another target stop p'_t in cell i. As a consequence, G_1 is entered, but the unflagged transfer $G_1[d] \to L[e]$ is not relaxed, while G_0 is not entered due to line pruning.

To fix this issue, we define the *augmented flags function* $\hat{b} : \mathfrak{T} \times \{1, \ldots, k\} \to \{0, 1\}$. Consider a line L, a trip $T_a \in \mathcal{T}(L)$ and a transfer $\mathfrak{t} = T_a[i] \to T_b[j] \in \mathfrak{T}$. We define the set of *successor transfers* $\mathfrak{T}^\uparrow(\mathfrak{t})$ as

$$\mathfrak{T}^\uparrow(T_a[i] \to T_b[j]) := \{T'_a[i] \to T_b[j] \in \mathfrak{T} \mid T'_a \in \mathcal{T}(L), T_a \preceq T'_a\}.$$

Then $\hat{b}$ is defined as follows for a transfer $\mathfrak{t} \in \mathfrak{T}$ and cell i:

$$\hat{b}(\mathfrak{t}, i) := \bigvee_{\mathfrak{t}' \in \mathfrak{T}^\uparrow(\mathfrak{t})} b(\mathfrak{t}, i)$$

In the example from Figure 3, using $\hat{b}$ instead of b resolves the problem, provided that all transfers which occur in J_1 are included in the set $\mathfrak{T}$ of transfers generated by the TB preprocessing phase. Note that flag augmentation alone (without departure time buffering) will not fix the issue shown in Figure 2 because the transfer $B_1[b] \to G_1[c]$ will not be flagged. Thus, both fixes must be combined.

TB Transfer Precomputation. A final issue is due to the TB preprocessing phase, which computes the set $\mathfrak{T}$ of transfers. Unfortunately, its rules for pruning unnecessary transfers are too strong to guarantee that the transfers required by the Arc-Flag TB preprocessing are always generated. It is possible to construct examples akin to Figure 2 where the transfer $G_1[d] \to L[f]$ is not included in $\mathfrak{T}$, e.g., because a transfer to a different trip than L is preferred. In this case, both Arc-Flag TB and TB-CST[1] will fail to return correct results. Adapting the pruning rules of the TB preprocessing phase to resolve these issues remains an open problem.

To show that they can be resolved in principle, we implemented a prototypical variant of the Arc-Flag TB preprocessing that performs the profile searches with rRAPTOR [16], the profile variant of RAPTOR. While rRAPTOR does not rely on precomputed transfers,

[1] We reported this issue to the author of TB-CST, who concurred with our findings. The original TB-CST publication [26] mainly focused on evaluating profile queries, where this issue does not occur.

it suffers from the same conflict between line pruning and self-pruning as Profile-TB. This conflict was previously noticed and resolved in the context of ULTRA [8, 9], a preprocessing technique for multimodal journey planning which is based on rRAPTOR. We applied the modifications proposed for ULTRA to our rRAPTOR implementation. The resulting variant of Arc-Flag TB has significantly higher preprocessing times than the Profile-TB-based one but answered all queries correctly in our preliminary experiments.

3.4 Comparison

We conclude this section by comparing Arc-Flag TB to similar approaches.

TB-CST. TB-CST (without split trees) stores a generalized shortest path tree for every possible source stop. This offers a near-perfect reduction in the query search space but at the expense of quadratic memory consumption. The memory consumption of Arc-Flag TB is in $\Theta(|\mathfrak{T}|\,k)$, where k is the number of cells. Thus, Arc-Flag TB can be seen as a way to interpolate between TB and TB-CST regarding query search space and memory consumption. For $k = 1$, every non-superfluous transfer will be flagged, and thus the search space will be identical to that of TB with a minimal set of transfers. For $k = n$, the flags provide perfect information about whether a transfer is required to reach the target node.

An advantage of our approach is that the transfer flags provide information about which specific trips should be entered. In contrast, the TB-CST search graph only provides information about entire lines. This means that Arc-Flag TB does not have to invest additional effort during the query phase to find the earliest reachable trip of each line.

