Efficient and Exact Public Transport Routing via a Transfer Connection Database

Abdallah Abuaisha, Mark Wallace, Daniel Harabor, Bojie Shen

Department of Data Science and Artificial Intelligence, Monash University, Australia
{abdallah.abuaisha, mark.wallace, daniel.harabor, bojie.shen}@monash.edu

Abstract

We explore the earliest arrival time problem in public transport journey planning. A journey typically consists of multiple scheduled public transport legs. The actual time required to transfer between these legs can substantially influence route planning. Therefore, we properly model transfers by incorporating their exact costs. We then introduce a novel oracle-based routing algorithm that constructs an efficient transfer database, considering the proposed transfer model. The database is leveraged online to quickly reconstruct the optimal journey in response to an earliest arrival time query. Our experimental results show that neglecting exact transfer costs often lead to either infeasible or suboptimal route plans. Furthermore, the findings highlight the efficiency of our algorithm in handling queries, demonstrated by response times within mere microseconds.

Introduction

We study the earliest arrival time problem in public transport routing, a well-known research problem in the literature. A public transport journey consists of one or more scheduled trips. Most works in the literature assume uniform transfer cost when changing vehicles within stations and/or between nearby stations. However, the actual time required to transfer between stops can have a large impact on the feasibility of the connection and the arrival time at the destination. In Figure 1, we show results from an experiment involving trip planning for the metropolitan area of Paris. We optimally solve 40,000 queries throughout a single day, modelling transfer costs in three different ways: exact, min, and max. The figure shows that underestimating transfer costs using min produces infeasibility for around 35% of journeys. Meanwhile, max introduces unnecessary delays for around 45% of all queries, in some cases up to several hours. Other fixed transfer time models produce similar results. Clearly, exact transfer costs are preferable.

In public transport routing, a central server is typically responsible for managing a high volume of queries from passengers across the network. Effective algorithms in the space must respond to such queries as quickly as possible. Many works exist on this topic, including a group of algorithms which are fast enough for practical applications (Geisberger 2010; Bast et al. 2010; Dibbelt et al. 2018; Delling, Pajor, and Werneck 2015; Delling et al. 2015). Unfortunately, these works do not delve into the details of transfer modelling. Instead, they opt for simplified models that assume uniform transfer costs within and/or between stations, although some of them could be adapted to handle variable transfer costs. A different group of algorithms also exists that allows for unlimited walking within the network and can compute optimal solutions with exact transfer costs (Phan and Viennot 2019; Delling et al. 2013; Giannakopoulou, Paraskevopoulos, and Zaroliagis 2019). However, this advantage comes at the expense of more complicated algorithms and considerably slower query times.

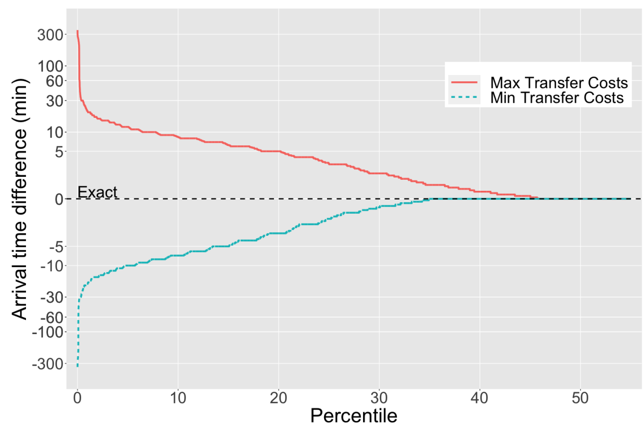

Figure 1: Illustration of arrival time difference when adopting different models for transfer costs. Three variants are explored: exact, minimum, and maximum transfer costs (within and between stations). We run 40,000 queries over a single day on the Paris metropolitan network, covering trains, trams, buses, and limited walking.

In this paper, we present a new approach for public transport routing which produces optimal solutions with exact transfer costs, similar to algorithms in the second group. Suitable for deployment in a centralised context, our approach is two to three orders of magnitude faster than the state-of-the-art algorithms in the first group. More specifically, our main contributions are: (i) introducing a novel algorithm that solves the earliest arrival time problem, both

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

accurately and efficiently; (ii) demonstrating the importance of modelling transfers using exact transfer costs; and (iii) proposing an efficient method for building a compressed database of optimal journeys in a public transport network. Our algorithm consists of two phases. In the offline phase, we build an efficient database of all optimal journeys, considering exact transfer times. The database encodes the first segment on the optimal journeys for each origin-destination (OD) pair across the different departure times. In the online phase, this database is utilised to quickly reconstruct the optimal journey that answers an earliest arrival time (EAT) query. Our experiments highlight the efficiency of our algorithm. Furthermore, the results emphasise the importance of adopting exact transfer models in providing feasible optimal solutions compared to simplified models.

Background

Timetable is the main input for any public transport journey planning system. Similar to existing works (Dibbelt et al. 2018; Delling, Pajor, and Werneck 2015), we model the timetable by directly utilising its structure, rather than constructing a graph. We represent a timetable as $TB = \{P, C, T, S, F\}$ with stops P, connections C, and trips T. Stations S and footpaths F are introduced to better model the transfers. A stop $p \in P$ is a departure and/or arrival point where a transport vehicle stops to pick up and/or drop off passengers. A typical example of a stop is a platform in a train station or a shelter in a bus stop. A connection $c \in C$ is represented as a 5-tuple $(p_{dep}, \tau_{dep}, p_{arr}, \tau_{arr}, t)$, indicating an event in which a transport vehicle departs from stop $p_{dep} \in P$ at time τ_{dep} and arrives at stop $p_{arr} \in P$ at time τ_{arr} via trip $t \in T$, without any intermediate halt between the two stops. Note that $p_{dep} \neq p_{arr}$ and $\tau_{dep} < \tau_{arr}$ always hold. A trip $t \in T$ corresponds to a scheduled transport vehicle that visits a specific order of stops. Such trip can be represented by a sequence of connections $\langle c_0, c_1, \ldots, c_k \rangle$ satisfying the following conditions: (i) $t(c_0) = t(c_1) = \ldots = t(c_k)$; (ii) $p_{dep}(c_i) = p_{arr}(c_{i-1})$; and (iii) $\tau_{dep}(c_i) \geq \tau_{arr}(c_{i-1})$, for all $i \in \{1 \ldots k\}$. Note that we frequently use the notation $a(b)$ throughout the paper, denoting “a of b”, unless otherwise stated. For instance, $t(c_i)$ corresponds to the trip of connection c_i. A station $s \in S$ is formed by a group of nearby stops which serve the same public transport mode and share a unique common name. These stops are then called the child stops of s, denoted as $p(s) = \{p_1, p_2, \ldots, p_n\} \subseteq P$. A footpath $f \in F$ is represented as $f = (p_i, p_j, \delta\tau(p_i \to p_j))$, which indicates the possibility of transferring (i.e., walking) from stop p_i to stop p_j, with a transfer duration of $\delta\tau(p_i \to p_j)$. Following previous works (Dibbelt et al. 2018; Delling, Pajor, and Werneck 2015), we naturally assume that footpaths in public transport networks satisfy the transitive closure property. This property implies that if there exists a sequence of footpaths between two locations, then the network must also include a direct footpath between these locations. Formally, for all $p_u, p_v \in P$, if there exists $p_0, p_1, \ldots, p_n$ such that $(p_u, p_0, \delta\tau(p_u \to p_0)), (p_0, p_1, \delta\tau(p_0 \to p_1)), \ldots, (p_n, p_v, \delta\tau(p_n \to p_v)) \in F$, then $(p_u, p_v, \delta\tau(p_u \to p_v)) \in F$. The details of footpath modelling will be discussed later.

