Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Optimal Routing In Public Transportation Networks Using PHAST

David Bruns

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Funke
Supervisor:	Dr. rer. nat. Claudius Proissl
Commenced:	July 11, 2023
Completed:	January 11, 2024

Abstract

Finding optimal routes in public transportation networks is a challenging and complex algorithmic task. Compared to routing in road networks, public transportation routing algorithms also have to take factors like schedules and transfer times into consideration. Because of this, shortest-path-algorithms that are used in road networks have to be modified in order to be useable in public transportation networks.
One method that is used to calculate shortest paths in road maps is the PHAST Hardware-Accelerated Shortest path Trees (PHAST) approach. It makes use of hierarchical structures in transport networks to allow a fast and efficient computation. As shown in [SZ21], it can also be used as a heuristic for the A*-algorithm [HNR68].
Given that PHAST has been observed to have a good performance in road networks and A* being an algorithm that provides great flexibility, the question rises whether a combination of both could also be used in public transportation networks.
This thesis presents the new algorithm PUBPHAST which combines the A*-Search Algorithm with the PHAST heuristic in a public transportation network context. It is experimentally explored whether A* in combination with PHAST also has similar performance advantages in public transportation networks as in road networks. To this end, openly available public transportation data in format of the General Transit Feed Specification (GTFS) [Goo05] is transformed into a fitting graph structure which serves as foundation for the algorithm. PUBPHAST is then compared to public transportation adjusted implementations of Dijkstra's algorithm [Dij22] and the Connection Scan Algorithm (CSA) [DPSW17].
It became clear that PUBPHAST is both faster than the CSA and Dijkstra's algorithm, suggesting that hierarchical routing has performance advantages in public transportation networks.

Kurzfassung

Das Finden von optimalen Pfaden im öffentlichen Personenverkehr ist ein komplexer Themenbereich und aus algorithmischer Sicht eine herausfordernde Aufgabe. Im Gegensatz zur Routenplanung im Straßenverkehr müssen zum Beispiel auch Fahrpläne und Umstiegszeiten in Betracht gezogen werden. Deswegen können herkömmliche Distanzberechnungsverfahren für den Straßenverkehr nicht ohne weitere Modifizierung auch für den öffentlichen Personenverkehr verwendet werden.
Eine Methode, die für die Distanzberechnung im Straßenverkehr entwickelt wurde, nennt sich PHAST [DGNW11]. Dieses Verfahren nutzt hierarchische Strukturen im Transportnetz aus, um eine effiziente Distanzberechnung zu gewährleisten. In [SZ21] wurde PHAST dazu verwendet, die Heuristikfunktion im A*-Algorithmus zu berechnen. Dieses Verfahren verknüpft die Flexibilät des A*-Algorithmus mit der Schnelligkeit des PHAST-Ansatzes.
Ziel dieser Arbeit ist es daher, diese Kombination im Kontext von öffentlichen Personenverkehrnetzen zu implementieren.
Es wird der neue Algorithmus PUBPHAST vorgestellt, der die beiden Ansätze A* und PHAST vereint. PUBPHAST wurde mittels GTFS-Daten getestet und anschließend mit anderen Routenplanungsverfahren für den öffentlichen Personenverkehr verglichen.
Es hat sich herausgestellt, dass PUBPHAST sowohl im Vergleich zum Connection Scan Algorithmus (CSA) als auch genenüber Dijsktra's Algorithmus Geschwindigkeitsvorteile bietet.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Acronyms

CH Construction Hierarchies. 20

CSA Connection Scan Algorithm. 3

GTFS General Transit Feed Specification. 3

PHAST PHAST Hardware-Accelerated Shortest path Trees. 3

1 Introduction

1.1 Motivation

PHAST [DGNW11] is a road network routing algorithm that is also used as an A* heuristic. It takes advantage of hierarchical network structures to be efficient and provide a good performance. However, so far it is only tested in road networks. Due to public transportation networks being different in nature, it is not known whether PHAST would perform well in them.
On one hand there are fundamental differences between public transportation and road networks, the main one being the introduction of the time element resulting from having schedules for buses, trains, and so on. However, another critical difference could potentially be the structure and layout of the network.
Road networks inherently have few network nodes that connect to many other network nodes. This is due to the way streets are built and structured. Most longer routes will lead through highways. This results in a small number of nodes being present in many routes. PHAST exploits this hierarchical structure to achieve its performance advantages.
But hierarchical structures could potentially also be found in public transportation networks. Looking at transit routes between two cities for example, a lot of routes would use the same train to get to their destination. This potential network characteristic would make advantages of PHAST applicable to public transportation networks as well.
Therefore, the goal of this work is to implement PHAST in a public transportation context and to experimentally evaluate whether PHAST has advantages compared to other routing algorithms in public transportation networks.
This thesis aims to implement the new algorithm PUBPHAST, which is an extension of the combination of A* and PHAST that makes it useable in public transportation networks. To this end, public transportation data in GTFS format is first transformed to a graph structure that can be used as an input for PUBPHAST. Using this data as underlying foundation, PUBPHAST is then tested and compared to Dijkstra's Algorithm and the CSA to measure its performance. The goal of this work is to give an indication whether the hierarchical routing approach of PHAST has advantages or disadvantages in public transportation networks.

1.2 Outline of this Thesis

Chapter 2 sets the necessary theoretical foundations of this work, introducing Connection Scan, Dijkstra and A* Algorithms. It also explains the functionality of Contracted Hierarchies and PHAST.
The implementation of PUBPHAST and the extraction of its input graphs from the raw public transportation GTFS data is shown in Chapter 3.
The test results of this implementation and of the other public transportation network routing

algorithms are presented in Chapter 4. It also explains on the test method and draws a comparison between the algorithms shown in Chapter 2.
Lastly, Chapter 5 draws a conclusion of the results and gives an outlook.

1.3 Related Work

1.3.1 Shortest Paths

While implementing PHAST in the context of public transportation is new, there has been done plenty of research in the field of public transportation or road network routing.
Given an arbitrary graph $G = (V, E)$ with V being a set of nodes and E being a set of edges, there are different techniques to find the shortest distance between a source s and a target t.
Basic techniques include Dijkstra's algorithm [Dij22], the Bellman-Ford algorithm [Bel58] or the Floyd-Warshall algorithm [Flo62]. These algorithms do not require preprocessing.
Goal-directed techniques try to search for the target more purposefully. Compared to Dijkstra's algorithm for instance, these techniques do not visit the nodes that are in the opposite direction of the target. Examples for such methods are the A*-algorithm [HNR68], Geometric Containers [WWZ05] or Arc Flags [HKMS09].
On planar graphs, the computation expense and the size of graph separators is small [LT77]. In these cases separator-based techniques that separate nodes or edges have its merits.
Contraction Hierarchies [GSSD08], a technique which PHAST [DGNW11] is based on, belongs to the Hierarchical Techniques. These techniques use the hierarchical structure of road networks.
Another category of techniques are the Bounded-Hop techniques, like for example labeling algorithms [Pel00] such as Hub Labeling [Wel14]. They precompute distances between pairs of nodes to create a new graph which has additional shortcut edges.
The above mentioned techniques can all be combined with each other or extended to serve different functionalities. [BDG+16]

1.3.2 Public Transportation Networks

Public transit data usually comes in the form of a timetable which contains stops, routes and trips. To each trip belongs a vehicle (i.e. bus, train, ferry) which traverses the stops of a route at a certain time of day. The above mentioned shortest path algorithms cannot be used in this timetable format which is why it is natural to try to build a graph $G = (V, E)$ corresponding to a timetable of public transit information.
One way to build such a graph is letting each node be correspondent to a departure or arrival event of the timetable. The edges are then used to connect the sequential events. These kinds of graphs are called Time-Expanded Models [PSWZ08][BDG+16].
On the other hand, Time-Dependent Models [PSWZ08][BDG+16] are created by assigning each node to a stop. Edges between two nodes are added when the two correspondent stops have at least one connection in the timetable. The cost of the edges is determined by a travel time function, which also takes the time of departure and the resulting waiting period into account.