Time-Expanded Arc-Flags. Conceptually, our approach is similar to Arc-Flags on a time-expanded graph, albeit with TB as a query algorithm instead of Dijkstra's Algorithm. Delling et al. [15] observed low speedups when applying Arc-Flags to time-expanded graphs. We analyze some of the issues causing this and how Arc-Flag TB overcomes them. In a time-expanded graph, each visit of a vehicle at a stop is modeled with three nodes: an *arrival node*, a *transfer node* and a *departure node*. A journey corresponds to a path between two transfer nodes. However, boundary nodes in the partition may also be departure or arrival nodes. Consider, for example, a boundary node v of cell i which is an arrival node corresponding to the stop event $T[i]$. Arc-Flags will compute and flag a backward shortest-path tree rooted in v. A path in this tree corresponds to a "journey" which ends with the passenger remaining seated in T. However, there is no guarantee that this path can be extended to a Pareto-optimal journey which ends at a transfer node in cell i. Entering T may never be required to enter cell i. In this case, Arc-Flags produces superfluous flags. Arc-Flag TB avoids this problem by performing a one-to-all profile search from all stops, including those which are not boundary nodes in the layout graph. While this requires $\Omega(|\mathcal{S}|^2)$ preprocessing time, it considerably reduces the number of set flags.

Another feature of Arc-Flag TB that reduces the search space is that it flags transfers between stop events. In the time-expanded graph, a transfer corresponds to an entire path between an arrival and a departure node, which may pass through several transfer nodes. Consider two flagged transfers $T_1[i_1] \to T_2[i_2]$ and $T_3[i_3] \to T_4[i_4]$ whose paths in the time-expanded graph intersect. This has the effect of creating "virtual" transfers $T_1[i_1] \to T_4[i_4]$ and $T_3[i_3] \to T_2[i_2]$, which may not be flagged. Arc-Flags on the time-expanded graph will explore these transfers, whereas Arc-Flag TB will not. Note that Arc-Flag TB only flags transfers, not trip segments. This is because flagging trip segments would not provide any benefit beyond the first round of an Arc-Flag TB query: If a

Table 1 An overview of the networks on which we performed our experiments. Stops, lines, trips and footpaths are from the GTFS datasets. Transfers were generated by the TB precomputation.

Network	Stops	Lines	Trips	Footpaths	Transfers
Germany	441 465	207 801	1 559 118	1 172 464	60 919 877
Paris	41 757	9 558	215 526	445 912	23 284 123
Sweden	48 007	15 627	248 977	2 118	14 771 466
Switzerland	30 861	18 235	559 752	20 864	9 142 826

trip segment is not flagged for a specific cell, then neither are its incoming or outgoing transfers. Thus, an unflagged trip segment can only be entered during the first round, and no further trip segments are reachable from there.

Finally, Delling et al. note that all paths in a time-expanded graph have optimal arrival time and that a speedup is only achieved with suitable tiebreaking choices between equivalent paths. They observed that their tiebreaking choices conflicted with their implementation of line pruning, Node-Blocking. In Arc-Flag TB, the tiebreaking choices are dictated by the self-pruning of Profile-TB. While this also produces conflicts with line pruning, we resolved them by applying departure time buffering and flag augmentation. This allows Arc-Flag TB to fully benefit from both pruning rules, unlike previous approaches.

4 Experimental Evaluation

We evaluate the performance of Arc-Flag TB on a selection of real-world public transit networks. All experiments were run on a machine equipped with an `AMD EPYC 7702P` CPU with 64 cores, 128 threads, and 1 TB of RAM. Code for TB and Arc-Flag TB was written in `C++` and compiled using GCC with optimizations enabled (`-march=native -O3`). For TB-CST, we used the original code provided to us by the author [25]. The preprocessing phases of TB-CST and Arc-Flag TB, which run one-to-all Profile-TB from each stop, were parallelized with 128 threads.

Our datasets are taken from GTFS feeds of the public transit networks of Germany[2], Paris[3], Sweden[4] and Switzerland[5]. Details are listed in Table 1. For each network, we extracted the timetable of two consecutive weekdays to allow for overnight journeys.

Partitioning. We use the KaHIP[6] [24] open-source graph partitioning library to partition our networks. KaHIP is based on a multilevel approach, i.e., the input graph is coarsened, initially partitioned, and locally improved during uncoarsening. In our experiments, overall better results are obtained when coarsening is computed using clustering rather than edge matching as usual. More specifically, we use the memetic algorithm `kaffpaE` with the strong social configuration and an imbalance parameter of 5% in all of our experiments. As a time limit, we set 10 minutes for all networks regardless of the number k of desired cells. In our experiments, higher time limits did not significantly improve the results regarding the total number of flags set and average query times.