In the context of public transport routing, the specific commencing (concluding) stop at the origin (destination) station typically holds minimal significance for users. Based on this observation, we define our queries on a station-level. Given an EAT query $q = (s_o, s_d, \tau_q)$, our objective is to identify a feasible journey J that departs from the origin station s_o no earlier than time τ_q and arrives at the destination station s_d as early as possible. Such journey J from s_o to s_d consists of a sequence of connections $J = \langle c_0, \ldots, c_n \rangle$, where $p_{dep}(c_0) \in p(s_o)$ and $p_{arr}(c_n) \in p(s_d)$. For every two consecutive connections, they either share the same trip $t \in T$ (i.e., do not involve a vehicle change), or else require a transfer via a footpath $f \in F$. As footpaths F are transitively closed, a transfer never requires more than one footpath. For two connections between which a transfer is required, the transfer must be feasible respecting the required transfer time. Formally, $\tau_{arr}(c_{i-1}) + \delta\tau(p_{arr}(c_{i-1}) \to p_{dep}(c_i)) \leq \tau_{dep}(c_i)$ must always hold for all $i \in \{1 \ldots n\}$.

Connection Scan Algorithm (CSA) CSA is a state-of-the-art online algorithm designed to mainly address the earliest arrival time problem (Dibbelt et al. 2018). CSA assembles all connections from a timetable TB into a single array C, sorted by their departure times (i.e., from earliest to latest). This enables CSA to efficiently answer queries by scanning through the sorted C array. Given an EAT query with a departure time τ_q, CSA maintains an array EA to store the tentative earliest arrival time for each stop $p \in P$. Initially, EA at the source is set to τ_q, while it is set to infinity for all other stops. The algorithm then proceeds to scan all connections in C that depart after τ_q in order. For each scanned connection c, the algorithm first checks its reachability. A connection is considered *reachable* if there exists a way to catch it on time (i.e., $EA[p_{dep}(c)] \leq \tau_{dep}(c)$). If the connection is reachable, the algorithm updates EA for the arrival stop $p_{arr}(c)$. It also accounts for transfers, updating EA for each reachable stop p_j via a footpath $f = (p_{arr}(c), p_j, \delta\tau(p_{arr}(c) \to p_j)$ as $EA[p_j] = \min(EA[p_j], \tau_{arr}(c) + \delta\tau(p_{arr}(c) \to p_j))$. However, if the connection c is not reachable, the algorithm simply skips it and proceeds to the next one. Finally, at the end of the scanning process, the algorithm returns EA at the destination. To extract full journeys, the algorithm is augmented by journey pointers or a post-processing phase is performed.

Transfers Modelling

One important aspect of modeling public transport networks is transfers where passengers change vehicles during their journey. Modeling the details of transfers vary significantly across the literature. Unexpectedly, this can have a great impact on the quality and feasibility of the solutions (as shown in the experiments sections). We noticed that transfer models in most existing works can be divided into two main types: (i) intra-station model where transfers only occur between stops within stations, either at no cost or at a uniform station-specific transfer time; and (ii) inter-station model where transfers can in addition happen between nearby stations, considering only a uniform transfer time for each station

pair. Both transfer models assume uniform transfer times, which can lead to the computation of infeasible or suboptimal solutions. On the other hand, there exists another type that incorporates variable transfer times by creating a comprehensive walking graph that connects all stops in P. Nevertheless, the computational burden and query time significantly increase for such a model due to the extensive integration of footpaths within this unrestricted walking scenario. To overcome the limitations of the existing transfer models, we propose our transfer model.

Our Model In public transport networks, transfers between stops within a station are essential. Therefore, our model first includes a footpath between every pair of stops that belong to the same station. Our model determines the exact transfer time of each footpath. To enable transfers between stops belonging to different stations, our model is built on the assumption that the majority of users are willing to walk a reasonable distance for the purpose of transferring. Specifically, we impose a global range limit that restricts transfers within a predefined radius r. For any station $s_j \in S$ that is located within r from a given station $s_i \in S$, a footpath from each stop in $p(s_i)$ to each stop in $p(s_j)$ is constructed considering the exact required transfer time. Due to the transitive closure property, we incorporate extra footpaths to ensure the completeness of the network. Such footpaths must always satisfy the triangle inequality theorem. Finally, given the modeled footpaths F, we associate each station $s \in S$ with a list of the neighbouring stations, $n(s) = \{s_1, s_2, \ldots, s_m\} \subseteq S$, to which there exists at least one footpath from s.