2 Theoretical Foundations

2.1 Dijkstra's Algorithm

Dijkstra's algorithm [Dij22] is a one-to-all search algorithm, meaning it computes the distance of one source node to all other nodes. It is a greedy algorithm and requires no preprocessing.
It uses two different data containers. The priority queue Q contains elements that are ordered by their distance to the source s (from small to big). The second container dist(n) is an array that keeps track of the current shortest distances from s to each node n.
Initially, Q contains only s and every distance is set to infinity, except dist(s) = 0. The algorithm then iteratively extracts the current top node u from Q and looks at all edges $e = (u, v) \in E$. For each edge it computes the distance to v by adding the current shortest distance to u and the length $l(u, v)$ of the edge. If this new distance to v is smaller than the current shortest distance dist(v), the value in the distances array is updated and the node v is pushed to the priority queue.
The running time of Dijkstra's algorithm using a standard binary heap is $\mathcal{O}((|V| + |E|)\log|V|)$ [BDG+16].

Algorithm 2.1 Dijkstra's Algorithm

Input: source s, target t with $s, t \in V$
Let Q be a priority queue; Q.push(s, 0)
Let dist be an array; dist(n) = infinity for $n \in V \setminus \{s\}$, dist(s) = 0
while Q is not empty **do**
    $u \leftarrow Q$.pop()
    for all edges $e = (u, v) \in E$ **do**
        if dist(u) + $l(u, v)$ < dist(v) **then**
            dist(v) = dist(u) + $l(u, v)$
            Q.push(v, dist(v))
        end if
    end for
end while
Output: dist(t)

A variation to this algorithm is the Bidirectional Dijkstra. This extension computes a one-to-one search from a source node s to a target node t. In this form, the Dijkstra computation is not only started from the source but also from the target. It terminates once the distances in the priority queue are higher than the current shortest distance between source and target.

2.2 A* Search Algorithm

The A* Search Algorithm [HNR68] is a goal-directed method. In its functionality it is very similar to Dijkstra's algorithm, with the crucial difference being the value that determines the position of an element in the priority queue Q.
It uses a potential function $h(n) : V \to \mathbb{R}$ which is a lower bound estimate of the distance of an arbitrary node n to the target t. This estimate is dependent of the context of the search. In case of a road network, such an estimate might be the airline distance between n and t.
Instead of just ordering the elements of Q by their distance from the source node s, the A* Search Algorithm computes a evaluation function $f(n) = g(n) + h(n)$. The value $g(n)$ of a node n is the current minimal distance from s to v and the value $h(n)$ is the estimated lower bound distance from n to t. So ultimately, the elements of Q are sorted by the estimated distance between s and t that result from the path the algorithm has taken to get to those elements.

Algorithm 2.2 A* Search Algorithm

1: **Input:** source s, target t, heuristic $h(n)$ with $s, t, n \in V$
2: Let Q be a priority queue; Q.push(s, 0)
3: Let dist be an array; dist(n) = infinity for $n \in V \setminus \{s\}$, dist(s) = 0
4: **while** Q is not empty **do**
5: $u \leftarrow Q$.pop()
6: **for** all edges $e = (u, v) \in E$ **do**
7: **if** dist(u) + $l(u, v)$ < dist(v) **then**
8: dist(v) = dist(u) + $l(u, v)$
9: f = dist(v) + $h(v)$
10: Q.push(v, f)
11: **end if**
12: **end for**
13: **end while**
14: **Output:** dist(t)

2.3 Contraction Hierarchies

Construction Hierarchies (CH) [GSSD08] is a hierarchical route planning technique that uses node contraction, which is the method of replacing the shortest paths going through a node with new shortcut edges. It also orders nodes by their importance. Both node contraction and node ordering are done in a preprocessing phase and are later used for routing during the query phase.
In more detail, a node n is contracted by removing it from the original graph while still preserving the shortest paths in the remaining graph. This is done by replacing paths of the form $\langle u, v, w \rangle$ by a shortcut edge $\langle u, w \rangle$. However, it is important to point out that this shortcut edge is only needed if $\langle u, v, w \rangle$ is the only shortest path from u to w.
The order in which nodes are contracted is determined by their importance (the less important nodes are first in the sequence) which is derived by a linear combination of several terms. Arguably the most important term is the *edge difference* which is computed by subtracting the incident edges from a node n of the number of shortcuts introduced when contracting n. The other terms used in

the linear combination are the *uniformity*, the *cost of contraction*, the *cost of queries* and *global measures*. Their functionality can be found in [GSSD08].
After preprocessing the shortcuts and the ordering of the nodes, one can conduct a query by starting a bidirectional shortest-path search. Doing this, the forward search uses only edges leading to more important nodes while that backward search uses only edges leading to less important ones.
This leads to a correct result because the shortest path must be a *up-down* path [GSSD08]. If there exists a shortest path from source node s to target node t, there must be a middle node m with $s \rightarrow m$ being a path only consisting of *up-edges* and $m \rightarrow t$ being a path only consisting of *down-edges*.

2.4 PHAST

PHAST (for PHAST Hardware-Accelerated Shortest path Trees) [DGNW11] is an algorithm that solves the non-negative single-source shortest path problem (NSSP). It computes shortest paths from one source s to all other nodes. It works well on graphs with low *highway dimension* (like road networks), which intuitively are graphs in which a small number of nodes is sufficient to hit all long shortest paths. It is based on the previously presented Contraction Hierarchies.
The preprocessing phase of PHAST is the same as the preprocessing of CH, resulting in set of shortcuts and an ordering of nodes.
In the query phase, PHAST initially sets all distances $d(n) = \infty$ (for all $n \neq s$) and sets $d(s) = 0$. The NSSP search is then done in two subphases. First, it executes a forward CH search by running Dijkstra's algorithm from s in the upward graph $G^{\uparrow} = (N, E^{\uparrow})$. This upward graph $G^{\uparrow}$ entails all edges $(v, w) \in E$ from the original graph G and the additional shortcuts, which fulfill the condition $\text{rank}(v) < \text{rank}(w)$ ($\text{rank}(n)$ being the rank of node n in the preprocessed CH node ordering).
In the second subphase, the algorithm iterates through all nodes in $G^{\downarrow} = (N, E^{\downarrow})$ ($E^{\downarrow} = \{(u, v) \in E : \text{rank}(u) > \text{rank}(v)\}$) in descending CH rank order and scans them. Scanning is done by looking at each incoming edge $(u, v) \in E^{\downarrow}$ and setting $d(v) = d(u) + l(u, v)$ if $d(u) + l(u, v) < d(v)$.

2.5 PHAST as an A* Heuristic

PHAST can also be applied as a heuristic for the A* Search Algorithm [SZ21]. The idea behind this is that one can apply PHAST on a lower bounds graph to swiftly compute heuristic estimations from all nodes to the target.
In the preprocessing phase of this A* variation, the algorithm gets an input lower bounds graph G_l and computes the CH shortcuts and node ordering, outputting G_l^+ and the node ranks.
After receiving a s to t shortest path query, one first executes a backward CH search from the target t on graph G_l^+, as can be seen in algorithm 2.3.

Algorithm 2.3 CH Backward Search

Input: graph G_l^+
Let $B[x]$ be an array that stores the tentative distance from x to target t
$B[x] \leftarrow \infty$ for all $x \neq t$, $B[t] \leftarrow 0$
Let Q be a priority queue; Q.push$(t, 0)$
while Q is not empty **do**
    $y \leftarrow Q$.pop()
    for (x, y) is down-edge in G_l^+ **do**
        if $B[x] > l(x, y) + B[<]$ **then**
            $B[x] \leftarrow l(x, y) + B[<]$
            Q.push$(x, B[x])$
        end if
    end for
end while
Output: $B[x]$

After completing the backward search, the algorithm now computes the all-to-one PHAST search and stores each lower bound distance in an array. This array is then used as heuristic by simply looking up the lower bound distance of the respective node.
The all-to-one PHAST search (shown in algorithm 2.4) iterates over the nodes of the different CH ordering levels (from most to least important). It looks at all outgoing up edges from each node and updates the value if it improves the current one. After completion, the algorithm runs a standard A* algorithm, which uses the computed lower bound distances as a heuristic to find the shortest path between s and t.