[2] `https://gtfs.de/`
[3] © `https://navitia.io/`
[4] `https://trafiklab.se/`
[5] `https://opentransportdata.swiss/`
[6] `https://github.com/KaHIP/KaHIP`

Table 2 Performance of Arc-Flag TB depending on the number of cells k. Departure time fixing and flag augmentation are enabled for all experiments. Query times and success rates are averaged over 10 000 random queries. Success rates are the percentage of queries for which Arc-Flag TB found a correct Pareto set, and the percentage of journeys in the correct Pareto sets for which Arc-Flag TB found an equivalent journey, respectively. Query times and memory consumption are measured with and without flag compression. Note that the preprocessing time does not include the partitioning, which was limited to 10 minutes in all configurations. Query times for $k = 1$ are for TB.

Network	k	Prepro. [hh:mm:ss]	Query time [μs]		Memory [MB]		Success rate [%]	
			Uncomp.	Comp.	Uncomp.	Comp.	Queries	Journeys
Germany	1	–	105 809	–	–	–	–	–
	1 024	34:12:44	739	1 068	5 120	725	93.53	95.12
	2 048	34:09:26	578	912	8 704	1 126	93.02	94.74
	4 096	34:10:59	548	745	16 384	2 474	92.22	94.24
Paris	1	–	4 502	–	–	–	–	–
	64	00:37:15	865	1 528	450	95	99.17	99.30
	128	00:37:19	671	1 486	513	133	99.08	99.25
	256	00:37:29	502	1 230	639	187	98.98	99.18
	512	00:37:35	393	982	891	282	98.76	99.03
	1 024	00:37:45	331	757	1 434	468	98.43	98.78
Sweden	1	–	7 583	–	–	–	–	–
	64	00:17:24	265	288	376	60	95.77	96.61
	128	00:17:26	167	202	428	69	95.36	96.24
	256	00:17:27	121	164	534	84	95.14	96.01
	512	00:17:30	97	140	744	116	94.86	95.82
	1 024	00:17:32	88	127	1 229	184	94.37	95.49
Switzerland	1	–	7 043	–	–	–	–	–
	64	00:13:07	223	225	222	35	96.74	97.53
	128	00:13:08	154	172	253	40	96.29	97.23
	256	00:13:09	112	136	315	50	95.75	96.86
	512	00:13:13	91	118	440	70	95.29	96.60
	1 024	00:13:14	81	108	698	114	94.80	96.29

Arc-Flag TB Performance. Performance measurements for Arc-Flag TB, including the impact of flag compression and the number of cells k, are shown in Table 2. For each configuration, we performed 10 000 queries with the source and target stops chosen uniformly at random and the departure time chosen uniformly at random within the first day of the timetable. As expected, the preprocessing time is mostly unaffected by k. Without flag compression and with the highest number of cells, Arc-Flag TB achieves a speedup of 193.1 on Germany, 13.6 on Paris, 86.2 on Sweden and 87.0 on Switzerland. Even without compression, the memory consumption for the computed flags is moderate at roughly 1 GB for the smaller networks and 16 GB for Germany. On all networks except Paris, flag compression is very effective: it reduces the memory consumption by a factor of 6–8 at the expense of 20–40% of additional query time. On Paris, the compression is less successful but still reduces the memory consumption by a factor of 3 while roughly doubling the query time. Figure 4 plots the speedup over TB and the memory consumption, with and without flag compression, depending on k. While the performance gains from doubling the