Transfer Connection Database (TCD)

Our approach, is centered around the key idea inspired by the Compressed Path Database (CPD) (Botea 2011; Botea and Harabor 2013), a state-of-the-art algorithm for finding shortest paths in road networks. The CPD serves as an oracle that forgoes the conventional state-space search, and instead extracts the shortest path using precomputed first move[1] data. However, unlike static road networks, where storing only one optimal first move for any OD pair is sufficient and can be simply computed via a Dijkstra search, extending the concept of CPD into a public transport network is not a trivial task. First, public transport networks are inherently time-dependent, with varying schedules and travel times. Therefore, the stored optimal path data must consider all potential departure times. This problem further intensifies when considering transfers between stops. Furthermore, while it is true that each stop is associated with a finite number of connections or trip departures, there may exist several stops within each station, making it challenging to efficiently compute and store the optimal path data between two stations. Finally, despite the fact that the *first move* concept can be extended to store the *first connection* on an optimal path, there may exist many connections along the same path. This complexity may affect the retrieval of the full path. To overcome these challenges, we present our solution, *Transfer Connection Database (TCD)*, an oracle-based approach that efficiently answers the EAT queries by storing the first transfer connection on the optimal paths for all OD station pairs. Our solution comprises two phases, offline preprocessing phase and online query phase.

[1]The first move data CPD$[o, d]$ reveals the identity of the first edge on the optimal path from any origin o to any destination d.

Transfer Connections

Recall that a journey from an origin station to a destination station consists of a set of connections. The number of connections can be large, especially if the two stations are far apart. For most oracle-based approaches in route planning (Abraham et al. 2011; Bast et al. 2010; Samet, Sankaranarayanan, and Alborzi 2008), extracting a complete solution often requires a step-by-step processing. As a result, the efficiency of the oracle is contingent on the representation of the solution, and enhancing this representation can significantly boost the performance (Shen et al. 2021). Fortunately, unlike road networks, where maintaining every edge along an optimal path is essential for successful navigation, public transport networks operate differently. In the latter, a journey connecting two stations depends on the trips scheduled in the timetable, where each trip must serve a specific sequence of stops. Consequently, the critical importance of a journey lies in preserving the information of transfers between different trips, since the connections within the same trip can be followed effortlessly. Based on this observation, we propose the concept of *transfer connections*.

Definition 1. *(Transfer Connection) Considering a journey $J = \langle c_0, c_1, \ldots, c_n \rangle$, a sub-sequence of connections $\langle c_j, c_{j+1}, \ldots, c_{j+k} \rangle \subseteq J$ forms a transfer connection tc, iff (i) these connections share the same trip (i.e., $t(c_j) = t(c_{j+1}) = \ldots = t(c_{j+k})$); and (ii) each of the preceding connection c_{j-1} and following connection c_{j+k+1}, if they exist, serves a different trip (i.e., $t(c_{j-1}) \neq t(c_j) \neq t(c_{j+k+1})$). To represent tc, we use the tuple $tc = (c_{dep}, c_{arr})$, where $c_{dep} = c_j$ and $c_{arr} = c_{j+k}$.*

As a result, a journey J can be represented as a set of transfer connections $J = \langle tc_0, tc_1, \ldots, tc_m \rangle$. This representation reduces the number of journey elements, enabling faster full journey reconstruction when answering queries.

Offline Preprocessing

During the offline phase, TCD utilises the concept of transfer connections to construct an oracle, called *first transfer table* (FT). This table stores labels encoding the first transfer connection on each optimal journey in the network.

First Transfer Table To support EAT queries, the oracle must have the ability to identify the first transfer connection for any given OD station pair, considering any possible departure time. However, precomputing such oracle is challenging due to the uncertainty associated with the query's departure time. Fortunately, in public transport networks, a key observation is that any optimal journey from an origin station s_o to any other destination station s_d must initiate from a connection departing either (i) directly from a stop $p_i \in p(s_o)$, or (ii) from a stop p_j that belongs to a neighbouring station in $n(s_o)$, accessible from s_o via a footpath.

Motivated by this, it suffices to consider only the departure times encoded by the accessible connections from s_o. A straightforward way to build the oracle involves maintaining an $|\mathcal{S}| \times |S|$ FT table, where the rows correspond to origin stations and the columns correspond to destination stations, and $|S|$ is the number of stations. However, such table is unlikely to be built efficiently and may contain many redundant labels across the rows. This is because, to store the labels for a row s_i, we must consider all accessible connections departing from stations $s_i \cup n(s_i)$, resulting in repeated labels computation and storage. To overcome this issue, we propose the idea of *neighbourhood stations* which leverages the clustering nature of public transport networks.

Definition 2. *(Neighbourhood Station) Given a timetable TB, a set of stations in S forms a neighbourhood station NS, iff all station pairs in NS are connected by footpaths in F (i.e., $\exists f(p_i, p_j, \delta\tau(p_i \rightarrow p_j)) \in F\ \forall p_i, p_j \in NS$).*

Lemma 1. *Given a timetable TB, stations in S can be decomposed into a set of independent neighbourhood stations $S = \{NS_0, NS_1, \ldots, NS_n\}$, such that there exist no footpaths between neighbourhood stations (i.e., $\forall p_i \in NS_i$ and $\forall p_j \in NS_j$, $\nexists f(p_i, p_j, \delta\tau(p_i \rightarrow p_j)) \in F$).*

Proof. (Sketch) Given two neighbourhood stations $NS_1 = \{s_i, s_j\}$ and $NS_2 = \{s_k, s_l\}$. Assume there exists a footpath from a stop in $p(s_i)$ to a stop in $p(s_k)$, then due to the transitivity closure property, there must exist footpaths connecting all stop pairs between s_i and s_k, and eventually between NS_1 and NS_2. As a result, NS_1 and NS_2 must be merged to form a new neighbourhood station. Thus, the lemma holds. □

Following Lemma 1, we decompose the stations in S into a set of neighbourhood stations $\mathcal{NS}$. Subsequently, we revise the oracle into an $|\mathcal{NS}| \times |S|$ FT table, where rows represent origin neighbourhood stations and columns represent destination stations, and $|\mathcal{NS}|$ is the number of neighbourhood stations. The revised FT table reduces both preprocessing time and space requirements. Within each cell $FT[NS_o][s_d]$, we store a list of labels $\{l_0, l_1, \ldots, l_n\}$, where each label l_i signifies the first transfer connection tc_i on an optimal journey from one of the stations in NS_o to station s_d. To ensure efficient query processing, labels in $FT[NS_o][s_d]$ are sorted by the arrival time (at s_d) of their corresponding journey. While it is possible, in the worst-case scenario, for all stations to be grouped into a single neighbourhood station, this is typically not the case in practice.