Algorithm 2.4 PHAST All-To-One Search

Input: array $B[x]$ that stores the tentative distances from x to target t
for all CH levels L from most to least important **do**
    for all up edges (x, y) in G_l^+ with $x \in L$ **do**
        if $B[x] < B[y] + l(x, y)$ **then**
            $B[x] \leftarrow B[y] + l(x, y)$
        end if
    end for
end for
Output: $B[x]$

This PHAST step however is comparatively expensive. In [SZ21], the authors propose a CH-Potentials algorithm. Algorithm 2.5 recursively follows the up-edges in G_l^+ to update the respective values and uses memoization to save time. Compared to algorithm 2.4, it is not computed before iterating over the priority queue. The heuristic is computed via the CH-Potentials algorithm whenever a new node is added to the queue.

Algorithm 2.5 CH-Potentials

Input: array $B[x]$ that stores the tentative distances from x to target t, which were computed by the backward CH search
Let $P[x]$ be an array that stores the memoized potential of a node; initially $P[x] = \perp$ for all $x \in N$
function POT(x)
    if $P[x] = \perp$ **then**
        $P[x] \leftarrow B[x]$
        for all up edges (x, y) in G_l^+ **do**
            $P[x] \leftarrow \min\{P[x], l(x, y) + \text{POT}(y)\}$
        end for
    end if
    return $P[x]$
end function

2.6 Connection Scan Algorithm

Compared to the before mentioned public transportation routing algorithms, the CSA [DPSW17] does not rely on a time-dependent or a time-expanded graph as input. Instead, it makes use of a timetable, a list of connections sorted by their departure time.
Formally, this timetable is a quadruple (S, C, T, F) with S being a list of stops, C being a list of connections, T being a set of trips and F being a set of footpaths. A connection is a vehicle that moves from one stop to another without interruptions. Such a connection c is defined as the quintuple $(c_{\text{dep\_stop}}, c_{\text{arr\_stop}}, c_{\text{dep\_time}}, c_{\text{arr\_time}}, c_{\text{trip}})$. Similarly, a footpath f is a link between two stops which can be traveled by foot. It is defined as the triple $(f_{\text{dep\_stop}}, f_{\text{arr\_stop}}, f_{\text{dur}})$.
There are multiple variants of the Connection Scan algorithm, fitted for different problem cases. For this work however, only the Earliest Arrival Connection Scan variant is relevant.
Similar to Dijkstra's and the A* Search Algorithm, CSA employs a tentative arrival time array which stores each stop's earliest known arrival time. However, in contrast to Dijkstra and A*, CSA does not maintain a priority queue. Instead, it iterates over all connections which are sorted by increasing departure time.
In every iteration, the algorithm tests whether the current connection c is reachable or not. To this end, the algorithm verifies whether it has visited a connection with the same trip ID as c or if the currently known earliest arrival time of the departure stop is earlier than the departure time of the connection.
After the algorithm is finished iterating over the connections, the result of the one-to-one query from s to t is the value of the arrival time array at index t.
Algorithm 2.6 uses an array S to keep track of the tentative arrival times and an array T which stores a bit for every trip. This bit is set whenever the algorithm first reaches the trip, indicating that this trip and every connection in it is reachable.
There are several optimizations that can be done to improve CSA's running time. For example, one can limit the amount of connections that need to be iterated over, by finding a first and a last connection. The first connection can be determined by employing a binary search that finds the first connection c^0 with a departure time higher than τ. The last connection, or the condition under which the algorithm stops execution, is the first iteration in which $S[t] \leq c_{\text{dep\_time}}$ holds true.

Algorithm 2.6 Connection Scan Algorithm

Input: source s, target t, departure time τ with $s, t \in N$ and $\tau \in \mathbb{N}$
Let S be an array; $S(n) \leftarrow$ *infinity* for $n \in N$
Let T be an array; $T(n) \leftarrow$ *false* for $n \in N$
for all footpaths f from s **do**
    $S[f_{\text{arr\_stop}}] \leftarrow \tau + f_{\text{dur}}$
end for
for all connections c increasing by $c_{\text{dep\_time}}$ **do**
    if $T[c_{\text{trip}}] ==$ *true* or $S[c_{\text{dep\_stop}}] \leq c_{\text{dep\_time}}$ **then**
        $T[c_{\text{trip}}] \leftarrow$ *true*
        $S[c_{\text{arr\_stop}}] \leftarrow \min\{S[c_{\text{arr\_stop}}], c_{\text{arr\_time}}\}$
        for all footpaths f from $c_{\text{arr\_stop}}$ **do**
            $S[f_{\text{arr\_stop}}] \leftarrow \min\{S[f_{\text{arr\_stop}}], c_{\text{arr\_time}} + f_{\text{dur}}\}$
        end for
    end if
end for
Output: $S[t]$

3 PUBPHAST

3.1 Approach

In this chapter, the A* search algorithm combined with the PHAST heuristic is implemented to be used on public transportation networks instead of road networks. This new algorithm called PUBPHAST[1] is implemented in C++. It is tested whether A* in combination with PHAST also provides efficiency advantages in public transportation. To conduct this test, PUBPHAST is compared to Dijkstra's algorithm and the CSA. The public transportation networks that are used in the tests are in GTFS format.
The implementation process is as follows: First, the original public transportation networks in GTFS format are processed to graph structures. Two different graph structures are needed for PUBPHAST. On the one hand, a time dependent graph is needed for the A* search. This graph is used to compute the actual shortest paths. On the other hand, the PHAST heuristic requires a graph consisting of lower bound edges. For this graph, the time variable is completely excluded. Each edge is compared to all other edges with the same source node, target node and trip ID and is only kept in this lower bounds graph if it is the edge with the minimal cost compared to its peers. Resulting from this procedure are two different graph files, one containing the lower bounds and one containing the whole, time dependent network.
The lower bounds graph is then expanded by a CH Constructor that creates the shortcut edges and CH ordering that are needed in the algorithm (see Section 3.3).
The lower bounds CH graph and the graph that stores the whole public transportation network are now used as inputs to the PUBPHAST algorithm.
Section 3.4 explains the implementation of the PUBPHAST algorithm: The A* first extracts both the CH lower bounds file and the whole network file to fitting data structures. It is then similarly implemented as the algorithm in [SZ21], using PHAST as a heuristic and is adjusted to the characteristics of public transportation networks.
In its foundation the CH-Potentials algorithm is the same as the PHAST all-to-one search, except for the optimization of using memoization. Thus, PUBPHAST makes use of CH-Potentials instead of implementing the standard PHAST algorithm.

3.2 Features and Restraints

First and foremost, PUBPHAST should be able to execute a one-to-one search in a public transportation network. This means, given an input $i = (s, t, d_0)$ with source node s, target node t and departure time d_0, it should return the arrival time a as output. Other features or restraints of this implementation are:

[1]Code can be looked up at https://github.tik.uni-stuttgart.de/st167120/Bachelorarbeit-DavidBruns

Transfer Times. They are implemented to create a realistic environment of public transit. Transfer times are considered by only allowing transfers (which are defined as choosing an outgoing edge that has a different trip ID than edge that was used to get to the current node) after a certain amount of time. In this implementation, this duration is set as five minutes. While this is only a simple approach to implementing transfer times, this should make the setting considerably more realistic than just leaving them out all together.

Day Limit. For the sake of simplicity, only edges of one day are extracted to create the whole network input graph. GTFS input data contains, as is closer described in Section 3.3.1 two text files called *calendar.txt* and *calendar_dates.txt*. The former gives information about in which time span and on which weekdays a trip is active, the later adds exception dates to this, that inform on which dates a trip is *not* active. This makes its possible to extract edges from only one day which can be determined by the user.
For the lower bounds graph, all edges are used to calculate the lower bounds. This is not a problem however, as underestimating the lower bounds is fine, although this might lead to a less effective PHAST implementation.

Footpaths. They are added using the *station_edges*, which will be described in Section 3.3.1. Footpaths allow the algorithm to "walk" inside a station to change platforms. GTFS uses the concept of stations and platforms. Only the platforms have edges, as the structure of stations defined by GTFS is hierarchical. One parent station has at least one and potentially many platforms. When extracting, *station_edges* are created between all platform nodes. This means one can walk to every platform of a station by using only one *station_edge* (as opposed to requiring multiple edges to get to the desired platform).
Due to this, the algorithm can restrict the use of footpaths to only one consecutive edge. No chains of more than one consecutive footpaths are allowed.