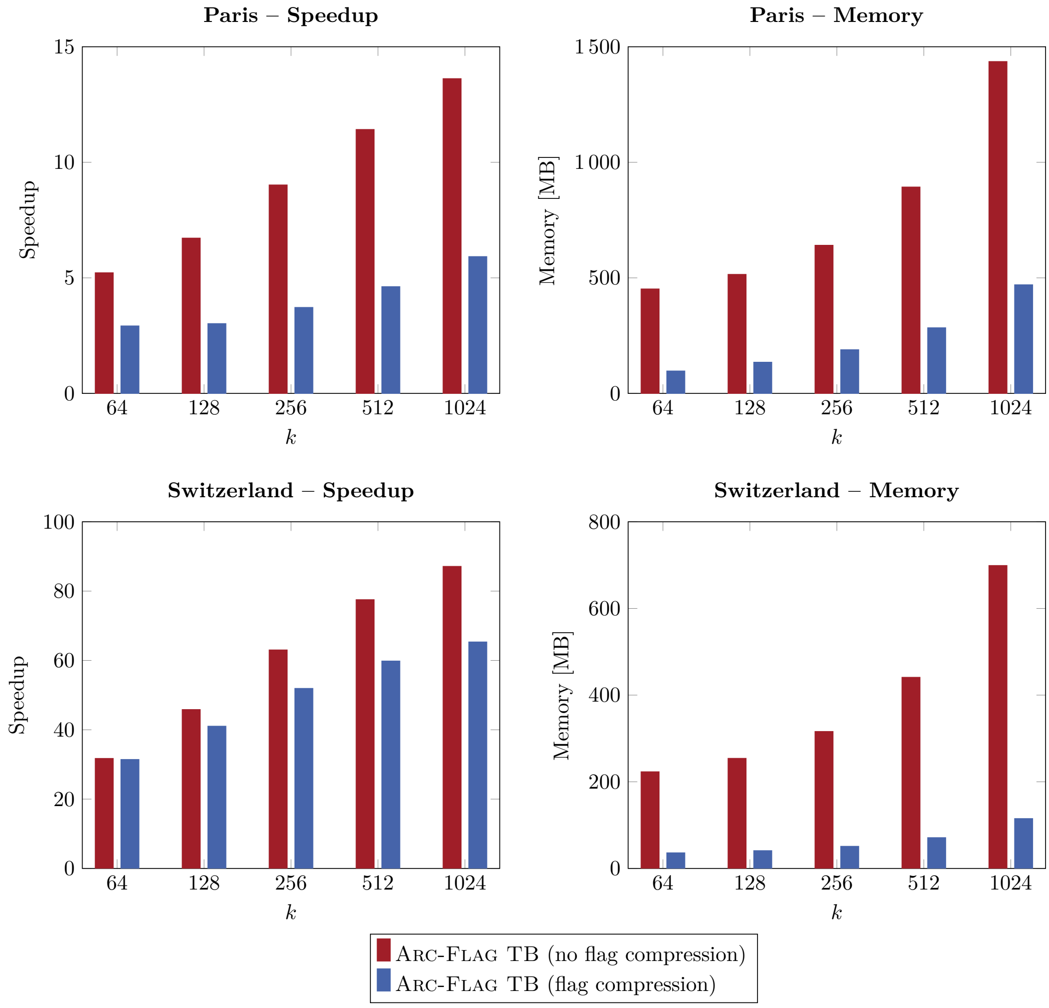

Figure 4 Average speedup over TB and memory consumption of Arc-Flag TB (with and without flag compression) on the Paris and Switzerland networks, depending on the number of cells k.

number of cells eventually decline, they still remain strong up to $k = 1\,024$. The rate of incorrectly answered queries is around 5% on the country networks and 1% on Paris, and only slightly increases with k. The lower speedup for Paris is explained by the fact that it is a dense metropolitan network with a less hierarchical structure and, therefore, harder to partition. Similar discrepancies in the performance between metropolitan networks and country networks were observed for Transfer Patterns [3] and TB-CST [26].

Result Quality. Table 3 shows the impact of departure time buffering and flag augmentation on the result quality of Arc-Flag TB. Departure time buffering significantly increases the preprocessing time, but this pays off in terms of the error rate, which is reduced from almost 30% to 6%. Flag augmentation on its own also reduces the number of incorrectly answered queries, but not as much. Combining both only slightly reduces the error rate compared to departure time buffering alone, which indicates that the scenario depicted in Figure 3 is rare. The results for our prototypical rRAPTOR-based preprocessing

Table 3 Impact of the preprocessing algorithm, departure time buffering (Buf.) and flag augmentation (Aug.) on the performance and success rate of Arc-Flag TB, measured on the Switzerland network with $k = 1\,024$. Query times are measured without flag compression.

Algorithm	Buf.	Aug.	Prepro. [hh:mm:ss]	Query time [μs]	Success rate [%]	
					Queries	Journeys
Profile-TB	○	○	00:05:03	46	70.37	75.54
Profile-TB	○	●	00:05:05	56	81.87	85.39
Profile-TB	●	○	00:13:13	77	94.05	95.70
Profile-TB	●	●	00:13:14	81	94.80	96.29
rRAPTOR	–	–	01:26:51	58	100.00	100.00

algorithm are promising: While the preprocessing times are not practical, all queries are answered correctly. Furthermore, query times actually decrease compared to Profile-TB with buffering since flags are no longer set unnecessarily.