Example 1. *Figure 2 shows a toy network where stations are decomposed into neighbourhood stations based on the footpaths extending between their stops. Stations s_1 and s_2, as well as stations s_4 and s_5 are grouped into neighbourhood stations NS_1 and NS_3, respectively. Meanwhile, s_3 is isolated, forming its own neighbourhood station NS_2. The neighbourhood stations are independent having no footpaths between them. Simultaneously, all stop pairs within each neighbourhood station are connected via footpaths. Table 1 shows the FT table for this network, where rows represent origin neighbourhood stations and columns represent destination stations. Each cell $FT[NS_o][s_d]$ stores the list of the first transfer connections for all optimal journeys from NS_o to s_d. For instance, the label (c_1, c_2) in $FT[NS_3][s_2]$ is the first transfer connection of the journey $J = \langle (c_1, c_2), (c_5, c_5) \rangle$ from NS_3 to s_2.*

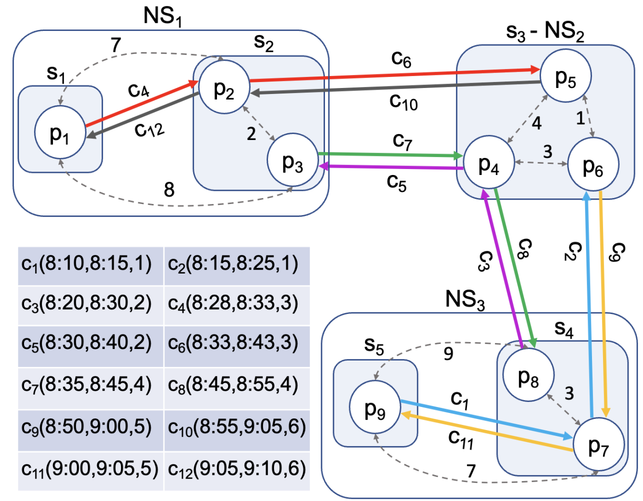

Figure 2: A toy network consisting of nine stops in five stations. The bidirectional arrows represent footpaths between stops with their walking time. Six trips are shown, each in a unique colour. Each coloured arrow represents a connection from p_{dep} to p_{arr} stops, in the form $c_i = (\tau_{dep}, \tau_{arr}, t)$

O\D	s_1	s_2	s_3	s_4	s_5
NS_1	(c_{12}, c_{12}) $\underline{(c_7, c_7)}$ $\underline{(c_6, c_6)}$	(c_4, c_4)	(c_6, c_6) (c_4, c_6) (c_7, c_7)	(c_7, c_8) (c_4, c_4) $\underline{(c_6, c_6)}$	(c_7, c_8) (c_4, c_4) $\underline{(c_6, c_6)}$
NS_2	(c_5, c_5) (c_{10}, c_{12})	(c_5, c_5) (c_{10}, c_{10})	-	(c_8, c_8) (c_9, c_9)	(c_8, c_8) (c_9, c_{11})
NS_3	(c_3, c_5) $\underline{(c_2, c_2)}$ $\underline{(c_1, c_2)}$	(c_3, c_5) $\underline{(c_2, c_2)}$ $\underline{(c_1, c_2)}$	(c_2, c_2) (c_1, c_2) (c_3, c_3)	(c_1, c_1)	(c_3, c_3) $\underline{(c_2, c_2)}$ (c_{11}, c_{11})

Table 1: The FT table for the toy network

Table Construction In order to compute the FT table, we adapt the CSA algorithm to compute the optimal journeys from each neighbourhood station in $\mathcal{NS}$, to each station in S. The pseudo-code of our algorithm is presented in Algorithm 1. To begin, the algorithm initialises an empty $|\mathcal{NS}| \times |S|$ FT table. Next, the algorithm iterates through each neighbourhood station $NS_o \in \mathcal{NS}$ to compute the row $FT[NS_o]$ in the FT table (line 1).

In each iteration, the algorithm examines each connection c_i departing from NS_o and runs a modified CSA starting from c_i (lines 2-3). The modified CSA functions similarly to before, with two key changes: (i) since the earliest arrival time at every stop $p_j \in P$ has to be computed, the scan propagates through the connections in C and can not be pruned before reaching a connection c_k with $\tau_{dep} \geq \tau_{EAmax}$, where τ_{EAmax} is the latest tentative arrival time at all stops in

Algorithm 1: Computing First Transfer Table

```
Input: TB: the timetable of public transport network.
Output: FT: the first transfer table of TB.
Initialisation: FT: an empty |NS| × |S| table.
1  for each station NS_o ∈ NS do
2      for each connection c_i ∈ NS_o do
3          R = modifiedCSA(p_dep(c_i),τ_dep(c_i));
4          for each station s_d ∈ S do
5              (c*_arr, τ*_EA) = getMinEarliestArrival(s_d, R);
6              tc_i(s_d) = (c_i, c*_arr);
7              if tc_i(s_d) ≠ ∅ then
8                  FT[NS_o][s_d] ← append(tc_i(s_d), τ_EA);
9      for each station s_d ∈ S do
10         sort FT[NS_o][s_d] in increasing order of τ_EA;
11         remove τ_EA of each label from FT[NS_o][s_d];
12 return FT;
```

P; and (ii) it returns an array of tuples R, where each tuple (c_{arr}, τ_{EA}) corresponds to the arrival connection of the first transfer connection and the earliest arrival time, respectively, for each stop $p_j \in P$. To determine c_{arr} for a stop p_j, the trip of the connection c_k that updates τ_{EA} at p_j is checked with the initial trip $t(c_i)$. If they are the same (i.e., $t(c_k) = t(c_i)$), then c_{arr} is set to c_k (no transfer required to get to p_j). Otherwise, it is set to c_{arr} of the departure stop $p_{dep}(c_k)$ (a transfer is required). The algorithm then processes the returned array R to identify the first transfer connection for each destination station $s_d \in S$. Specifically, the algorithm checks the tuples of all stops in s_d, and find the tuple (c^*_{arr}, τ^*_{EA}) with the minimum (earliest) τ_{EA} (lines 4-5). Next, the first transfer connection for s_d can be formed as $tc_i(s_d) = (c_i, c^*_{arr})$ (line 6). Note that the computed $tc_i(s_d)$ can be empty, indicating that the optimal journey to s_d is reached by a footpath. In this case, we avoid storing the empty tc_i and instead handle this situation during the query phase. For the first transfer connection $tc_i(s_d)$ that is not empty, the algorithm appends (tc_i, τ_{EA}) to the cell $FT[NS_o][s_d]$ in the FT table (lines 7-8).