Zero Cost Edges. The GTFS inputs also include edges that have a cost of zero, meaning the departure time of one stop of a trip is equal to the arrival time of the next stop. Because the departure times are often equal to the arrival times, this creates situations in which the departure times of two subsequent stops of a trip are equal.
This poses a problem for the CSA, since the departure times within one trip can no longer be clearly sorted. It also causes difficulties when implementing PUBPHAST, because the query push can not be simply implemented by checking if the arrival time at a target node t is smaller than the current earliest arrival at that target. The inclusion of transfer times makes its necessary to also push nodes that arrive later at t. This makes endless cycles possible.
Due to this, the trips that contain zero cost edges are removed from the graph, as the remaining graph is still of sufficient size for the purpose of testing.

3.3 Deriving the Input Graphs

3.3.1 GTFS Format

The data used to test PUBPHAST is formatted in General Transit Feed Specification (GTFS) [DGNW11]. This format defines the formalities and rules for public transportation networks. Bus, train and other vehicles follow routes. Routes are used in cycles, multiple times a day. These iterations are called trips. Trips have their own *trip ID* and consist of a sequence of multiple stops.

Listing 3.1 Example for a GTFS stops.txt file

```
stop_name,parent_station,stop_id,stop_lat,stop_lon,location_type
's-Heerenberg Gouden Handen,,560069,51.87225,6.2473383,1
's-Heerenberg Gouden Handen,560069,241176,51.87228,6.247406,
's-Heerenberg Molenpoort,,64315,51.87649,6.247513,1
's-Heerenberg Molenpoort,64315,170535,51.87649,6.247513,
...
```

Each of these stops has an arrival time and a departure time.
GTFS stores this data in multiple text files. The relevant files for this implementation are *stop_times.txt*, *stops.txt*, *trips.txt*, *calendar.txt* and *calendar_dates.txt*. There are other files to store more information about the public transportation network available in GTFS, but those are not needed for PUBPHAST. The goal of this section is to create two graphs $G = (V, E)$ and $G_l = (V, E_l)$ from the source GTFS data.

Extracting Nodes

The nodes can be read from the *stops.txt* (see Listing 3.1) file by extracting the *stop_id* and location and saving it in a vector.
Important to note is the functionality of the *parent_station* column. This column defines the hierarchy of nodes. GTFS differs between stations and platforms. While stations have an empty *parent_station* entry, platforms use this field to reference their parent station. These parent stations however are not used as stops in trips. Due to this, only the platform stops need to be stored.
Platform stops with the same parent station are saved in additional vectors and are called *station_edges*. These stops are then connected by creating edges between all of them. The cost of these edges are set as the estimated walking duration which are calculated using a walking speed of 1.42 m/s and assuming direct paths between the coordinates. The *station_edges* vectors are later added to the edges extracted from by the GTFS data. This enables footpaths inside of a station and for example makes it possible to walk from platform 1 to platform 2 of the same station.

Extracting Edges

Edges are derived using the *stop_times.txt* file (Listing 3.2). This file lists the stops of each trip, their arrival and departure times and their number in the sequence of their respective trips. To get edges, one can now take two consecutive lines at a time and store the two nodes as source and target node of the edge. The edge cost is calculated by subtracting the departure time of the source node from the arrival time of the target node. The *trip_id* is also stored to be able to identify transfers between trips during route computation.
The files *calendar.txt* and *calendar_dates.txt*, which store the weekdays a trip is active and the exception dates, are used to sort out the edges that are not in the desired time span of the query.
Besides the day limit, another restriction being done is the removal of the trips that contain edges with a cost of zero. This is necessary because the CSA requires the departure times of the stops of one trip to be unequal.

Listing 3.2 Example for a GTFS stop_times.txt file

```
trip_id,arrival_time,departure_time,stop_id,stop_sequence,pickup_type,drop_off_type
1000092,20:48:00,20:48:00,87162,0,,
1000092,20:50:00,20:50:00,250088,1,,
1000092,20:51:00,20:51:00,371578,2,,
1013119,26:58:00,26:58:00,439766,0,,
...
```

In the next step, the previously saved *station_edges* are added to the vector that was used to store all edges extracted from the GTFS data. This vector is then sorted by ascending source node IDs. To get the lower bound edges E_l, one must now compute the lower bounds. This is done by iterating over the edges with the same source and target node and keeping only the edge with the lowest cost.

Graph Format

After extracting the nodes and the edges, both graphs are put out as two different text files, one containing the lower bounds, the other the whole network. The graph format is divided into three parts. In the first part, the amount of nodes and the amount of edges is displayed. After that come the nodes of the graph. And at the end, the edges are stored.

Nodes. The information that is stored for each node is their ID, their old ID and the longitude and latitude values of the nodes' coordinates. The old ID is the *stop_id* of the original GTFS data correspondent to the node. Because the parent station nodes are excluded and the enumeration of stops starts at one instead of zero, nodes are assigned a new ID. This new ID starts from zero and has no missing IDs in its enumeration.

Lower Bounds Graph Edges. The edges of the lower bounds graph only consist of the source node, the target node and the cost. This graph does not require any information about the time, so these three values are sufficient.

Whole Network Graph Edges. Edges of the whole network graph need some more data. Same as the lower bounds graph edges, they store the source and target nodes and the cost of each edge. But they also display the departure time (in seconds) and the trip ID of the edges. The trip ID of footpaths is defined as -3, so that they can be detected by the algorithm.

3.3.2 Computing Contraction Hierarchies

The lower bounds graph still has to be processed further to obtain the CH shortcuts and the node ordering. A CH-Constructor[2] by the Institute of Formal Methods in Computer Science of the Universität Stuttgart is used to process the lower bounds graph. This CH-Constructor creates a graph file which contains the original lower bounds graph plus the shortcut edges and an extra value

[2] https://theogit.fmi.uni-stuttgart.de/nusserae/chconstructor/, accessible only in Universität Stuttgarts network

Listing 3.3 *Edges*-Struct of the A*-Algorithm

```
struct Edge
{
  int target;
  int cost;
  int tripID;
  std::vector<int> departures;
};
```

that determines each node's CH-level. This level value can be used to order the nodes.
Its nodes now additionally store a value the determines the CH-level of each node, with which up- and down edges can be determined.

3.4 Implementation

3.4.1 Definitions

In this section, the data structures that are used in the implementation are defined and described. It is important to note that the PHAST and A* parts of the algorithm are implemented in different classes. In the code of the A* algorithm, an object of the PHAST class is created and is used to compute the heuristics' values. Due to this separation, one part this section is reserved for definitions in the A* class, the other for definitions of the PHAST class.

A*-Search Algorithm:

- *std::vector<Node> graph*
  The whole network graph that serves as input for the A* is stored in the *graph* vector. This vector contains elements of the struct *Node*. Each *Node* has a vector *edges* which is made of Elements of the struct *Edge* (see Listing 3.3). Each *Edge* stores a target node ID, the cost, the trip ID and a vector of integers called *departure_times*. Resulting from this, edges are not only differentiated by their source and target nodes, but also by their trip ID. The *departure_times* vector contains departure times of its particular edge. This vector is needed as trips can use the same connection between to nodes multiple times. In the further description of the algorithm, this data container is called $graph_{A*}$.
- *std::vector<int> arrivals*
  This vector contains the current earliest arrival time of each node. Its size is defined by the amount of nodes in the graph.
- *std::priority_queue<State, std::vector<State>, std::greater<State>> pq*
  The priority queue *pq* contains *states* $s = (heuristic, time, node\_id, trip\_id)$. It is ordered by the value of *heuristic* (with the smallest *heuristic* value being on top). To be able to differentiate it from the priority queue used in the PHAST implementation it is called PQ_{A*} in the following sections.

- *std::vector<int> lowerBounds*
  This vector stores the estimated lower bounds of the different nodes. Its size is also defined by the amount of nodes. The *lowerBounds* vector is used to cache heuristic values, as is later described in Section 3.4.3.