TB-CST. Finally, we compare Arc-Flag TB against Witt's implementation of TB-CST with split trees [26] on our networks. The results are shown in Table 4. We do not report the performance of TB-CST with unsplit prefix trees since the precomputed data requires over 100 GB of memory even on the smaller networks. Therefore, a comparison would not be fair. Excluding the partitioning step, which always took 10 minutes in our experiments, the precomputation time of Arc-Flag TB is 2–6 times higher, depending on the network. Although both techniques perform a one-to-all Profile-TB search from every stop, our algorithm additionally performs departure time buffering, which increases the precomputation time. The remaining difference, which amounts to a factor of 2 on the Switzerland network, is due to the fact that our implementation of Profile-TB is less optimized than Witt's. The memory consumption of Arc-Flag TB is much lower than that of TB-CST, even with 1 024 cells and without flag compression. On the three smaller networks, our query times are similar or better. A proper comparison for the Germany network is difficult because we were not able to execute the provided code for TB-CST on this instance, and the query times reported in the original paper [26] are for a smaller version of the network. Nevertheless, we observe that Arc-Flag TB with $k = 2\,048$ is at most four times slower than TB-CST while consuming less than a tenth of the space.

Overall, Arc-Flag TB matches the query performance of TB-CST while requiring much less space. This is for two reasons: Firstly, the query time of TB-CST is dominated by the time required to construct the query graph. Arc-Flag TB does not require this step. Secondly, the TB-CST query algorithm must reconstruct the earliest reachable trip of each used line at query time, whereas Arc-Flag TB can rely directly on the precomputed transfers. Furthermore, we observe that TB-CST has a much higher error rate on Sweden and Switzerland than Arc-Flag TB. We expect that this is because Arc-Flag TB aggregates the flags by cell. Thus, even if the precomputation fails to find a required journey to a particular target stop, the transfers in that journey may still be flagged if they occur in journeys to other target stops in the same cell.

Table 4 Performance of TB-CST with split trees for 10 000 random queries. Note that we were not able to run TB-CST queries on the Germany network due to issues with the provided code. We instead list the query time reported in [26].

Network	Prepro. [hh:mm:ss]	Query time [μs]	Memory [MB]	Success rate [%]	
				Queries	Journeys
Germany	06:36:27	(156)	114 080	–	–
Paris	00:20:30	507	6 992	98.98	99.05
Sweden	00:07:42	91	3 400	75.99	91.67
Switzerland	00:02:22	66	1 586	80.72	89.88

5 Conclusion

We developed Arc-Flag TB, a speedup technique for public transit journey planning which combines Arc-Flags and Trip-Based Public Transit Routing (TB). We demonstrated that the stronger pruning rules of TB allow our approach to overcome previous obstacles in applying Arc-Flags to public transit networks. This allows Arc-Flag TB to achieve up to two orders of magnitude speedup over TB. Compared to TB-CST, a state-of-the-art speedup technique for TB, our algorithm achieves roughly the same query times with a similar precomputation time and only a fraction of the memory consumption. Unlike TB-CST, the query performance and memory consumption are configurable via the number of cells in the computed network partition. Currently, both algorithms answer some queries incorrectly due to an issue with the TB precomputation phase. However, we showed that the error rate of Arc-Flag TB is low and presented a prototypical variant of the algorithm which answers all queries correctly. In the future, it would be interesting to examine whether the performance of Arc-Flag TB can still be achieved with a subquadratic precomputation phase which only runs searches from the boundary nodes of the partition.

References

1 Prateek Agarwal and Tarun Rambha. Scalable Algorithms for Bicriterion Trip-Based Transit Routing. Technical report, Department of Civil Engineering, Indian Institute of Science, 2022. `arXiv:2111.06654`.

2 Hannah Bast. Car or Public Transport – Two Worlds. In *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science (LNCS)*, pages 355–367. Springer, 2009. `doi:10.1007/978-3-642-03456-5_24`.

3 Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast Routing in Very Large Public Transportation Networks using Transfer Patterns. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*, volume 6346 of *Lecture Notes in Computer Science (LNCS)*, pages 290–301. Springer, 2010. `doi:10.1007/978-3-642-15775-2_25`.