Once the algorithm finishes examining all connections departing from NS_o, it proceeds to sort the labels (tc_i, τ_{EA}) of each cell $FT[NS_o][s_d]$ based on the increasing order of the earliest arrival time τ_{EA} and then removes τ_{EA} accordingly (lines 9-11). Finally, the algorithm continues the iterations to process the next neighbourhood station. When all neighbourhood stations in $\mathcal{NS}$ have been processed, the algorithm concludes and returns the first transfer table FT (line 12). Note that due to the independence of each row $FT[NS_o]$, the construction of the FT table can be efficiently parallelised, with a potential speedup proportional to the number of processors available in the machine.

Online Query Processing

With the precomputed FT table built in hand, we can extract optimal journeys efficiently. Given an EAT query $q = (s_o, s_d, \tau_q)$, Algorithm 2 is called to extract the first transfer connection $tc = (c_{dep}, c_{arr})$ from the neighbourhood station that s_o belongs to towards s_d, considering departure time τ_q. Once tc is extracted, the next extraction follows immediately using the arrival stop $p_{arr}(c_{arr})$ and arrival time $\tau_{arr}(c_{arr})$. The entire journey can thus be extracted as a simple recursion until an optimal transfer connection reaches s_d. Once the algorithm finishes, each transfer connection is recovered to form the complete journey.

Algorithm 2: Extracting First Transfer Connection

```
Input: TB: timetable of public transport network; FT: first
       transfer table of TB; P_cur: set of departure stops; τ_cur:
       departure time from any stop in P_cur; s_d: destination
       station.
Output: tc: the next optimal transfer connection to s_d.
Initialisation: tc_n = ∅.
1  NS = neighbourhood station of P_cur;
2  for each (c_dep, c_arr) ∈ FT[NS][s_d] in sequence do
3      if τ_cur ≤ τ_dep(c_dep) then
4          if p_dep(c_dep) ∈ P_cur then
5              tc_n = (c_dep, c_arr) and break;
6          else
7              f_tr = min footpath from P_cur to p_dep(c_dep);
8              if τ_curr + δτ(f_tr) ≤ τ_dep then
9                  tc_n = (c_dep, c_arr) and break;
10 if s_d is within the same NS then
11     f_tr = min footpath from P_cur to s_d;
12     if τ_cur + δτ(f_tr) ≤ τ_arr(c_arr(tc_n)) then
13         return s_d;
14 return tc_n;
```

First Transfer Connection Extraction To extract a first transfer connection from the FT table, the algorithm needs to consider the departure location as (i) a station when departing from s_o, or (ii) a stop when departing from the arrival stop of the previous transfer connection. To maintain the generality of the algorithm, the input is taken as a set of departure stops P_{cur}, within a station. The pseudo-code of our algorithm is shown in Algorithm 2. To begin, the algorithm initialises the next transfer connection tc_n as empty and finds the neighbourhood station NS that contains the departure stops P_{cur} (line 1). Next, the algorithm scans the (presorted) transfer connections of $FT[NS][s_d]$ in increasing order of their arrival time at s_d to find the first reachable transfer connection (line 2). For each transfer connection (c_{dep}, c_{arr}), the algorithm first checks whether its departure time is no earlier than the current time τ_{cur} (line 3). If not, the algorithm simply skips it. Otherwise, the transfer connection can be potentially reached on time, and the algorithm further checks if the departure stop $p_{dep}(c_{dep})$ belongs to P_{cur} (line 4). If this is the case, the transfer connection can always be reached by waiting, and the algorithm sets tc_n accordingly, then breaks (line 5). Otherwise, the algorithm finds the shortest footpath f_{tr} from the stops in P_{cur} to the departure stop $p_{dep}(c_{dep})$, and checks whether the transfer connection can be reached on time via f_{tr} (lines 6-8). If possible, the algorithm sets tc_n to the transfer connection and breaks (line 9), otherwise the transfer connection is skipped.

Recall that the FT table does not store any transfer connection when the destination is optimally reached via a footpath. Therefore, before returning tc_n, the algorithm has to check whether there exists a footpath from P_{cur} arriving at

s_d before the transfer connection tc_n. Particularly, the algorithm first checks whether s_d is within the same neighbourhood station NS (line 10). If so, the shortest footpath from P_{cur} to s_d is found and checked whether it can lead to reaching s_d earlier compared to tc_n (lines 11-12). In such case, the algorithm returns a dummy transfer connection s_d, indicating that the next transfer connection is to directly reach s_d by a footpath (line 13). Alternatively, the transfer connection tc_n extracted from FT is returned (line 14).

Theorem 1. *Given an EAT query $q = (s_o, s_d, \tau_q)$, the optimal journey can be obtained by recursively extracting the first transfer connections using Algorithm 2.*

Proof. (Sketch) The correctness of the algorithm is ensured by the way the FT table is constructed. According to Lemma 1, all connections are considered as potential departures. In addition, Algorithm 2 clearly considers all possible scenarios to reach s_d as early as possible, whether by public transport connections or walking footpaths. □

Example 2. *Given a query $q = (s_2, s_5, 8{:}30)$ in our network (Figure 2), the precomputed FT table (Table 1) is used to find the optimal journey J solving q. Initially, the algorithm needs to extract the first reachable transfer connection from the neighbourhood to which s_2 belongs, NS_1, towards s_5 by evaluating labels in $FT[NS_1][s_5]$. The first transfer connection that has departure time $\tau_{dep} \geq 8{:}30$ is (c_7, c_8). Since c_7 departs from the origin station s_2, it can always be reached by waiting. The arrival connection c_8 reaches stop p_8 at $8{:}55$. As no footpaths exist from s_2 to s_5, this transfer connection is safely added to $J = \langle (c_7, c_8), \ldots \rangle$. For the next extraction, the algorithm checks $FT[NS_3][s_5]$ as p_8 belongs to NS_3. Following the same procedure, the first reachable transfer connection with $\tau_{dep} \geq 8{:}55$ is (c_{11}, c_{11}). As c_{11} does not depart from the current stop p_8, the algorithm checks if $p_{dep}(c_{11})$ can be reached on time. Since the transfer time from p_8 to p_7 is respected ($8{:}55 + 0{:}03 \leq 9{:}00$), c_{11} can be accessed, arriving at stop p_9 at $9{:}05$. However, a footpath from p_8 in s_4 to p_9 in s_5 exists, resulting in an arrival time of $8{:}55 + 0{:}09 = 9{:}04$, earlier than $9{:}05$ achieved by the transfer connection. Thus, a dummy transfer connection is appended to $J = \langle (c_7, c_8), s_5 \rangle$. Finally, as s_5 is reached, the algorithm terminates, and J is recovered and returned.*

Optimisations

We introduce optimisation techniques to improve the efficiency of query processing. These techniques focus on reducing the number and size of labels stored in the FT table without affecting the optimality of the solutions.