PHAST:

- *Graph g*
  The input lower bounds graph is stored in g. As can be seen in Listing 3.4, g entails the node_count of the graph, a vector that contains all nodes $n = (id, level)$ (with level being the CH-level) and two vectors that separately store the up- and the down-edges $e = (target, cost)$. An edge with a source s and a target t is an *up-edge* if the CH-level of t is greater that that of s. Vice versa, it is a *down-edge* if the CH-level of t is smaller that that of s. In the following this data container is called $graph_{PHAST}$.
- *std::vector<int> B, P*
  Both the vectors B and P store tentative distances (in seconds) to the target. Their size is defined by the amount of nodes in the lower bounds graph. While B is used to store the intermediate values computed by the backward CH-search, P contains the actual CH-Potentials.
- *std::priority_queue<State, std::vector<State>, std::greater<State» Q*
  This priority queue contains *states* $s = (node\_id, distance)$ and is ordered by *distance* (with the smallest distance value being on top). This priority queue is called PQ_{PHAST} in the following sections.

Listing 3.4 *Graph*-Struct of the PHAST-Algorithm

```
struct Graph {
  int node_count;
  std::vector<Node> nodes;
  std::vector<std::vector<Edge>> up_edges;
  std::vector<std::vector<Edge>> down_edges;
};
```

3.4.2 Data Structures

The nodes are stored in a vector and each respective node has a vector of outgoing edges. The edge structure is different for the lower bounds graph and the whole network graph. While the lower bounds graph only requires a target node and the edge cost as information for its edges, the whole network graph also needs the trip ID and the different departure times. A rough sketch of the organization of the whole network graph data structure can be seen in figure 3.1.
The trip ID is needed to identify transfers between trips. Different departure times result from multiple edges having the same source node, target node and trip ID in the original graph file. In these cases the different departure times are stored as an extra vector for each edge. Due to this, each edge has a unique combination of source node, target node and trip ID and has a vector that

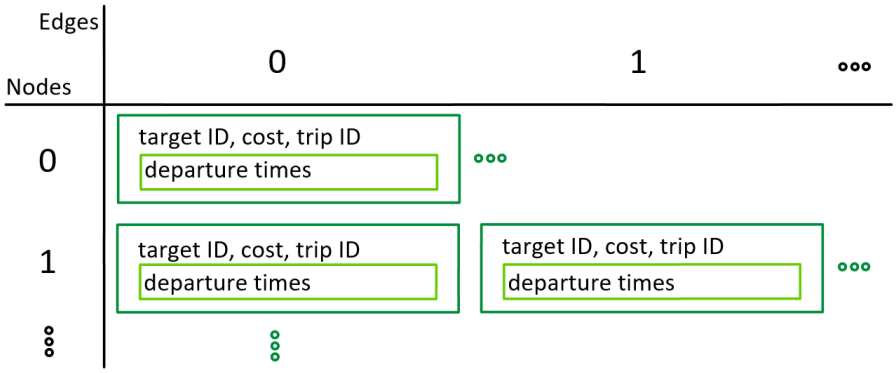

Figure 3.1: Graph Data Structure for Whole Network Graph

stores at least one and potentially multiple departure times.
The algorithm now has two main data structures, one graph data structure for the lower bounds graph and one graph data structure for the whole network graph.

3.4.3 Algorithm

Given a query consisting of source node s, target node t and departure time d, the algorithm first starts with an initialization phase. After that begins the computation phase of the A* algorithm. The A* is fundamentally implemented the same as the pseudo code given in Section 2.2, with the biggest difference being the usage of PHAST as heuristic function. When computing the heuristic, the algorithm uses an object of the PHAST-class to call the CH-potentials function which returns the estimated lowest distance to the query target t (in seconds). The final result is then calculated by subtracting the query departure time d from the value in $arrivals[t]$ (which is the earliest arrival time at target t) to get the minimal time it takes to traverse the graph from source s to target t.

Initialization Phase

The initialization first sees to resetting all values of vectors that contain distances ($arrivals$, B and P) to infinity. Also, the priority queues PQ_{A*} and PQ_{PHAST} are cleared.
Next, $arrivals[s]$ is set to the departure time d and $B[t]$ is set to 0. The first element of the priority queue PQ_{PHAST} is now pushed. It is defined as the state $s_{0,\text{PHAST}} = (t, 0)$.
After that, the backward CH-search with input t is run, to get the values of vector B. This backward CH-search only needs to be done once at the start of the algorithm. Its implementation can be seen in Listing 3.5. Remember that the queue Q that is used in the code corresponds to PQ_{PHAST} in this description of the algorithm. As mentioned above, the first element of this queue is the state $s_{0,\text{PHAST}}$. The algorithm now pops the top element from this queue until the queue is empty. For each state s that is taken from the queue, the algorithm looks at all down-edges with $s.id$ as source node. It checks, whether the value for B at the edge target is set to infinity *or* is set to a value greater than the cost of the edge plus value of B at the edge source. If that is the case, the value $B[edge.target]$ is updated and a new state n is pushed to PQ_{PHAST}.
To be able the push the first state to the priority PQ_{A*}, the heuristic that gives the estimated value for the earliest arrival time at t when departing at s is needed. Because the backward CH-search has

Listing 3.5 Implementation of backward CH-Search

```
void PHAST::backward_ch(const int t){
  State s;
  while(!Q.empty()){
    s = Q.top();
    Q.pop();
    for(Edge edge : g.down_edges[s.id]){
      if((B[edge.target] == INF) || (B[edge.target] > edge.cost + B[s.id])){
        B[edge.target] = edge.cost + B[s.id];
        State n; n.distance = B[edge.target]; n.id = edge.target;
        Q.push(n);
      }
    }
  }
}
```

already been done, it is now possible to run the CH-Potentials function which computes the lower bound estimate. The code for this function can be seen in Listing 3.8 and is closer described in the computation phase section.
After calculating the heuristic h, one can now push the state $s_{0,A*} = (d + h, d, s, UNDEF)$, with $d + h$ being the earliest estimated arrival time at the target t, d being the departure time, s being the source node and *UNDEF* being the trip ID.

Computation Phase

The implementation of the while-loop that entails the computation phase can be seen in Listing 3.6. The algorithm now takes the top state from the queue PQ_{A*} until it is empty.
First, the algorithm verifies whether it is possible to transfer between trips in the current state. To this end, it checks if *currentState.time* is smaller than the current earliest arrival value stored in *arrivals*[*currentState.node*] plus the fixed transfer time. If that is the case, no transfers are allowed. After that, the algorithm iterates over all outgoing edges of *currentState.node*. For each edge it first verifies whether it is a footpath edge or a public transit edge. If it is a footpath, the algorithm makes sure that the edge that was taken to arrive at *currentState.node* was not also a footpath (to avoid chains of footpaths). It then uses the `updateArrivalsAndPush` function (see Listing 3.7) to set the new *arrivals*[*edge.target*] value and push a new state to the priority queue.
If the edge is not a footpath, the algorithm checks if a transfer is needed to take this edge and, in case transfers are not allowed, skips the iteration. It also sets the variable *transfer_time* either to zero or to the fixed *TRANSFER_TIME* of five minutes, depending on whether a transfer is needed to use this edge.
In case the current time of the state added plus *transfer_time* is greater than the latest departure time of the edge, the iteration is also skipped.
If the algorithm has made it through all the checks, it now searches for the next possible departure time using binary search, updates the *arrivals* vector and pushes a new state to the queue PQ_{A*}.