4 Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. In *Algorithm Engineering: Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science (LNCS)*, pages 19–80. Springer, 2016. `doi:10.1007/978-3-319-49487-6_2`.

5 Hannah Bast, Matthias Hertel, and Sabine Storandt. Scalable Transfer Patterns. In *Proc. 18th Workshop on Algorithm Engineering and Experiments (ALENEX'16)*, pages 15–29. Society for Industrial and Applied Mathematics (SIAM), 2016. `doi:10.1137/1.9781611974317.2`.

6 Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. *Journal of Experimental Algorithmics (JEA)*, 14:4.1–4.29, 2009. doi:10.1145/1498698.1537599.

7 Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study of Speed Up Techniques for Timetable Information Systems. *Networks*, 57:38–52, 2011. doi:10.1002/net.20382.

8 Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA'19)*, volume 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.14.

9 Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution. Technical report, ITI Wagner, Department of Informatics, Karlsruhe Institute of Technology (KIT), 2023. arXiv:1906.04832.

10 Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller-Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'09)*, volume 12 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2009. doi:10.4230/OASIcs.ATMOS.2009.2148.

11 Gerth S. Brodal and Riko Jacob. Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries. In *Proceedings of the 3rd Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'03)*, volume 92, pages 3–15. Elsevier, 2004. doi:10.1016/j.entcs.2003.12.019.

12 Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and Distance Queries via 2-Hop Labels. *SIAM Journal on Computing (SICOMP)*, 32(5):1338–1355, 2003. doi:10.1137/S0097539702403098.

13 Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. Public Transit Labeling. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, volume 9125 of *Lecture Notes in Computer Science (LNCS)*, pages 273–285. Springer, 2015. doi:10.1007/978-3-319-20086-6_21.

14 Daniel Delling, Julian Dibbelt, Thomas Pajor, and Tobias Zündorf. Faster Transit Routing by Hyper Partitioning. In *Proceedings of the 17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'17)*, volume 59 of *OpenAccess Series in Informatics (OASIcs)*, pages 8:1–8:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/OASIcs.ATMOS.2017.8.

15 Daniel Delling, Thomas Pajor, and Dorothea Wagner. Engineering Time-Expanded Graphs for Faster Timetable Information. In *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems*, volume 5868 of *Lecture Notes in Computer Science (LNCS)*, pages 182–206. Springer, 2009. doi:10.1007/978-3-642-05465-5_7.

16 Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. *Transportation Science*, 49:591–604, 2015. doi:10.1287/trsc.2014.0534.

17 Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959. doi:10.1007/BF01386390.

18 Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria Shortest Paths in Time-Dependent Train Networks. In *Proceedings of the 7th International Workshop on Experimental and Efficient Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science (LNCS)*, pages 347–361. Springer, 2008. doi:10.1007/978-3-540-68552-4_26.

19 Pierre Hansen. Bicriterion Path Problems. In *Multiple Criteria Decision Making Theory and Application*, volume 177 of *Lecture Notes in Economics and Mathematical Systems*, pages 109–127. Springer, 1980. doi:10.1007/978-3-642-48782-8_9.

20 Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 41–72. American Mathematical Society (AMS), 2009. doi:10.1090/dimacs/074/03.

21 Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speed Up Dijkstra's Algorithm. *Journal of Experimental Algorithmics (JEA)*, 11:2.8:1–2.8:29, 2006. doi:10.1007/11427186_18.

22 Matthias Müller-Hannemann and Mathias Schnee. Finding All Attractive Train Connections by Multi-Criteria Pareto Search. In *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science (LNCS)*, pages 246–263. Springer, 2007. doi:10.1007/978-3-540-74247-0_13.

23 Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *Journal of Experimental Algorithmics (JEA)*, 12:2.4:1–2.4:39, 2008. doi:10.1145/1227161.1227166.

24 Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science (LNCS)*, pages 164–175. Springer, 2013. doi:10.1007/978-3-642-38527-8_16.

25 Sascha Witt. Trip-Based Public Transit Routing. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*, volume 9294 of *Lecture Notes in Computer Science (LNCS)*, pages 1025–1036. Springer, 2015. doi:10.1007/978-3-662-48350-3_85.

26 Sascha Witt. Trip-Based Public Transit Routing Using Condensed Search Trees. In *Proc. 16th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'16)*, volume 54 of *OpenAccess Series in Informatics (OASIcs)*, pages 10:1–10:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASIcs.ATMOS.2016.10.