Dominance Check Recall that in each cell $FT[NS_o][s_d]$, the optimal first transfer connection from each departure event from NS_o to s_d is recorded. While each first transfer connection represents an optimal journey with respect to its departure connection, there may still be cases where such a journey is dominated by the optimal journey of another departure connection. This implies that departing from the dominated connection always results in a later arrival. To address this issue, we explore the following lemma to identify and remove the dominated transfer connections.

Lemma 2. *Let (tc_i, τ_{EA_i}) and (tc_j, τ_{EA_j}) be two transfer connections that depart from NS_o with their earliest arrival time at s_d. The transfer connection (tc_i, τ_{EA_i}) dominates (tc_j, τ_{EA_j}) iff (i) tc_i departs no earlier than tc_j, allowing a sufficient time for a feasible transfer from tc_j to tc_i (i.e., $\tau_{dep}(tc_i) \geq \tau_{dep}(tc_j) + \delta\tau(p_{dep}(tc_j) \rightarrow p_{dep}(tc_i))$); and (ii) tc_i arrives at s_d no later than tc_j, (i.e., $\tau_{EA_i} \leq \tau_{EA_j}$).*

Due to the triangle inequality, Lemma 2 holds. We omit the details of the proof. To eliminate the dominated transfer connections within each cell $FT[NS_o][s_d]$, we conduct dominance checks during the preprocessing phase (Algorithm 1). Recall that to execute CSA, connections in C must be sorted in increasing order of their departure times. However, for a transfer connection tc_i to dominate another transfer connection tc_j, its departure time must be later than that of tc_j (Lemma 2). Therefore, during construction of each row $FT[NS_o]$, we modified the algorithm to access connections in NS_o in reverse order of C (Algorithm 1 - line 2). Subsequently, for each optimal transfer connection tc_j computed for station s_d, a dominance check is performed with each transfer connection tc_i already in $FT[NS_o][s_d]$. Only if tc_j is not dominated by any of these connections, it is appended to $FT[NS_o][s_d]$ (Algorithm 1 - lines 7-8).

Example 3. *Considering our toy network, the dominated transfer connections are underlined in the FT table (Table 1). Such transfer connections can be safely removed from the FT table. Considering the cell $FT[NS_1][s_4]$, the third transfer connection $tc_3 = (c_6, c_6)$ with τ_{EA} of $9{:}00$ is dominated by the first transfer connection $tc_1 = (c_7, c_8)$ with τ_{EA} of $8{:}55$. This is because tc_1 arrives earlier at s_4 ($8{:}55$ vs. $9{:}00$), departs later ($8{:}35$ vs. $8{:}33$), and the difference between their departure times is no greater than the walking time between their departure stops (2min vs. 2min).*

Transfer Connection Compression In public transport networks, the optimal journey from a neighbourhood station NS_o to a destination station s_d often involves a first transfer (if required) at the same stop, within the same time period of the day. Based on this observation, we introduce two compression techniques to reduce size and number of labels stored in each cell $FT[NS_o][s_d]$. Specifically, we update the representation of transfer connections from (c_{dep}, c_{arr}) to (c_{dep}, p_{arr}), where p_{arr} is the arrival stop of c_{arr} (i.e., $p_{arr}(c_{arr})$). Consequently, the size of each transfer connection can be reduced due to the limited number of stops in a timetable TB. Each p_{arr} can be efficiently stored using a two-byte integer, while maintaining c_{arr} would typically require four bytes due to the considerably larger number of connections in TB. With this updated representation, we can effectively merge consecutive transfer connections $(c_0, p_{arr}), (c_1, p_{arr}), \ldots, (c_n, p_{arr})$ sharing the same arrival stop, into a single label $((c_0, c_1, \ldots, c_n), p_{arr})$, to eliminate the redundant storage of multiple p_{arr} values. While simple, we later showcase the effectiveness of these techniques.

Experiments

We evaluate our algorithm, Transfer Connection Database (TCD), against our implementation of Connection Scan

Algorithm (CSA) (Dibbelt et al. 2018). Additionally, we compare the performance with another state-of-the-art algorithm, Round-bAsed Public Transit Optimized Router (RAPTOR) (Delling, Pajor, and Werneck 2015). To the best of our knowledge, CSA and RAPTOR represent the state-of-the-art algorithms for solving the earliest arrival time problem. While more recent works in the general area exist, including (Potthoff and Sauer 2022; Lehoux-Lebacque and Loiodice 2021; Delling, Dibbelt, and Pajor 2019; Phan and Viennot 2019), they all have different focuses, such as considering multiple modalities or dynamic changes. According to the results of these works, such extensions and enhancements come at the cost of reduced performance compared to our benchmarks. Therefore, we focus on CSA and RAPTOR for our static earliest arrival time problem. Similar to CSA, RAPTOR is an online algorithm that directly operates on the timetable, rather than a graph. It inherently optimises both earliest arrival time and number of transfers. RAPTOR is centred around modelling routes, which are then traversed in rounds to efficiently update arrival times at stops and solve queries. The implementation of RAPTOR is taken from a public repository[2]. For the reproducibility, the implementation of our algorithm is available online[3]. All experiments were implemented in C++17 with full optimisation on a 3.20 GHz Apple M1 machine with 16 GB of RAM, running macOS Ventura 13.4, and utilizing a single thread.