Listing 3.6 Implementation of the A* While-Loop

```
    while(!pq.empty()){
      State currentState = pq.top();
      pq.pop();

      transferAllowed = true;
      if(currentState.time < arrivals[currentState.node] + TRANSFER_TIME){
        transferAllowed = false;
      }

      for(const Edge& edge : graph[currentState.node].edges){
        if(edge.tripID == FOOTPATH){
          if(currentState.tripID != FOOTPATH){
            updateArrivalsAndPush(currentState, edge, currentState.time);
          }
          continue;
        }

        if(edge.tripID != currentState.tripID && !transferAllowed){
          continue;
        }

        transferTime = 0;
        if(edge.tripID != currentState.tripID){
          transferTime = TRANSFER_TIME;
        }

        if(currentState.time + transferTime > edge.departures.back()){
          continue;
        }

        auto nextDeparture = std::lower_bound(edge.departures.begin(), edge.departures.end(),
currentState.time + transferTime);
        updateArrivalsAndPush(currentState, edge, *nextDeparture);
      }
    }
```

The `updateArrivalsAndPush` function computes the PHAST heuristics, writes to the *arrivals* vector and pushes new states in the priority queue PQ_{A*}.
It first computes the lower bound estimate from the edge target to the query target t. This is done with the `phast.pot` function (see Listing 3.8). If this lower bound distance is a negative number it means that there has been no path found between the edge target and t. In this case the function returns without updating *arrivals* or pushing to PQ_{A*}, as taking this edge will not lead to the shortest path from s to t.
After that, the function differentiates between three cases. If the edge target has not been visited yet (which means that $arrivals[edge.target]$ is still set to infinity), the *arrivals* entry for *edge.target* is set and a new state is created and pushed to the queue.
If the edge target has already been visited but the arrival time taking this edge (which is the departure

Listing 3.7 `updateArrivalsAndPush`-Function

```
    void updateArrivalsAndPush(const Edge& edge, const int departureTime){
      int lowerBoundDistance = phast.pot(edge.target);
      if (lowerBoundDistance < 0) { return; }

      if(arrivals[edge.target] == INF){
        arrivals[edge.target] = departureTime + edge.cost;
        pq.push(State(departureTime + edge.cost + lowerBoundDistance, departureTime + edge.
cost, edge.target, edge.tripID));
        return;
      }
      if(arrivals[edge.target] + TRANSFER_TIME > departureTime + edge.cost){
        pq.push(State(departureTime + edge.cost + lowerBoundDistance, departureTime + edge.
cost, edge.target, edge.tripID));
      }
      if(arrivals[edge.target] > departureTime + edge.cost){
        arrivals[edge.target] = departureTime + edge.cost;
      }
    }
```

time plus the edge cost) is smaller than the *arrivals* entry for the edge target plus the transfer time, a new state is created and pushed.
A specification of this is when the departure time plus the edge cost is even smaller than $arrivals[edge.target]$ (without the added transfer time). In this case the function also assigns the smaller arrival time to the *arrivals* entry at the edge target.

The estimated lower bounds, or the CH-Potentials are computed by the `pot` function (see Listing 3.8) of the PHAST class. This function recursively calls itself to traverse the *up-edges* of the lower bounds graph.
When the function is called, the vector B is already filled by the backward CH-search. The entries of B are now the shortest paths between each node and the query target t that are reachable by only using *down-edges*. If there is no *down-edges* path from a node n to t, then $B[n] = \infty$.
The first thing the function does is to check whether $P[x]$ is undefined (x being the argument given to the function, a node ID). If that is the case, $P[x]$ is set to $B[x]$. This means that the value of $P[x]$ is by default set to the shortest path between x to t using only *down-edges*.
Now, the function iterates over all outgoing *up edges* of x. It calls itself again, but this time with the edge target y as its argument. The return value of that function call is the lower bound estimate distance of the path from y to t. This return value is stored in the variable *pot*.
After that $P[x]$ is reassigned as the minimum of itself (which at this point is still equal to $B[x]$) and the edge cost plus *pot*. This is done because it is possible, that the shortest path between x and t is only made up of *down-edges*. In that case, the $P[x]$ ($= B[x]$) value is smaller than $edge.cost + pot$.
Because every shortest path between x and t is an up-down path, meaning there exists a middle node m with $x \rightarrow m$ being a path that only consists of *up-edges* and $m \rightarrow t$ being a path that only consists of *down-edges*, this function returns the correct lower bound estimate.

Listing 3.8 `Pot`-Function

```
int pot(const int x){
  if(P[x] == UNDEF){
    P[x] = B[x];

    for(Edge edge : g.up_edges[x]){
      int pot = pot(edge.target);
      if(P[x] != INF){
        if(pot != INF){
          P[x] = min(P[x], edge.cost + pot);
        }
        else {
          P[x] = P[x];
        }
      }
      else {
        if(pot != INF){
          P[x] = edge.cost + pot;
        }
        else{
          P[x] = INF;
        }
      }
    }
  }
  return P[x];
}
```

Optimizations

There are some optimizations that can be implemented to make PUBPHAST faster.

- The implementation of a stopping criterion for the A* while-loop makes sense, as it unnecessarily processes many nodes even after improving the $arrivals[target]$ value is no longer possible.
  This criterion is implemented by checking directly after popping the current state from the queue whether *currentState.heuristic* is greater or equal the value of *arrivals* at target t. If that is the case, we can stop the algorithm as the heuristic is a lower bound and all other heuristic values left in the queue are greater than the heuristic of the current state.

- Skipping queue iterations is also possible because it is not needed to take priority queue states into consideration whose current time is greater than the smallest known arrival time plus the fixed transfer time. In short, this means if

  $$currentState.time > arrivals[currentState.node] + TRANSFER\_TIME$$

  holds true, then the iteration can be skipped. This is correct because every possible state that can be pushed to the queue in this iteration could've also been pushed in the iteration which set the $arrivals[currentState.node]$ entry.

- Implementing a vector for the CH-Potential values is another way of saving time. Instead of computing the `pot` function each time `updateArrivalsAndPush` is called, it is more efficient to store the already computed potentials in a vector.
  Now, one can check at the beginning of the `updateArrivalsAndPush` function whether the CH-Potential of the respective node ID has already been calculated and, if that is the case, use the corresponding value of the vector.

4 Evaluation

4.1 Dijkstra and Connection Scan

This section gives a short overview of the implementation of Dijkstra's algorithm and the CSA which is used as comparison to the PUBPHAST algorithm. Essentially they were implemented as they were defined in Chapter 2, but to provide full context of the results presented in this chapter, the following gives a brief description of both implementations.

4.1.1 Dijkstra's Algorithm

Dijkstra's algorithm is implemented exactly the same as PUBPHAST, except that it does not use a heuristic. Instead, its priority queue is sorted by the current time of the state.
It only requires the whole network graph as input as it does not need to compute lower bound estimates.
The algorithm terminates once the time of the current state is greater than the earliest known arrival time at the target.

4.1.2 Connection Scan

Input Timetable: Opposed to PUBPHAST, this implementation does not extract nodes and edges but connections. A connection consists of source, target, departure time, arrival time and trip ID. Footpaths are stored separately. Each footpath entails a source, a target and the cost in seconds. The connections are sorted by their departure time with the lowest departure time value being the first connection and the highest departure time value being the last connection. Due to the exclusion of zero cost edges, each departure time of connections within the same trip is unequal.

Algorithm: The implementation of the CSA algorithm is done very akin to the pseudo code presented in Algorithm 2.6. A vector *stops* is used to store the earliest arrival times of the different stops and a vector *trips* is used to indicate whether a trip has already been visited or not.
The algorithm (see Listing 4.1) now iterates over all connections and updates the *stops* value if either the *trips* entry of the trip ID of the current connection is set, or the departure time of the trip is greater than the current earliest arrival time of the current stop plus the fixed transfer time.
The `updateStops` function that is used in the code checks whether the arrival time of the input connection is lower than the currently know earliest arrival time at that stop. If that is true, it updates the *stops* vector. It then iterates over all footpaths that have the arrival stop of the input connection as source and updates the *stops* entry for each of the target stops (if the footpath arrival time is an improvement over the old entry).

Listing 4.1 Implementation of the CSA For-Loop

```
for(int i = 0; i<connections.size(); i++){
  if(stops[connections[i].depStop] == INF){
    continue;
  }

  if(trips[connections[i].tripID]){
    updateStops(connections[i]);
  }
  else {
    if(stops[connections[i].depStop] + TRANSFER_TIME <= connections[i].depTime){
      updateStops(connections[i]);
      trips[connections[i].tripID] = true;
    }
  }
}
```

Optimizations: The usage of both a starting criterion and a stopping criterion creates a speed-up. Instead of iterating over the whole connections vector, it is possible to start at the "first" connection after the query departure time d, skipping all connections which have a departure time smaller than d.
It is also possible to stop the for-loop earlier by checking if the departure time of the current connection is greater than the *stops* value of the input target t. If that is the case, the value of $stops[t]$ can no longer be improved which means the algorithm can be terminated.

4.2 Method of Testing

This section explains the testing procedure. It takes a closer look at inputs and outputs of the tests. The tests were made with 10 000 randomly generated queries on four different public transportation networks. The algorithms under test, PUBPHAST, Dijkstra and CSA, were all tested with the same queries and public transportation network feeds.
Each query consists of a source node ID, a target node ID and a departure time. The source and target node have to be between zero and the node count of the respective graph. The departure time must be between 00:00:00 and 23:59:59.
The input networks are German public transit data. The four different variations are:

- *DE_FERN* ("Schienenfernverkehr Deutschland"), the long-distance rail transport of Germany. This network consists of ICE and IC train routes. The whole network graph generated from this feed has 939 nodes and 11 340 edges.
- *DE_REG* ("Schienenregionalverkehr Deutschland), the regional rail transport of Germany. This data set includes "S-Bahn" and "Regionalbahn" routes (regional railway routes). It results in a graph with 12 977 nodes and 693 733.

- *DE_NAH* (“Öffentlicher Nahverkehr Deutschland”), the close public transportation. In this network, the short range public transports like buses, trams or subways are represented. The resulting graph consists of 407 684 nodes and 10 721 255 edges.
- *DE_GES* (“Deutschland gesamt”), the combination of all of the above mentioned networks. All together, the graphs combine to 419 121 nodes and 11 462 026 edges.

These GTFS data sets were created from the officially released German public transportation schedules [19]. They contain data for January 2024. However, as mentioned in chapter 3, only the trips of one particular day is taken to create the input graphs. For this evaluation in particular, the public transport data of January 8, 2024 is used as input.
As using completely random query inputs and restricting the used trips on one day will lead to a substantial amount of cases where there is no path between source and target, the values that the algorithms are tested for are presented once for all 10 000 queries and once for the successful queries only.
The values that are observed in the tests are the average computation time (in ms) and the amount of visited nodes. The average computation times are presented for every algorithm. The amounts of nodes that are visited by the algorithm is only shown for PUBPHAST and Dijkstra, as CSA does not use a graph with nodes as underlying data set.
For PUBPHAST, the amount of visited nodes is divided into the nodes visited by the A* part of the algorithm and the PHAST part of the algorithm.
In the next section, the results of the test are shown. After that, in the comparison section the values are discussed and set side by side in a graphical representation.

4.3 Results

Feed	all queries			
	avg. comp. time	avg. visited nodes		
		A*	PHAST	total
DE_FERN	0.0704 ms	154.797	210.054	364.851
DE_REG	11.3714 ms	2 117	2 403.64	4 520.64
DE_NAH	221.591 ms	22 834	26 401.5	49 235.5
DE_GES	350.413 ms	46 167	52 366.4	98 533.4
Feed	successful queries			
	avg. comp. time	avg. visited nodes		
		A*	PHAST	total
DE_FERN	0.0776627 ms	189.409	255.269	444.678
DE_REG	13.0432 ms	2 232.24	2 572.06	4 804.3
DE_NAH	106.033 ms	18 985.5	22 436.4	41 421.9
DE_GES	242.035 ms	38 329.8	46506.1	84 836

Table 4.1: PUBPHAST Results

In total, processing 10,000 random queries, the *DE_FERN* graph had 4 056, the *DE_REG* graph had 4 911, the *DE_NAH* graph 883 and the *DE_GES* graph had 2 914 successful queries.
In Table 4.1, it can be observed that the PHAST heuristic visits more nodes than the underlying A* algorithm. Even though the *up-edges* and *down-edges* graphs are a lot smaller than the original one, the `pot` function is called so many times that it visits a lot of nodes. On top of that, the backward CH-search done in the initialization phase is essentially a one-to-all Dijkstra with no stopping criterion which also leads to a lot of additional visited nodes.

Feed	all queries		successful queries	
	avg. comp. time	avg. visited nodes	avg. comp. time	avg. visited nodes
DE_FERN	0.098 ms	262.821	0.116617 ms	359.142
DE_REG	20.7034 ms	4 003.55	27.5219 ms	5 059.08
DE_NAH	233.207 ms	32 800.8	198.143 ms	52 935.8
DE_GES	527.317 ms	87 055	618.073 ms	13 0753

Table 4.2: Dijkstra Results

For Dijkstra's algorithm, the average computation time of successful queries is substantially larger than the computation time of unsuccessful ones. The amount of visited nodes is also higher when the query turns out successful.
The CSA has fast computation times for the smaller graphs, but compared to PUBPHAST, it seems to get slower the bigger the graph is.

Feed	all queries	successful queries
	avg. comp. time	avg. comp. time
DE_FERN	0.0639 ms	0.0633629 ms
DE_REG	20.5353 ms	23.9485 ms
DE_NAH	269.824 ms	393.718 ms
DE_GES	522.136 ms	600.903 ms

Table 4.3: CSA Results

4.4 Comparison

PUBPHAST is the fastest of the three algorithms, only needing about two thirds of the time of CSA and Dijkstra in the biggest network. As can be seen in Figure 4.1, PUBPHAST really gets an advantage in the *DE_GES* graph. This might be because *DE_GES* is the most hierarchical graph of the four, as it contains every level of public transportation including regional trains but also close transport like trams and buses.
It can be observed that Dijkstra and CSA stay close to each other, growing steadily the bigger the input graph gets.
The average computation time for PUBPHAST on the other hand grows only until the *DE_NAH* graph. After that, the curve gets more flat. This confirms that PUBPHAST is especially well fitted for the *DE_GES* graph.

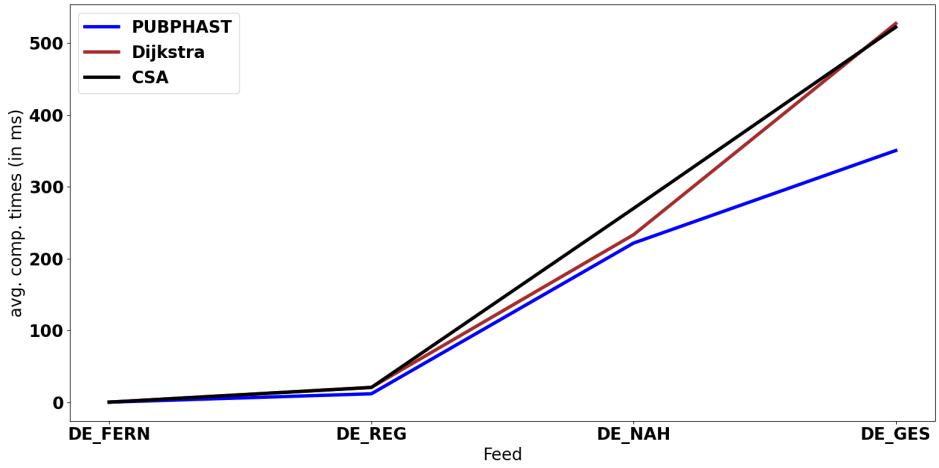

Figure 4.1: Average Computation times of all Queries

As mentioned above, PUPHAST probably produces the best results on the *DE_GES* graph because of its hierarchical nature. Compared to *DE_GES*, the other three graphs are much more plain. *DE_NAH* only covers the close and *DE_FERN* and *DE_REG* only cover the larger scale transportation. While routing, these three graphs might not have as many reoccurring shortest paths.
DE_GES is probably the most common public transportation graph type used in this test, as it entails all forms of public transportation. PUBPHAST doing well on this graph suggests that there is promise in using it for public transportation routing.