Datasets Three large metropolitan networks, namely Berlin, Paris, and London, are considered with all available public transport modes, including trains, subways, trams, buses, and ferries. The first dataset is based on open data made available by the service provider[4]. The other two datasets are sourced from (Phan and Viennot 2019). Each network is based on a weekday timetable. Table 2 summarises the main attributes for each dataset. To model the transfers, a conservative walking speed of 1m/s is used to compute the required walking time. Initially, a footpath is constructed between every two stops within each station considering exact transfer times. In addition, we include footpaths between the stops of a station pair if the two stations fall within a predefined radius r from each other. Finally, additional footpaths are introduced to make the transfer graph transitively closed. A wide range of values can be used for r. Setting $r = 0$ potentially results in unsolvable queries since walking between stations is disallowed. On the other hand, choosing a large value (e.g., $r = 800$m) leads to the extensive addition of footpaths connecting far-apart stations, which is unlikely to significantly improve travel time savings. For our experiments, we set r to 250m.

Query Generation We generated a sample of 5,000 station pairs chosen uniformly at random for each dataset. Eight fixed departure times evenly spaced across the day (from 3 am to midnight) were assigned for these station pairs.

Experiment 1: Transfer Model Impact To evaluate the impact of different transfer models, we compare our exact transfer model with the inter-station transfer model widely used in existing literature. This model employs uniform transfer times both within and between stations. Two variants are derived from this model, one utilising the maximum transfer times within and between stations from our exact transfer model (denoted as *max model*), and the other utilising the minimum transfer times (denoted as *min model*). For the max model, users may receive a feasible yet suboptimal journey leading to delays. Conversely, the min model may return an infeasible journey due to violation of actual transfer times. In this case, we assume that users would attempt to replan their journey at each infeasible transfer, often leading to cumulative delays. Figure 3 illustrates the median delays associated with the impacted queries in both the min and max models, compared to our extract transfer model. The x-axis represents the various departure times throughout a day. From the figure, it becomes evident that both the min and max models lead to considerable delays. The delays in the min model are notably more pronounced across all times, due to the cumulative transfer effects discussed earlier. Interestingly, we also note that the delays during peak periods (i.e., 6 am - 6 pm) align consistently for both min and max model, with a prevalent delay of 5 minutes in the max model and 10-15 minutes in the min model. In contrast, during off-peak periods (i.e, 3 am, 9 pm, and 12 am), a notable increase is observed in the min model, with delays reaching 20-30 minutes in Berlin and Paris, and to a lesser extent in London. Lastly, the values shown on each line indicates the percentages of impacted queries. It is evident that, in the absence of implementing an exact transfer model, a substantial portion of queries could encounter delays, averaging around 45% in Berlin and Paris and 26% in London.

Dataset	Stations	Stops	Conns.	Trips	Footpaths	N'hoods.
Berlin	3,365	8,359	1,006	42,518	45,553	2,598
Paris	6,263	12,047	1,836	78,757	148,844	3,055
London	9,798	14,516	3,089	87,898	162,543	4,414

Table 2: Main attributes of the test datasets, where Conns. is connections (x10^3) and N'hoods. is neighbourhoods

Experiment 2: Preprocessing Cost A summary of the preprocessing phase undertaken for each instance is presented in Table 3. This includes the oracle size in gigabytes, preprocessing time in minutes, compression percentage, and number of labels per OD pair, considering different combinations of the optimisations Transfer Connection Compression (TCC) and Dominance check (Dom.). The latter significantly reduces oracle size by 68-76%. When combined with the TCC technique, an overall compression of 79-85% is achieved across all instances, with a growing rate for larger instances. Notably, these optimisations incur negligible time overhead. The number of labels per OD pair is considerably decreased in all instances, allowing for faster query times. Note that the offline phase can be easily parallelised, leading to a substantial reduction in preprocessing time.

Experiment 3: Runtime Comparison In Figure 4, we compare the average query processing time for our algorithm, TCD, with that of the competitors, CSA and RAP-

[2]https://github.com/ducminh-phan/RAPTOR

[3]https://github.com/abdallah-abuaisha/TCD

[4]https://www.vbb.de/vbb-services/api-open-data/datensaetze

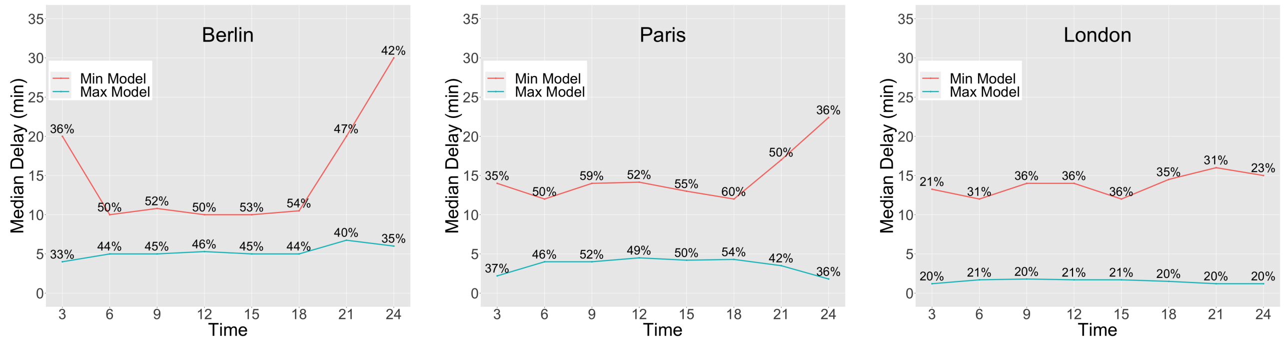

Figure 3: Delays of inter-station models compared to exact model for affected queries (percentage shown) across the day

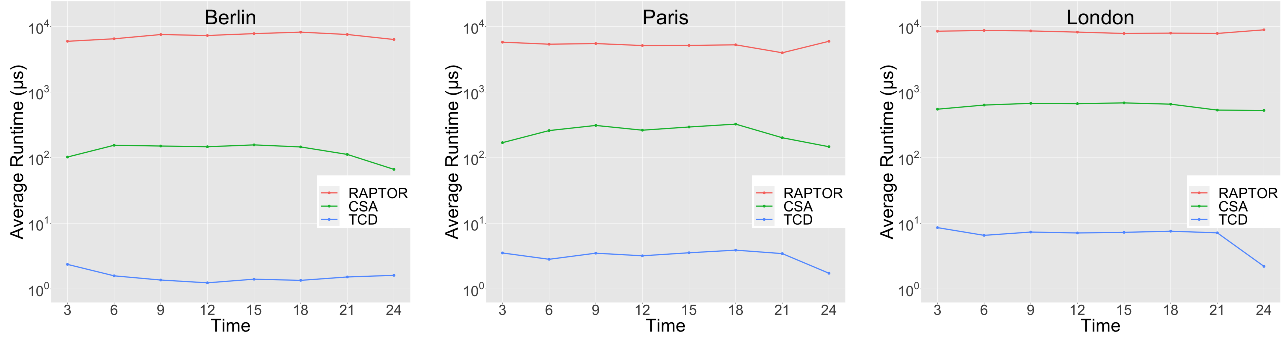

Figure 4: Query performance comparison throughout the day

Dataset	TCC	Dom.	Size	%Comp.	Time	# OD labels
Berlin	-	-	25.4	-	8	362
Berlin	✓	-	18.5	27%	8	332
Berlin	-	✓	8.1	68%	8	116
Berlin	✓	✓	5.2	79%	8	68
Paris	-	-	88.0	-	32	575
Paris	✓	-	64.4	27%	32	533
Paris	-	✓	24.4	72%	33	160
Paris	✓	✓	16.0	82%	33	99
London	-	-	223.8	-	111	647
London	✓	-	159.2	29%	111	547
London	-	✓	53.5	76%	113	155
London	✓	✓	34.2	85%	113	86

Table 3: Preprocessing figures for different optimisation combinations, with Size in gigabytes and Time in minutes

TOR. The results demonstrate that TCD consistently outperforms both algorithms throughout all times of the day, achieving a speedup of two to three orders of magnitude. The average runtime for TCD is 1.5, 3.2, and 6.7 microseconds in the instances of Berlin, Paris, and London, respectively. A key factor contributing to the superior performance of our algorithm is the utilisation of transfer connections. This strategic implementation results in our TCD algorithm requiring extraction of only a notably limited number of labels from the oracle. In the cases of Berlin, Paris, and London, the average count of extracted transfer connections per query stands at 3.7, 4.1, and 6.3, respectively.

Discussion In this paper, we focus on addressing the journey planning problem at an urban and metropolitan network scale. Concerning preprocessing costs, we demonstrate that our proposed database-driven approach, TCD, can efficiently construct within a reasonable timeframe and comfortably fits into memory. While we acknowledge that preprocessing time and space demands may increase quadratically for significantly larger networks with tens or hundreds of millions of connections, such scales fall outside the scope of our metropolitan focus. Regarding query performance, we showcase that TCD outperforms two competing algorithms, CSA and RAPTOR. There are other state-of-the-art algorithms in the area, such as TB (Witt 2015) and TP (Bast et al. 2010). However, TCD is a single-criteria algorithm optimising earliest arrival time, whereas TP and TB are multicriteria algorithms optimising both earliest arrival time and minimum number of transfers. In addition, according to their experimental analysis, TB and TP are only several factors faster than RAPTOR. In our results, we show that TCD is around three orders of magnitude faster than RAPTOR.

Conclusion and Future Work

We have presented an efficient solution to the earliest arrival time problem, integrating exact transfer costs and leveraging a well-structured transfer connection database. The significance of employing exact transfer models has been demonstrated. Our future work involves extending TCD to address additional aspects, such as the multicriteria problem, and enhancing the compression of the database even further.

Acknowledgments

This work is partially funded by The Australian Research Council (ARC) under grant DP190100013.

References

Abraham, I.; Delling, D.; Goldberg, A. V.; and Werneck, R. F. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*, volume 6630 of *Lecture Notes in Computer Science*, 230–241. Springer.

Bast, H.; Carlsson, E.; Eigenwillig, A.; Geisberger, R.; Harrelson, C.; Raychev, V.; and Viger, F. 2010. Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns. *ESA (1)*, 6346: 290–301.

Botea, A. 2011. Ultra-fast optimal pathfinding without runtime search. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 7, 122–127.

Botea, A.; and Harabor, D. 2013. Path planning with compressed all-pairs shortest paths data. In *Twenty-Third International Conference on Automated Planning and Scheduling*.

Delling, D.; Dibbelt, J.; and Pajor, T. 2019. Fast and exact public transit routing with restricted pareto sets. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, 54–65. SIAM.

Delling, D.; Dibbelt, J.; Pajor, T.; Wagner, D.; and Werneck, R. F. 2013. Computing multimodal journeys in practice. In *International Symposium on Experimental Algorithms*, 260–271. Springer.

Delling, D.; Dibbelt, J.; Pajor, T.; and Werneck, R. F. 2015. Public transit labeling. In *Experimental Algorithms: 14th International Symposium, SEA 2015, Paris, France, June 29–July 1, 2015, Proceedings 14*, 273–285. Springer.

Delling, D.; Pajor, T.; and Werneck, R. F. 2015. Round-based public transit routing. *Transportation Science*, 49(3): 591–604.

Dibbelt, J.; Pajor, T.; Strasser, B.; and Wagner, D. 2018. Connection scan algorithm. *Journal of Experimental Algorithmics (JEA)*, 23: 1–56.

Geisberger, R. 2010. Contraction of timetable networks with realistic transfers. In *Experimental Algorithms: 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings 9*, 71–82. Springer.

Giannakopoulou, K.; Paraskevopoulos, A.; and Zaroliagis, C. 2019. Multimodal dynamic journey-planning. *Algorithms*, 12(10): 213.

Lehoux-Lebacque, V.; and Loiodice, C. 2021. Transfer Customization with the Trip-Based Public Transit Routing Algorithm. In *21st Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Phan, D.-M.; and Viennot, L. 2019. Fast public transit routing with unrestricted walking through hub labeling. In *International Symposium on Experimental Algorithms*, 237–247. Springer.

Potthoff, M.; and Sauer, J. 2022. Efficient Algorithms for Fully Multimodal Journey Planning. In *22nd Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Samet, H.; Sankaranarayanan, J.; and Alborzi, H. 2008. Scalable network distance browsing in spatial databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, 43–54. ACM.

Shen, B.; Cheema, M. A.; Harabor, D. D.; and Stuckey, P. J. 2021. Contracting and compressing shortest path databases. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, 322–330.

Witt, S. 2015. Trip-based public transit routing. In *Algorithms-ESA 2015: 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, 1025–1036. Springer.