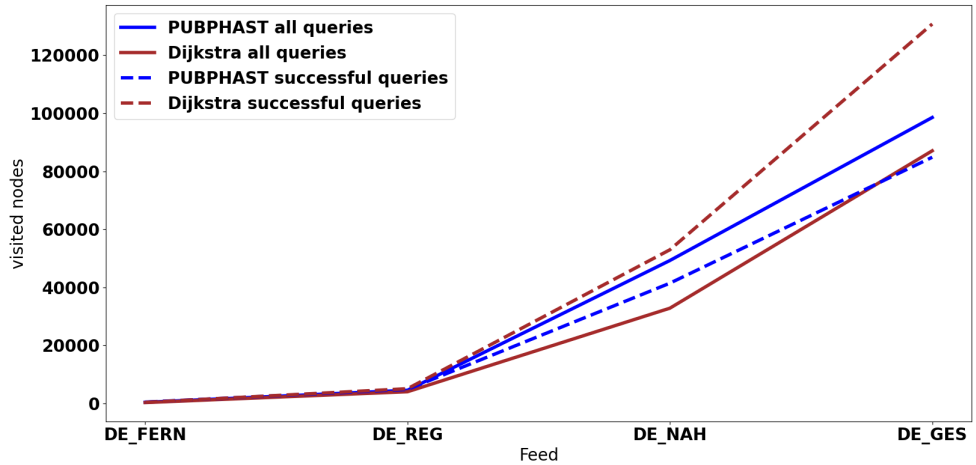

Figure 4.2: Average Amount of Visited Nodes

In Figure 4.2, the amount of visited nodes needed for PUBPHAST and Dijkstra are depicted. The dashed lines show the visited node count of successful queries. While, looking at all queries, PUBPHAST visits more unique nodes of the graph, the results for the successful queries are reversed. Here, PUBPHAST needs a significant amount of node visits less than the Dijkstra algorithm to

compute the shortest distance.
The discrepancy between successful and failed queries might stem from PUBPHAST being an unoptimized algorithm. It could be possible to find better stopping or skipping criterions which would results in lesser visited nodes.
Another possible reason for the comparatively large amount of node visits in failed queries could be a too relaxed choice of lower estimates. If the lower estimates are too low, PUBPHAST might inquire unnecessary paths.

Concluding this chapter, it became clear in the tests that PUBPHAST was the fastest of the three tested algorithms. This could imply that the usage of a PHAST heuristic has advantages in public networks as well. While PUBPHAST did use a larger amount of unique nodes in its computations when compared to Dijkstra, the fact the results were reversed when only looking at successful queries is promising. With a better choice of lower estimates one might be able to abandon unnecessary paths earlier, resulting in less visited nodes on failed queries.

5 Conclusion and Outlook

In this thesis, the hierarchical routing algorithm PHAST, originally intended for road networks, was implemented in a public transportation context. This was done by employing PHAST as a heuristic for an A* Search Algorithm and using a modified form of time-dependent graphs as input.
This implementation of A* with PHAST as a heuristic using public transportation networks as underlying data sets was called PUBPHAST. It was tested using GTFS public transportation data of Germany.
To this end, the GTFS source data that was used in this implementation was first transformed to two different graph structures, one whole network graph consisting of the entire schedule and departure times of the public transportation routes, and one lower bounds graph, including only lower bound edge costs. The lower bounds graph was then used to calculate the heuristic estimates with PHAST.
The tests were conducted using different types of public transportation networks, for example close vicinity public transportation or larger scale railway public transportation.
The results indicated that PUBPHAST shows promise and can create performance advantages in public transportation network routing. Even though it is not heavily optimized, it still outperformed Dijkstra's Algorithm and the CSA, delivering lower computation times and needing less unique node visits on successful queries.
This suggests that the hypothesis that was put forward in the introduction of this work was correct: Public transportation networks are similar to road networks, as it seems like their structure can also be taken advantage off by hierarchical routing. This claim was supported by PUBPHAST doing especially well on the most hierarchical graph used during testing.

Looking forward, it would be interesting to see more hierarchical routing algorithms get implemented for public transportation networks, as the concept shows potential. Intuitively, public transportation networks are similarly structured as road networks, being dense in urban areas and relying on few routes for transport in between them.
It is also intriguing to see an implementation of PHAST combined with A* tested more thoroughly. One could conduct expanded tests in a more realistic routing algorithm setting in which not only public transportation but also road networks and footpaths are taken into consideration.
Lastly, focusing on PUBPHAST, the algorithm could also be optimized further. For example, the space that is used for having two graph data structures for both A* and PHAST could be saved by combining them to one singular data container.
It would also be interesting to see more carefully calculated lower bounds, for instance by only using the minimum of costs of the edges that are still traversable at a given time of the algorithm.

In conclusion, the results produced by the experimental implementation of PUBPHAST imply that there are advantages, similar to road networks, of using hierarchical routing in public transportation networks and that this concept is worth putting more research into.

Bibliography

[19] *GTFS Deutschland*. 2019. URL: https://gtfs.de (cit. on p. 39).

[BDG+16] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, R. Werneck. "Route Planning in Transportation Networks". In: vol. 9220. Nov. 2016, pp. 19–80. ISBN: 978-3-319-49486-9. DOI: 10.1007/978-3-319-49487-6_2 (cit. on pp. 18, 19).

[Bel58] R. Bellman. "ON A ROUTING PROBLEM". In: *Quarterly of Applied Mathematics* 16 (1958), pp. 87–90. URL: https://api.semanticscholar.org/CorpusID:123639971 (cit. on p. 18).

[DGNW11] D. Delling, A. Goldberg, A. Nowatzyk, R. Werneck. "PHAST: Hardware-accelerated shortest path trees". In: vol. 73. June 2011, pp. 921–931. DOI: 10.1109/IPDPS.2011.89 (cit. on pp. 3, 17, 18, 21, 26).

[Dij22] E. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: July 2022, pp. 287–290. ISBN: 9781450397735. DOI: 10.1145/3544585.3544600 (cit. on pp. 3, 18, 19).

[DPSW17] J. Dibbelt, T. Pajor, B. Strasser, D. Wagner. "Connection Scan Algorithm". In: *Journal of Experimental Algorithmics* 23 (Mar. 2017). DOI: 10.1145/3274661 (cit. on pp. 3, 23).

[Flo62] R. W. Floyd. "Algorithm 97: Shortest path". In: *Communications of the ACM* 5 (1962), pp. 345–345. URL: https://api.semanticscholar.org/CorpusID:2003382 (cit. on p. 18).

[Goo05] Google. *General Transit Feed Specification Reference*. 2005. URL: https://developers.google.com/transit/gtfs/reference (cit. on p. 3).

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks". In: May 2008, pp. 319–333. ISBN: 978-3-540-68548-7. DOI: 10.1007/978-3-540-68552-4_24 (cit. on pp. 18, 20, 21).

[HKMS09] M. Hilger, E. Köhler, R. Möhring, H. Schilling. "Fast Point-to-Point Shortest Path Computations with Arc-Flags". In: vol. 74. July 2009, pp. 41–72. ISBN: 9780821843833. DOI: 10.1090/dimacs/074/03 (cit. on p. 18).

[HNR68] P. E. Hart, N. J. Nilsson, B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136 (cit. on pp. 3, 18, 20).

[LT77] R. J. Lipton, R. E. Tarjan. "A Separator Theorem for Planar Graphs". In: *Siam Journal on Applied Mathematics* 36 (1977), pp. 177–189. URL: https://api.semanticscholar.org/CorpusID:14218471 (cit. on p. 18).

[Pel00] D. Peleg. “Proximity-preserving labeling schemes”. In: *Journal of Graph Theory - JGT* 33 (Mar. 2000), pp. 167–176. DOI: 10.1002/(SICI)1097-0118(200003)33:33.0.CO;2-5 (cit. on p. 18).

[PSWZ08] E. Pyrga, F. Schulz, D. Wagner, C. Zaroliagis. “Efficient Models for Timetable Information in Public Transportation Systems”. In: *ACM J. Exp. Algorithmics* 12 (June 2008). ISSN: 1084-6654. DOI: 10.1145/1227161.1227166. URL: https://doi.org/10.1145/1227161.1227166 (cit. on p. 18).

[SZ21] B. Strasser, T. Zeitz. *A Fast and Tight Heuristic for A* in Road Networks*. 2021. arXiv: 1910.12526 [cs.DS] (cit. on pp. 3, 21, 22, 25).

[Wel14] M. Weller. *Optimal Hub Labeling is NP-complete*. 2014. arXiv: 1407.8373 [cs.CC] (cit. on p. 18).

[WWZ05] D. Wagner, T. Willhalm, C. Zaroliagis. “Geometric Containers for Efficient Shortest-Path Computation”. In: *ACM J. Exp. Algorithmics* 10 (Dec. 2005), 1.3–es. ISSN: 1084-6654. DOI: 10.1145/1064546.1103378. URL: https://doi.org/10.1145/1064546.1103378 (cit. on p. 18).

All links were last followed on January, 2024.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature