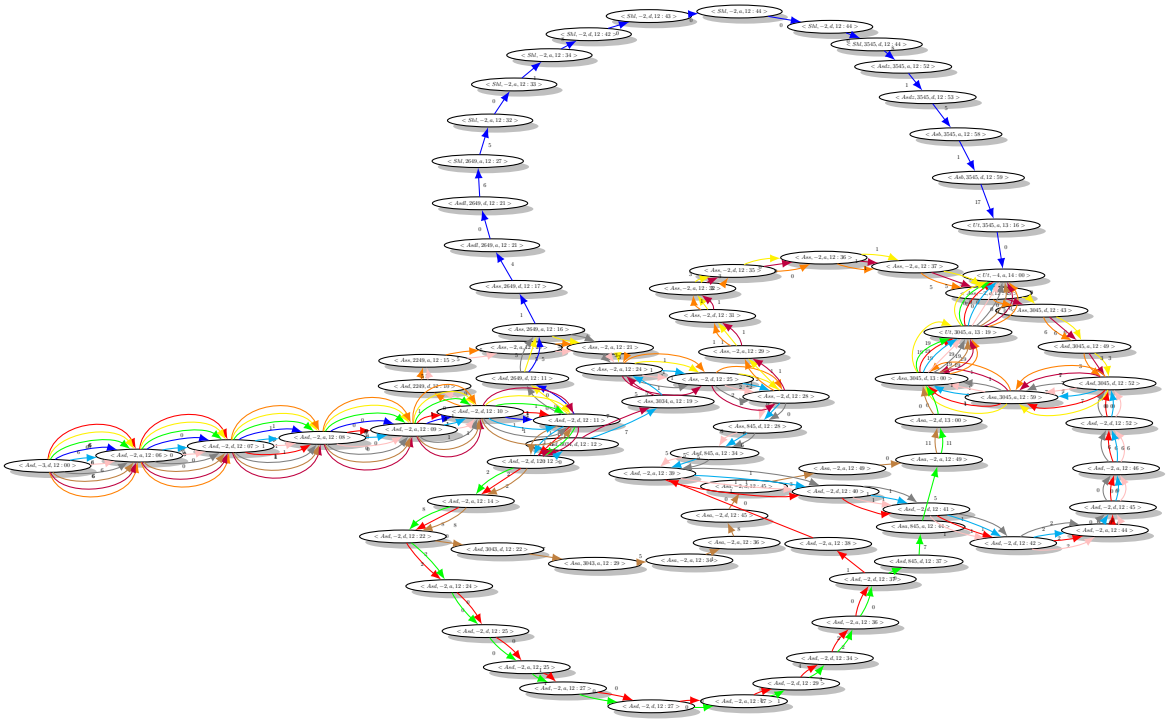
Applying Path Planning Algorithms to Train Schedules

Tim Soethout
3117901
Universiteit Utrecht

February 1, 2012



Bachelor's thesis Cognitive Artificial Intelligence (15 ECTS)
Supervisor: dhr. prof. dr. Vincent van Oostrom

Universiteit Utrecht

Contents

1 Introduction

This bachelor's thesis for Cognitive Artificial Intelligence began with an idea to do something with trains. Of course it had something to do with one or more of the foundations of the study of cognitive artificial intelligence. It soon became clear the emphasis would be on logic and computer science.

This rather broad subject had soon to become more defined than this vague notion of combining trains, logic and computers. There are many possibilities spreading from an artificial intelligence for building the perfect train transportation systems in game like environments like the classic computer game Transport Tycoon to giving the best travel advice to a person to get from A to B or generating a train time schedule from scratch.

As every enthusiastic student I set out to do as much as possible, but quickly limited my view to the shortest path problem. This is the problem of getting from A to B in the cheapest way. It soon became clear that to tackle this problem much more comes to the surface than simply running a shortest path algorithm such as the well known Dijkstra's algorithm.

The first step is to figure out a way to represent the train time table into a graph model on which the algorithms can be run. It all began with a very simple representation and all seemed trivial at first, but a lot more had to be done to really grasp all the features of the train time table. Also extensibility and generality had to be kept in mind.

The second step was to study some algorithms to generate the correct travel advise. This could be done with shortest path algorithms. The obvious Dijkstra's algorithm is of course described. But more interesting was the k shortest path algorithm. Especially for its relevance for this application to give alternative travel advise. The main focus lies on Eppstein's version of the k shortest path algorithm.

This thesis is divided in three main parts. The first part is the logic, theoretic part of the representation of the time table graph and can be found in section 2. The second part is a study into the shortest path problem and the different shortest path algorithm and is found in section 3. The third and biggest part is the implementation of the representation and the shortest path algorithms and found in section 4. The accompanying source code of the implementation is attached.

Before I start off with the content I would like to thank the following people for their support: Vincent van Oostrom for supervising me for this extended period of time, Jochem Bongaerts for constantly being available for discussions to re-order my mind on the subject and many more people for constantly reminding me to finish my bachelors thesis already. It is here at last.

2 Representation

2.1 Formal

2.1.1 Basic vertices and edges

We represent a time table of a train as a graph over which the algorithms can generate their response. We start by defining a directed graph G containing vertices (V) and edges (E).

$G = \langle V, E \rangle$[1]

Each vertex is a set with the following properties:

$V = \langle S, R, K, T \rangle$

Where S is a station, $R \in \mathbb{R}^+ \cup \{aa, ad, st\}$ where the numbers denote a route number or the arrival (aa) or departure (ad) vertex for an algorithm or station line (st), $K \in \{d, a\}$ indicates a departure or an arrival and T is the corresponding time. S is denoted with an abbreviation for a specific station, mapping can be found in the Appendix on page 25.

A vertex can be seen on figure 1.

Transitions (edges) are of form:

$E = \langle V, W, V \rangle$

The first V is the source vertex and the second V the target vertex. $W \in \{t, w\}$ describes whether the transit is a real transfer or a wait in a train, this is important for the number of hops. In most cases W will be w, only when a transfer from station line into a train is done t will be used. This will become clear in section 2.1.3. An edge can be seen on figure 2. Note that the ΔT is also displayed in the pictures

[1] The structure of the graph is inspired on the graphs used to create the dutch railways timetables[Sch08].

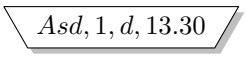


Figure 1: Vertex representing Amsterdam central station, route number 1 and departure at 13.30h.

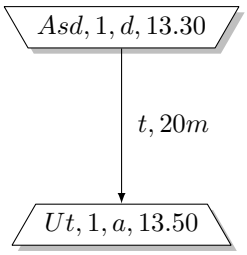


Figure 2: Graph portion representing a transition.

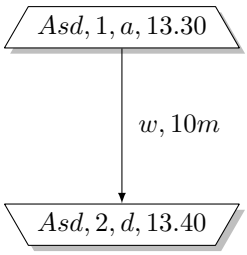


Figure 3: Graph portion representing a wait.

for convenience, but not in the definition. This value is derivable out of the data in the two vertices that are connected.

An example transition from a vertex with $K = a$ to $K = d$ denotes a wait (w) at a station for the next departure can be seen on figure 3. The route number (R) cannot be the same in case of a transit.

2.1.2 Costs

We can associate each edge in the graph with a certain costs. With this costs we can tune the shortest paths in the graph according to user preferences, such as the total trip time and the number of hops

Definition We can calculate the total trip time (c_{time}) in our model by taking the sum of all ΔT between the vertices. To do this we define a couple of functions: $time :: V \to T$ that given a vertex returns its value of T: $time(\langle s, r, k, t\rangle) = t$. $source, target :: E \to V$ which return respectively the source and target vertices of an edge: $source\left(\langle v_s, w, v_t\rangle\right) = v_s$ and $source\left(\langle v_s, w, v_t\rangle\right) = v_t$.

A path is a list of edges from which the the source and target vertices can be gotten with the *source* and *target* functions. Path $P = \langle e_1, e_2, \ldots, e_{i-1}\rangle$ where $e_i \in E$, contains $i - 1$ edges and this corresponds with path length of i vertices. Now we can construct equation 1, which is the same as taking the difference between the last and the current vertex in P, since time cannot be counted multiple times, this follows from the graph definition.

$$c_{time} = \sum_{k=1}^{i-1} \left(time\left(target\left(e_k\right)\right) - time\left(source\left(e_k\right)\right)\right) = time\left(target\left(e_{i-1}\right)\right) - time\left(source\left(e_1\right)\right) \quad (1)$$

The number of hops (c_{hops}) can be calculated by counting the amount of times an edge with $W = t$ gets passed during the path. We construct a function $label :: E \to W$ that gets the label of an edge: $label(v_1, w, v_2) = w$. With this function we can define the c_{hops}, see equation 2.

The $toInt :: bool \to int$ function maps the result of the equality to 0 or 1: $toInt = \{\langle true, 1\rangle, \langle false, 0\rangle\}$.

$$c_{hops} = \sum_{k=1}^{i-1} \left(toInt(label(e_k) \equiv t)\right) \quad (2)$$

The total cost is acquired by using a weight $f \in \mathbb{Q}$ per cost type to determine what to take in account more. If $C = \{c_1, c_2, \ldots, c_i\}$ is an ordering of all the available cost types with length n, we get equation 3 as the total cost. Setting f_i to 0 will effectively ignore the i-th cost type.

$$c_{total} = \sum_{k=1}^{n} \left(f_k \cdot c_k\right) \quad (3)$$

Realistic Scenarios in which the costs function can be used It is possible to define more kinds of costs. Such as:

- Hops during a specific time period (for example during the night).

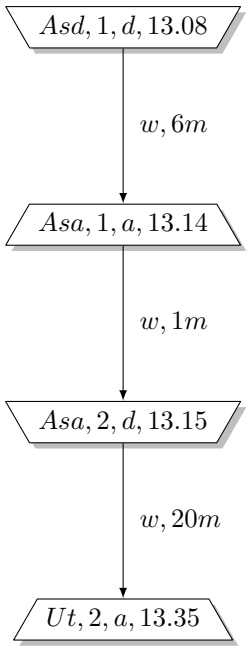


Figure 4: Example train trip.

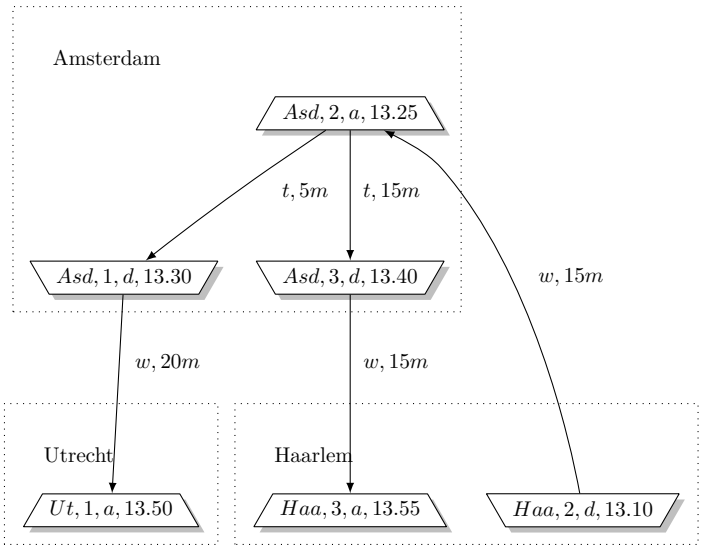


Figure 5: Example graph without station lines.

- Take into account the geographical location of the departure and arrival location.
- Penalise long wait times at a station.

Example Thus a simple train trip from Amsterdam to Utrecht can be seen in figure 4. The trip time is $6 + 1 + 20 = 27m$. The number of hops is 0. Hops will become clear when station lines are discussed in the next section.

In figure 5 we can see a graph with multiple routes and stations. Note that this is not an example of the full representation, but mere to get some intuÃ¯tion.

2.1.3 Station Lines

Modelling a timetable this way and connecting the vertices results in creating edges for every vertex with a T at a later point in time than the T of the current vertex with the same S. For n vertices in a station this would be $n!$ edges, meaning that worst case (in case there is only one unique S) there are $|V|!$ edges. This is far from desirable when finding the best path, since the amount of possible paths is huge!

To decrease the number of paths we create station lines. This is a line for each station on which trains arrive and depart. An example can be found in figure 6. For each non station line vertex we create a vertex on the station line, corresponding to the vertex, except for the route number, which we give the unique station line route number st. Another convenience of the station lines is that we can model a minimal transfer time. If we look at figure 6, we can see that the station line vertices corresponding with the arrival vertices ($K = a$), get a T of x minutes later; this x is the transfer time. Station line departure vertices ($K = d$), get the same T as their corresponding vertex, but get an edge with $W = t$, this way this edge is the equivalent of boarding a train and thus counted as a transit for cost purposes.

Note that if a person stays in the train that waits at station (R is the same), the station line gets never crossed, and no transfer ($W = t$) is counted. This is important for the hop cost function.

For each arrival or departure there are 2 vertices: one arrival vertex and one corresponding in the station line and with them an edge connection and an edge connecting the station line. The number of vertices in the graph thus become $2n$ where n is the number of arrivals and departures. The number of edges also becomes $2n$ for the edges in the station line plus an extra one if it concerns a wait, thus worst case $3n$.

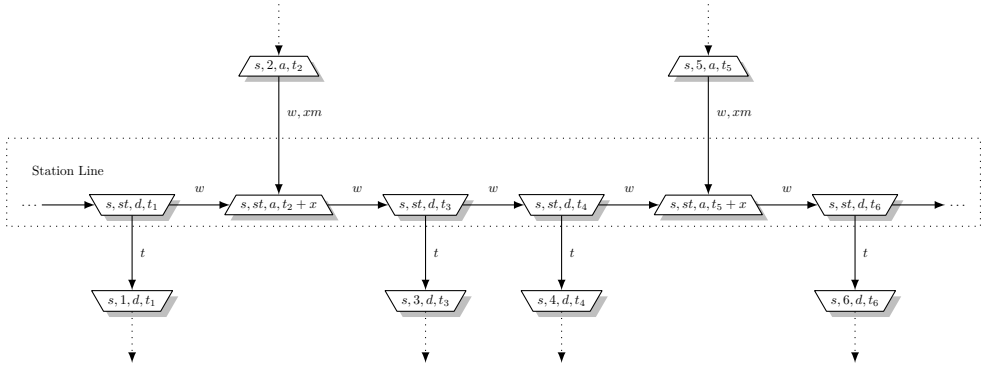


Figure 6: Example station line with corresponding arrival and departure vertices.

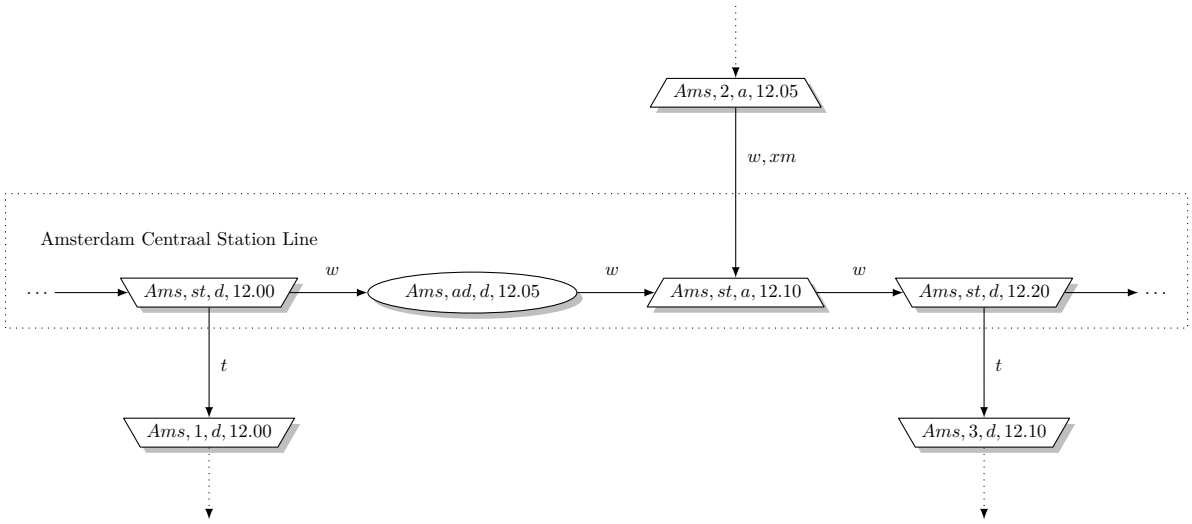


Figure 7: Example partial graph with start vertex.

Station lines double the number of vertices, but limit the number of edges and thus the number of possible paths.

2.1.4 Departure and Arrival vertices

We need to insert departure or arrival vertices to search depending on the requested trip advice.

Departure Vertex If we want to give a passenger a correct trip advice we have to model the departure in our graph. We insert a vertex s. This vertex will be of Kind (K) *departure (d)* and will contain the starting station and time. For the route number (R) we pick the algorithm departure *ad*.

For the departure vertex to be usable in the algorithms it needs to be connected to the rest of the graph. This can be easily done by inserting s into the station line of the station concerned.

The generation of this vertex can be done in $O(1)$, updating the station line results in recreation of the edges between the station line vertices.

An example can be found in figure 7.

Arrival Vertex To apply the path finding algorithms there also needs to be an end vertex to find the shortest path to. This vertex is also inserted into the station line of the corresponding station with a T

far enough in the future, at least larger than the largest value of T in the graph with a kind of *arrival* (a) and an R of the algorithm arrival aa.

If the user requests a specific arrival time, the simplest way to get the shortest path is to reverse all edges in the graph and insert the arrival station as if it were a start vertex as is described above.

2.1.5 Possible extensions

Walks between stations It is also possible to insert extra edges that denote a walk from station to station, for example when walking is quicker than waiting for a train. To make this work there would need to be some geographical information about the stations and to keep it from growing the graph too much there would need to be some kind of limitation to specify a reasonable distance to walk. These extra walks can be build by inserting for each possible walk and each departing vertex (in the station line) an extra pair of vertices into the station line, just like would be done for arriving and departing trains. This means the graph would grow about 50%. Also it would come in hand when there would be an extra possibility for the value of K to denote the walk, so it can be used for specific cost functions.

The use of walks is probably not a good idea since train stations are not close to each other most of the time and grows the graph unnecessarily large. Therefore it is not used in this model, though it could be trivially added.

Different platform inside a station The same way as walks between station as discussed before, it is possible to differentiate between platforms in stations and automatically determine a reasonable transfer time. This transfer time value is now determined up front for all transfers in the time table. It could also be used for better estimates for transfers to other forms of (public) transport.

2.2 Correct/usable?

This representation is usable because...

The graph has topological ordering (time).

2.2.1 Pruning

If we cut away connections (during a specific time).

3 Shortest Path Problem

3.1 Introduction

The shortest path problem is about finding a path between two vertices in a weighted graph with the least total weight. This notion can be written as:

$$\arg\min_{P} \left(\sum_{p \in P} w(p) \right)$$

where P are all possible paths between the two vertices and $w :: E \to \mathbb{R}$, returning the weight of the edge.

With the train time table represented as a graph, we can try to find the shortest path in this graph according to the requested arrival and departure locations.

3.2 Algorithms

3.2.1 Dijkstra's Algorithm

Dijkstra's Algorithm
Worst case runtime: $O(|V|^2)$

Introduction Dijkstra's Algorithm[Dij59] is one of the most well known shortest path algorithms. This is also why I applied it on the train trip planning problem. It should give a good comparison with the other algorithms.

Algorithm

Intuition Consider the following problem. We need to find a path from a vertex P to a vertex Q in a graph G. We start constructing the minimal paths from P to all other vertices in order of increasing length until Q is reached. This is done in an iterative fashion.

Pseudocode[Wik10]

```
1  function Dijkstra(Graph, source):
2      for each vertex v in Graph:              // Initializations
3          dist[v] := infinity                  // Unknown distance function
4                                               //   from source to v.
5          previous[v] := undefined             // Previous node in optimal
6                                               //   path from source.
7      dist[source] := 0                        // Distance from source to
8                                               //   source.
9      Q := the set of all nodes in Graph
10     // All nodes in the graph are unoptimized - thus are in Q
11     while Q is not empty:                    // The main loop
12         u := vertex in Q with smallest dist[]
13         if dist[u] = infinity:
14             break                            // all remaining vertices are
15                                              //   inaccessible from source.
16         remove u from Q
17         for each neighbor v of u:            // where v has not yet been
18                                              //   removed from Q.
19             alt := dist[u] + dist_between(u, v)
20             if alt < dist[v]:                // Relax (u,v,a)
21                 dist[v] := alt
22                 previous[v] := u
23     return dist[]
```

3.2.2 Bellman-Ford

Bellman-Ford<<RPackages,results=hide>>=
require(Hmisc)
Worst case runtime: $O(|E||V|)$

Introduction Bellman-Ford[Bel56] is also a well know shortest path algorithm, which is said to be slower than Dijkstra's algorithm. It is also included for comparison. The main advantage over Dijkstra's algorithm is that this algorithm is able to detect negative edges.

Algorithm

Intuition For each vertex each edge gets relaxed. This means that for each edge if the source shortest path costs plus the edge weight is smaller than the current shortest path costs of the target, then the costs to get to the target are updated. This means we get a value for each reachable vertex with the shortest path distance from the source. This is the shortest path if and only if there are no more target vertices which have a bigger shortest path than the predecessor plus the edge weight. If this is the case, then there are negative cycles in the graph.

Pseudocode[Wik11]

```
procedure BellmanFord(list vertices, list edges, vertex source)
    // This implementation takes in a graph, represented as lists of
    // vertices and edges, and modifies the vertices so that their
    // distance and predecessor attributes store the shortest paths.

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then v.distance := 0
        else v.distance := infinity
        v.predecessor := null

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge uv in edges: // uv is the edge from u to v
            u := uv.source
            v := uv.destination
            if u.distance + uv.weight < v.distance:
                v.distance := u.distance + uv.weight
                v.predecessor := u

    // Step 3: check for negative-weight cycles
    for each edge uv in edges:
        u := uv.source
        v := uv.destination
        if u.distance + uv.weight < v.distance:
            error "Graph contains a negative-weight cycle"
```

3.2.3 k Shortest Paths

k Shortest Paths
Worst case runtime: $O(|E| + |V| \log |V| + k\,|V| \log k)$

Introduction The k Shortest Paths Problem is the problem to generate a set $\{p_1, \ldots, p_k\}$ containing the k shortest paths between two vertices in a graph. The set is ordered by length: $\ell(p_n) < \ell(p_{n+1})$ for $n > 1$ and $\ell(p)$ is the length of path p.

This kind of problem is especially interesting considering the use in route planning. When a user queries a route planning system, it is very likely the case that he also wants to know one or more lesser that optimal paths. It isn't unlikely that the system gives out a path which is more desirable for the user than the the optimal path. Perhaps the user didn't specify the path cost influencing features (such as the number of hops) correctly or has other reasons to "like" a specific route more. This way the user is presented a couple of good paths next to the optimal path to choose from. These can be seen as travel alternatives.

Algorithm A possible implementation for the k shortest paths problem is suggested by David Eppstein[Epp94]. His algorithm will be explained in the following sections and implemented and compared with the other algorithms.

Intuition Eppstein starts by explaining a way to represent the paths. By representing the suboptimal paths as a derivative of the most optimal path. This way the paths don't have to be calculated separately and can be stored in the same data structure.

The algorithm itself results in a heap $H(G)$ that gives out the shortest paths is the correct order. $H(G)$ is built from path graph $P(G)$, which in turn is built out of $D(G)$ which contains heaps formed by looking at the edges not found in the shortest path tree.

Let's take a closer look at how this is done.

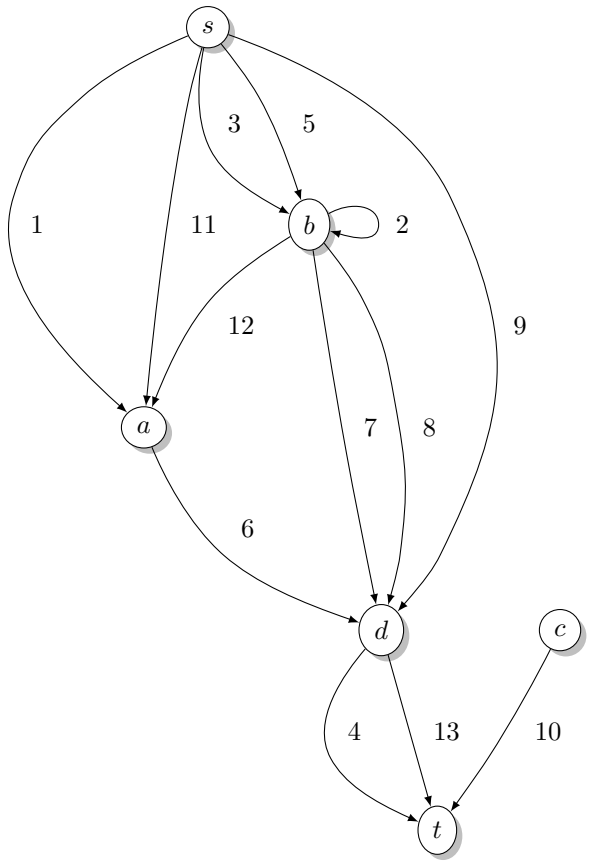


Figure 8: Example directed graph G with weights

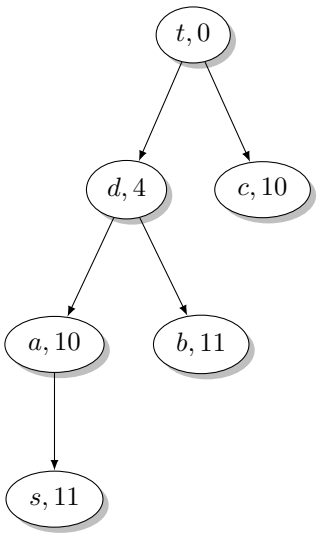


Figure 9: Shortest path Tree T with corresponding vertices and distances to target t in G

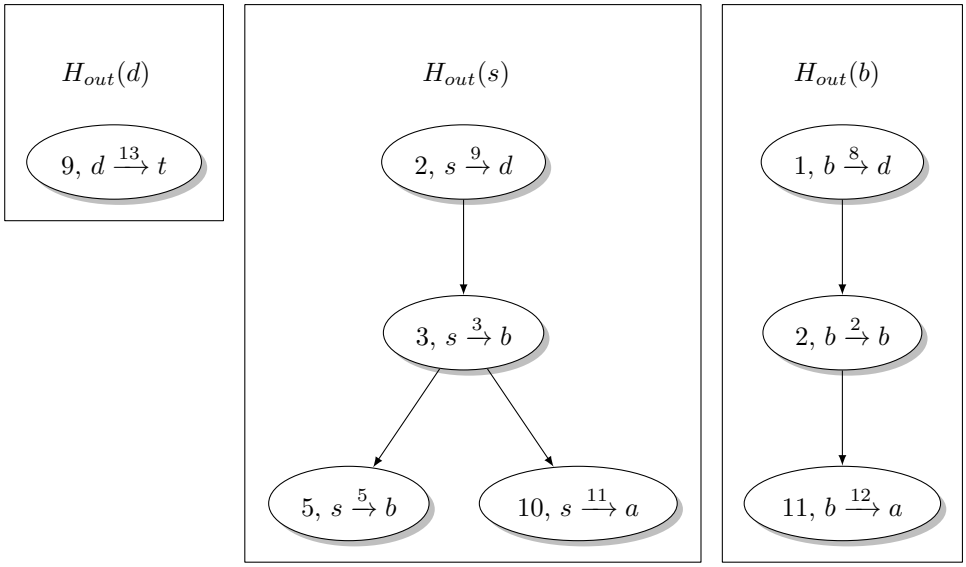


Figure 10: $H_{out}(v)$, with weight and corresponding edge from G

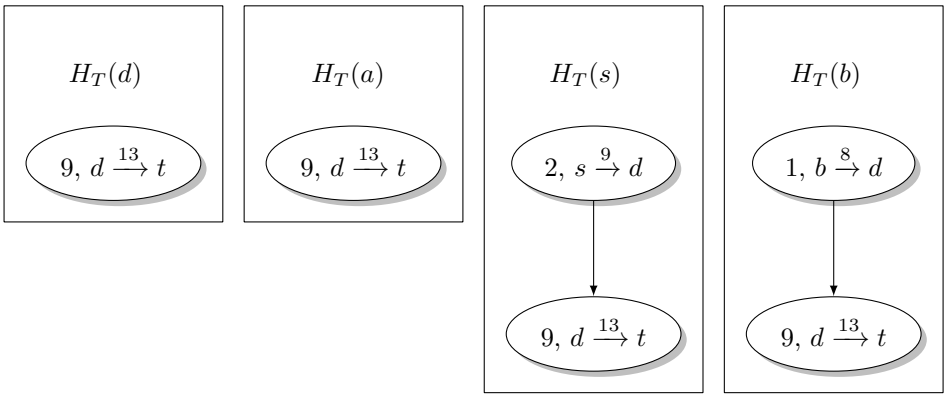


Figure 11: $H_T(v)$ with weight and corresponding edge in G

How it is done... The paths are specified by sequences of edges that are not in the shortest path tree. These edges can be found by generating a shortest path tree T[2] containing all the shortest paths to a target vertex t in G and subtracting this tree T from G, denoted as $G - T$. This sequence of edges is called *sidetracks*. Every path can then be uniquely defined by its sidetracks. The shortest path has no sidetracks. If we take a look at graph G from figure 8, the shortest path p_1 is via the edges with weights $\{1, 6, 4\}$ and $sidetracks(p_1) = \{\}$. The second shortest path p_2 with edges $\{9, 4\}$ is the same as $sidetracks(p_2) = \{9\}$.

$out(v)$ gives the edges tailing at vertex v in $G - T$.

For each vertex v in T one can create a heap $H_{out}(v)$ containing all the $out(v)$ and another heap $H_T(v)$ as seen in figure 10 and 11. Vertices with empty $H_{out}(v)$ or $H_T(v)$ are omitted from the figures.

$H_T(v)$ is constructed using $H_{out}(v)$. This is done by inserting the root node of $H_{out}(v)$ into $H_T(w)$, where w is the parent of v in shortest path tree T. Thus $H_T(v)$ can only be constructed when $H_T(w)$ is already constructed and should be done in an top-down fashion.

Out of all $H_T(v)$ and $H_{out}(v)$ a directed acyclic graph $D(G)$ can be build. This is done by taking the each vertex from $H_T(v)$ and connecting the corresponding vertices from $H_{out}(v)$to them.

Path graph $P(G)$ can then be constructed in five steps.

[2]This can be done in $O(m + n \log n)$ with Fibonacci heaps.

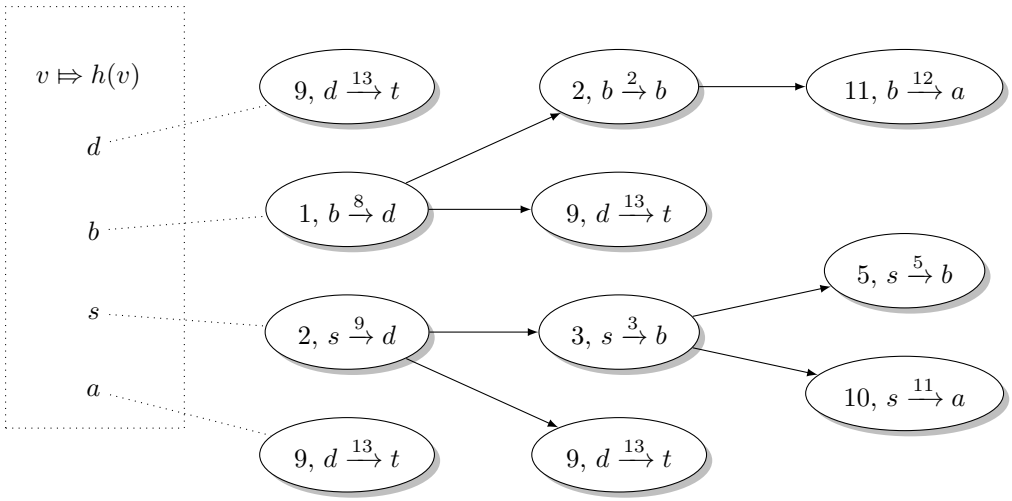


Figure 12: $D(G)$ with weight and corresponding edge in G and $h(v)$ which maps v to the corresponding vertex in $D(G)$

1. Add all vertices of $D\,(G)$ to $P\,(G)$
2. Add all edges from $D\,(G)$ with weight $\delta\,(v) - \delta\,(u)$, where are the edges in $D\,(G)$ from u to v and where $\delta\,(v)$ is the extra costs for the path containing sidetrack v. Note that a vertex v in $D\,(G)$ corresponds to an edge in G.
3. Add a root vertex corresponding to the source vertex s: $r(s)$
4. Add all edges from $G - T$
5. Add an edge between r and $h(s)$

The resulting $P(G)$ is seen in figure 13.

The last step is done constructing heap $H(G)$ by creating vertices for each path in $P(G)$ with as parent the vertices representing the path with one fewer edge. The weight of the heap vertices is the length of the corresponding path. Note that $H(G)$ can be infinite in size if the original graph contains cyclic paths and must thus be constructed in a lazy way.

Since each vertex in $H(G)$ corresponds with $sidetracks(p)$ of a path p in G, we can get the k shortest paths by removing the first k vertices from the $H(G)$. From $sidetracks(p)$ we can easily recover path p.

We can now see when looking at $P(G)$ that the root vertex s denotes the shortest path p_1, meaning no $sidetracks(p) = \{\}$, the next shortest path is the first path in $P(G)$ corresponding with $sidetracks(p_2) = \{9\}$.

The total time complexity of this implementation is $O\,(m + n \log n + k\, n \log k)$, where n and m are respectively the number of vertices and edges of G and k is the requested number of shortest paths.

Eppstein continues discussing a method to improve the space and time for this algorithm even more. This is not discussed in this thesis. For more information see the paper[Epp94].

4 Implementation

4.1 General

The implementation consists of multiple parts. One part is representing a train time table in the form of a graph as described in section 2. The second part is the implementation of the algorithms described in section 3, especially the *k shortest paths* algorithm discussed in section 3.2.3.

The objective in mind while creating the implementation for this thesis was to create high quality, easily adaptable code. This is achieved by using as much object oriented principles and design patterns as possible and the use of libraries wherever possible.

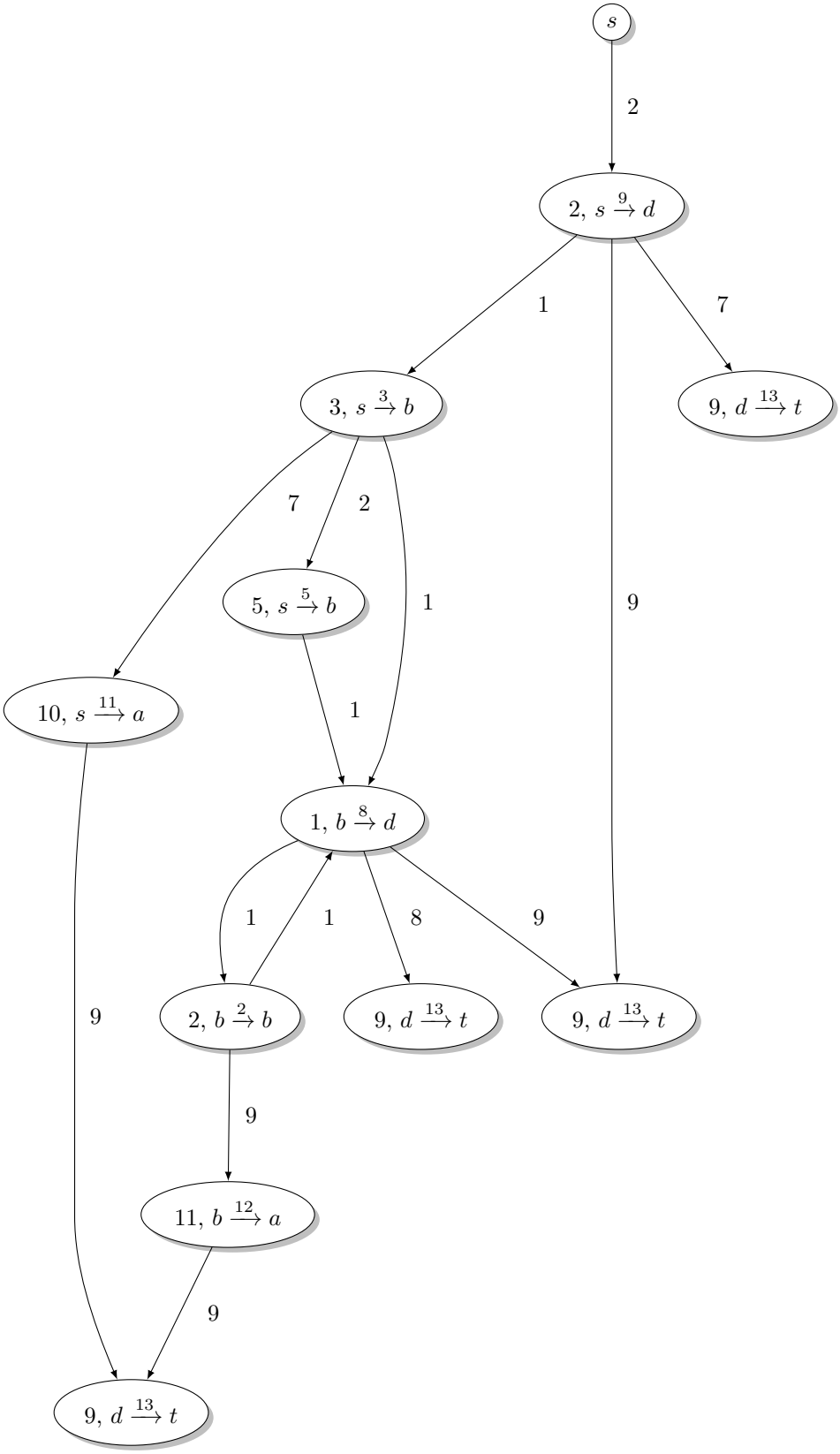


Figure 13: $P(G)$ with weight and corresponding edge in G and costs on the edges

The java source code provided with this document is built up in the following important packages:

- SCRIPTIE – executables and testing classes
- SCRIPTIE.DATASTRUCTURES – Generic data structures, mainly for use with the algorithms.
- SCRIPTIE.GRAPH.TIMETABLE – Representation for the time table graph as described in section 2.
- SCRIPTIE.GRAPH.ALGORITHMS – Algorithms that can be run on graphs, in particular SCRIPTIE.GRAPH.ALGORITHMS.KSHORTESTPATHS
- SCRIPTIE.OUTPUT.DOT – Representation of the Graphviz Dot file format and convenience functions for generating them from the JGraphT graph representations.

Throughout the code extensive use is made of the *Guava libraries*[gua11]. Guava enables functional programming style like programming using the Java language. It also provides possibilities to easily create anonymous functions. Other libraries used are *net.datastructures* as used in [GT06], *JGraphT*[NC09] and *Joda Time*[jod11].

The source code can be found with this document in digital form and a .PROJECT file is provided for opening in Eclipse.

Generating the graph can be done by running SCRIPTIE.TRAINTIMETABLEINITIALIZER. Without parameters it will generate the whole dataset.

Parameters are in this order and example between brackets: dataset ("data/voorbeeld.csv") name (voorbeeld) departureStationAbbreviation (asd) departureTime (1200) createDotOutput (false) debug (true). The debug flag gives extensive output on the command line and generates graphs of all k shortest path stages in the output directory.

This call also runs the k shortest paths algorithm for $k = 10$ and returns a nicely formatted graph of the paths taken.

4.2 Representation

4.2.1 Data Set

The Dutch Railways[3] was so kind to provide me with a real life test data set containing a time table from a Tuesday in 2009. This way the selected algorithms can be run on real data and hopefully we can conclude something about the runtime and complexity with this data.

The provided set contains 73447 records with the following structure:

```
"TreinNr";"AVK";"Spoor";"Dienstregelpunt";"Tijd"
104;"V";1;"Basels";1512
104;"A";1;"Baselb";1518
104;"V";1;"Baselb";1522
...
```

Each line of data could easily be converted to a vertex from the proposed graph definition.

The “Dienstregelpunt” column corresponds to a Dutch railway station or point of interest, a mapping of the abbreviations to their full names can be found in Appendix A.1.

One mismatch between the data and our model is the occurrence of ’K’ in the ’AVK’ column next to the known ’A’ (Arrival) and ’V’ (Departure). ’K’ means that the current record denotes an arrival and departure within the same minute. This could easily be fixed by inserting two vertices in the graph; one denoting the arrival, and one denoting the departure with both the same T (and S and R).

The edges can then be inferred from the vertices.

[3] Nederlandse Spoorwegen (NS) in the person of Leo Kroon.

4.2.2 Implementation

I created a java program that contains data structures to represent the graph. I used a Java library JGraphT that already implemented the basic graph features. The actual graph is represented in the SCRIPTIE.GRAPH.TIMETABLEDIGRAPH class. The program reads each line in the data set and creates a vertex representation of it. The next step is to infer the edges from the vertices.

See figure 14 for an example generated time table from a made up time table, specified in the appendix on page 28. This figure also gives a good impression on how the station lines and transfers discussed earlier work.

The system can be manipulated in a couple of ways to change the form of the resulting graph by changing specific variables of the TIMETABLEDIGRAPH class. Variable MINIMALTRANSFERTIME specifies hoe much time is minimally needed to transfer. This value is used in determining the wait costs for edges from arrival vertices to the station line. TIME_EDGE_COST_DIVIDER and HOP_EDGE_COST_DIVIDER specifies the weight divider of respectively the time cost and hop cost in the total weight of each edge in the graph as discussed in section 2.1.2.

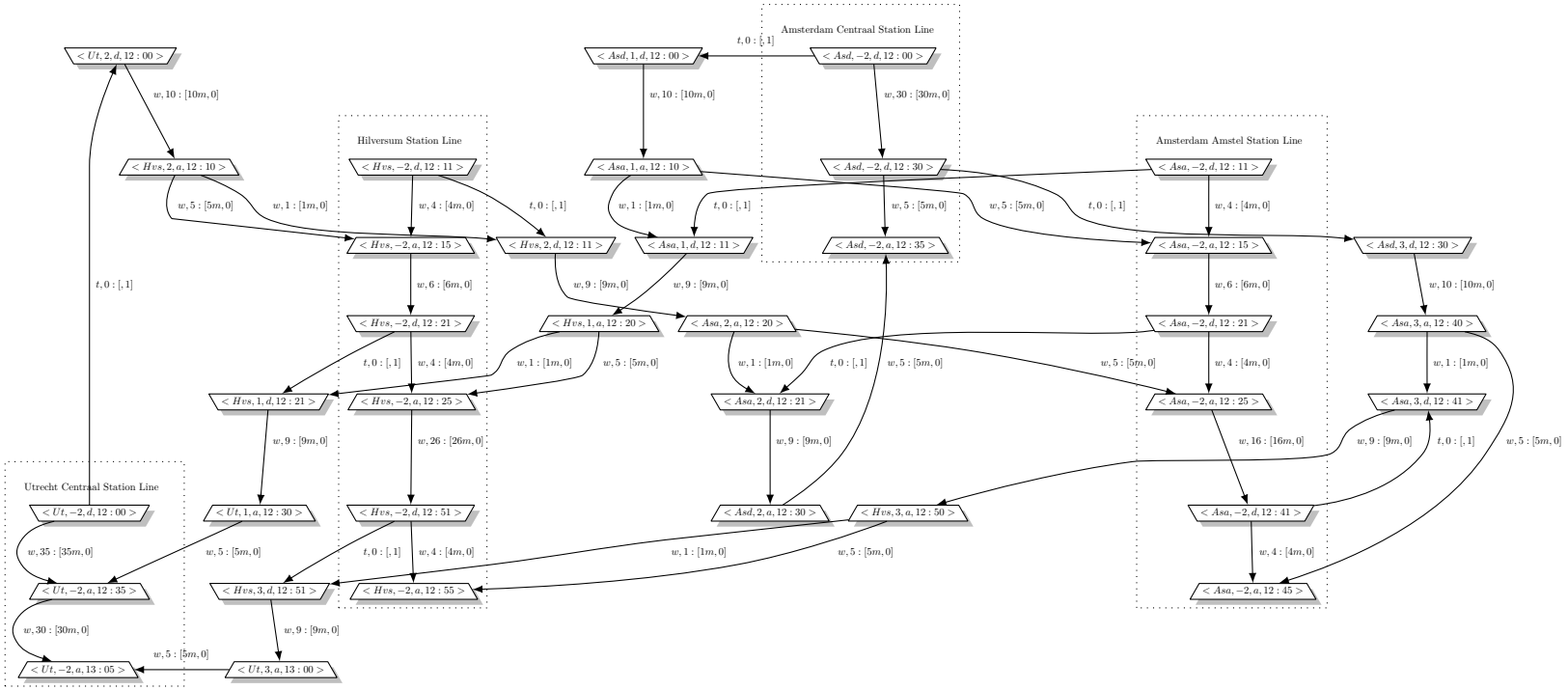


Figure 14: Example Time Table; Generated from a made up time table found in Appendix A.2.

Name	Dataset Size	Vertices	Edges	Mean CPU (ms)	Mean Memory (bytes)
Example	20	40	53	2188990	697392
Quarter	18362	46082	67315	230555660	41111023
Half	36724	103088	150236	504082810	82394598
Full	73448	207758	300031	1037188900	170077142

Table 1: Runtime results of generating the graph representation on several datasets.

4.2.3 Time Complexity

The order of complexity of the generations is in quasilinear time ($O(n \log(n))$) complexity. Given that n is the number of records, the creating the initial vertices costs n time. For generation of the edges first the vertices are sorted in $n \log(n)$ time. Then the sorted vertices get looped over with for each element at most $\log(n)$ for inserting it in the StationLine TreeSet, which results in $n \log(n)$. Last for each vertex in each station line an edge gets inserted resulting in n more steps. This totals in a worst case time complexity of $O(n + n \log(n) + n \log(n) + n)$, which is simplified to $O(n \log(n))$.

4.2.4 Performance

To test the performance of the generation it is run on the following 4 data sets. The made up time table found in Appendix A.2, and a quarter, halve and the whole of the Dutch Railways data set.

All tests are run on a MacBook Air 1.86 GHz Intel Core 2 Duo with 4GB of RAM with Java run options: -Xmx1636M -server. Each test is first run 100 times and the server option is used to make sure Java's HotSpot features finished optimising the code and the results are representative, then 100 samples are taken. The test can be done by running scriptie.graph.timetable.test.performance.

Figure 15 contains a set of box plots of the runs for each data set. Table 1 contains the mean values of the runs. In figure 16 we see the CPU Time and Memory Usage plotted against the size of the input dataset and that it scales nicely to the input size.

4.3 Algorithms

On the representation discussed in section 4.2 we can now run some shortest path algorithms to get the routing information to go from some source s to a target t. The algorithms used are documented in the following sections.

4.3.1 Dijkstra's Algorithm

For Dijkstra's algorithm the implementation found in JGraphT in the org.jgrapht.alg.DijkstraShortestPath class is taken.

Sanity Check If we take the same made up time table and run a simple Dijkstra on it from Amsterdam on 12:00h to Utrecht we get the follow output:

```
Total path length: 3164.0
From <Asd, 1, d, 12:00> to <Asa, 1, a, 12:10> (Cost 10.0)
From <Asa, 1, a, 12:10> to <Asa, 1, d, 12:11> (Cost 1.0)
From <Asa, 1, d, 12:11> to <Hvs, 1, a, 12:20> (Cost 9.0)
From <Hvs, 1, a, 12:20> to <Hvs, 1, d, 12:21> (Cost 1.0)
From <Hvs, 1, d, 12:21> to <Ut , 1, a, 12:30> (Cost 9.0)
Stations passed: [Asd, Asa, Hvs, Ut]
```

It nicely takes the direct route as we would expect.

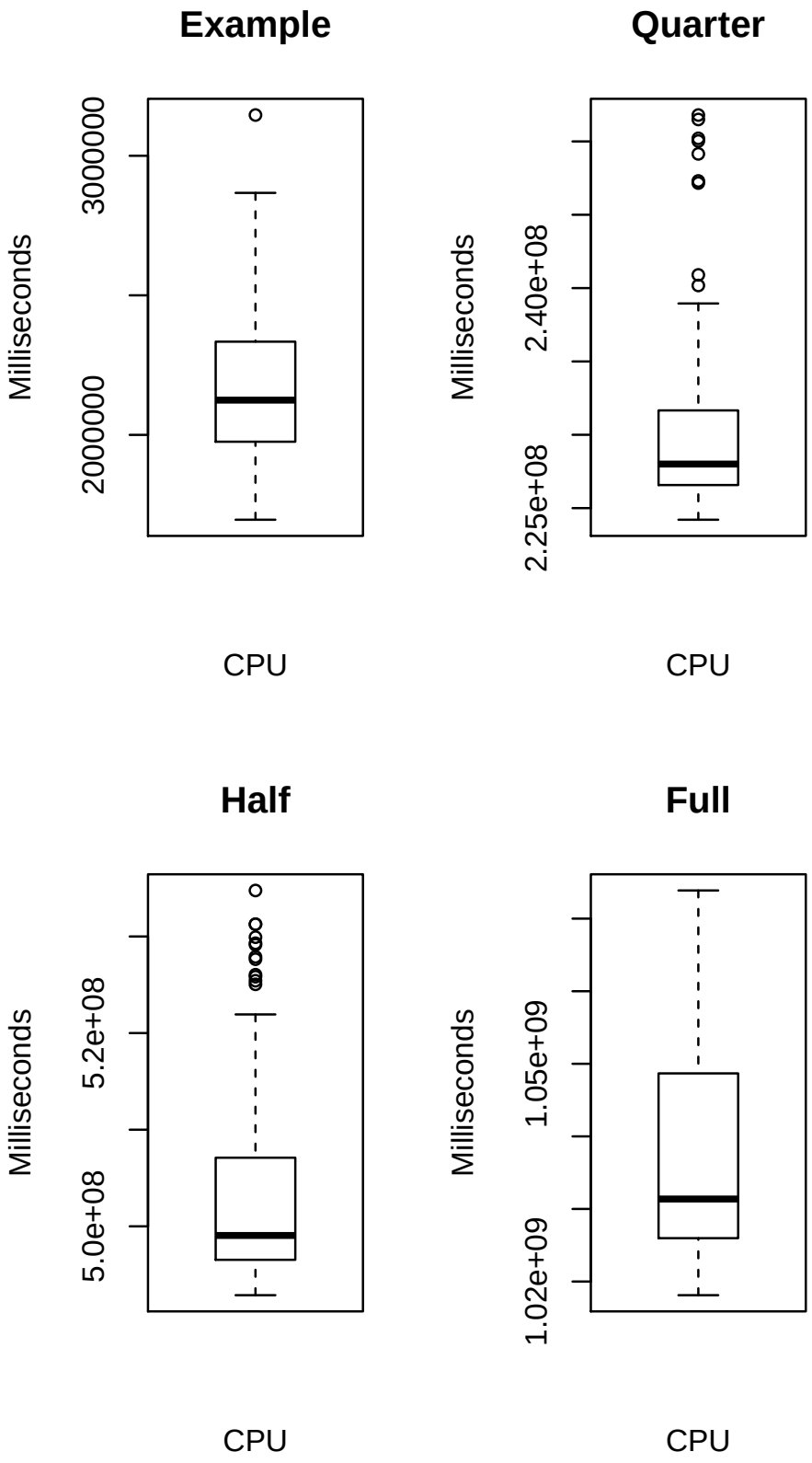


Figure 15: Box plot of data set graph generation CPU time results

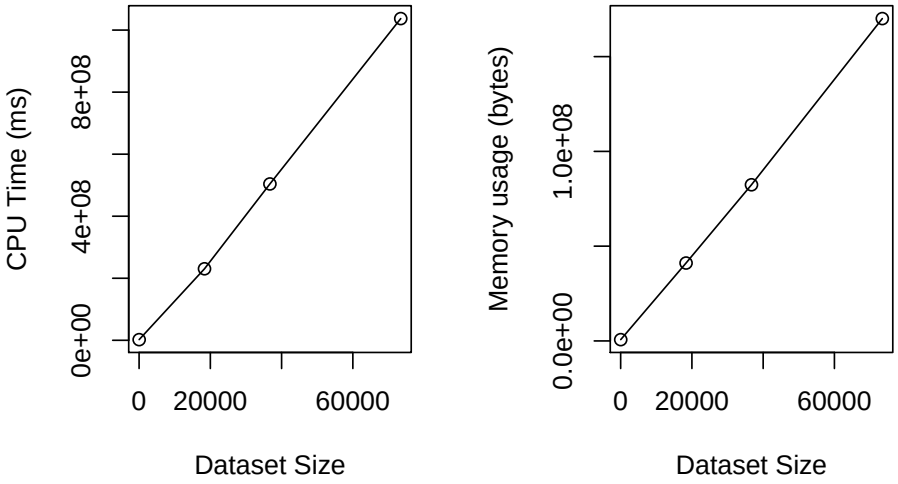


Figure 16: Line plots of dataset size against CPU Time and Memory Usage of graph generation.

4.3.2 Bellman-Ford

For the Bellman-Ford algorithm the implementation found in JGraphT in the ORG.JGRAPHT.ALG.BELLMANFORDSHORTESTI class is taken.

4.3.3 *k* Shortest Paths

I've implemented the k Shortest Paths algorithm as described by Eppstein and explained in section 3.2.3.

The source can be found in SCRIPTIE.GRAPH.ALGORITHMS.KSHORTESTPATHS. Each data structure such as $H_{out}(v)$, $H_T(v)$, $D(G)$, $P(G)$ and $D(G)$ has its representation in a class in the SCRIPTIE.GRAPH.ALGORITHMS package. KSHORTESTPATHS creates the data structures in order and returns the k shortest paths in function *getPaths(int k)*.

When an instance is created of KSHORTESTPATHS it initialises the following data structures in this order:

SCRIPTIE.DATASTRUCTURES.SHORTESTPATHTREE	Shortest path tree T
SCRIPTIE.GRAPH.ALGORITHMS.KSHORTESTPATHS.DG	$D(G)$
SCRIPTIE.GRAPH.ALGORITHMS.KSHORTESTPATHS.PG	Path graph $P(G)$
SCRIPTIE.GRAPH.ALGORITHMS.KSHORTESTPATHS.HG	Heap $H(G)$

Generating SHORTESTPATHTREE takes $n \log(n)$ time, where n is the number of vertices in the graph. This is build up from n steps for inserting the vertices in a fibonacci heap[FT87], then each vertex v in the heap is removed in $\log(n)$ per vertex, inserted in the resulting tree in constant time and the direct successors of v are updated in the heap. The total of vertices updated in the heap is equal m, where m is the number of edges in the graph, since each edge is traversed once. This makes the total number of time step $n + n(\log(n) + 1) + m$, which results in a time complexity of $O(n \log n + m)$ to create the shortest path tree T. This corresponds nicely to the complexity Eppstein claims.

The construction of $D(G)$ takes $O(m + n \log_2 n - (c+1)n)$, where $1.91 < c < 2$. according to Eppstein. This corresponds to the worst case number of vertices and each vertex can be constructed in constant time. $D(G)$ depends on T, $H_{out}(v)$ and $H_T(v)$. All $H_{out}(v)$ can be created in $O(\sum |out(v)|) = O(m)$, since together they contain a vertex for at most each edge in the graph. $H_T(v)$ takes at most n time since it has to look up it's parent in shortest path tree T. Thus $D(G)$ can be created in $O(n \log n + m)$.

The DG class constructs $D(G)$ in at least $O(n^2)$, this is due to that it is not clear if this implementation inserts the children of $H_{out}(v)$ and $H_T(v)$ efficiently.

PG is implemented straight forward by executing the steps as discussed in section 3.2.3. It contains

all vertices from $D(G)$ plus one root vertex and edges from $D(G)$ and $G - T$ plus one edge to connect the root. Since the construction of these vertices and edges can be done in constant time, this results in $O(n + 1 + m + 1)$ given $D(G)$ and $G - T$ are already constructed.

HG is implemented as an iterator that provides a breath first view of $P(G)$ in a lazy evaluated way. Each call to the iterator's NEXT function results in a *sidetrack* (p_n) corresponding to the shortest paths increasing in size. k calls to this iterator result in getting the representation of the k shortest paths in $O(k \log k)$.

This results in a total worst case time complexity of $O\left(n \log n + m + n^2 + n + m + k \log k\right)$ which is equivalent to $O\left(n^2 + m + k \log k\right)$. This is not the same as Eppstein's claim $O(m + n \log n + kn \log k)$. This is due to the unclear analysis of the complexity of $D(G)$.

4.4 Visualisation

To visualise this graph I've build a representation of the Graphviz format[EGK+02] and can be found in the SCRIPTIE.OUTPUT.DOT package. This is build up in a manner that it tries to capture as much as needed from the graphviz capabilities and as much generality as possible. To accommodate this it has factory classes to generate DOTGRAPH instances for ORG.JGRAPHT.GRAPH and SCRIPTIE.DATASTRUCTURES.ITREE.

Additionally it has support for LaTeX rendering for the *dot2tex* package[Fau06] to provide correct and aesthetic LaTeX rendering in this thesis. This can be accomplished in two ways. Either by implementing the LATEXREPRESENTABLE interface on items that are rendered as vertices or edges or by specifying the the RENDERLATEXNODELABELFUNCTION and RENDERLATEXEDGELABELFUNCTION when constructing the GRAPHDOTGRAPH item.

The Graphviz representation can be output to a .DOT file by calling the TOFILE function.

Some other features supported by this implementation is merging of DOTGRAPHS's and colouring given paths in the graph.

This file generated works fine with small to medium graphs, but with the full generated graph of the supplied data set with 207757 vertices and 21103 edges the dot renderer can not handle it anymore. The rendering with much less vertices and edges, which dot also can not really handle, is shown on the front page of this thesis.

4.5 Performance

The performance test is set up in the following way. The same datasets for graph generation are used, see section 4.2.4. The departure and arrival vertices are determined randomly by selecting a random station (which has at least one more vertex associated with it in the graph) and a random time for the departure vertex and another random station for the arrival vertex. This is done to make sure the test is done on all sizes of paths.

The tests are run on the same hardware and settings as the performance test of the graph generation, though only on the Example, Quarter and Half datasets because of OUTOFMEMORYERROR's occurring when the k shortest path algorithm was run. The test can be done by running SCRIPTIE.GRAPH.ALGORITHMS.TEST.PERFORMANCE.

For Dijkstra and Bellman-Ford first 100 tests were run to let the Java HotSpot optimiser finish optimising and then 100 runs on both algorithms. Figure 17 shows these results. Both algorithms show that the CPU Time grows when larger datasets are used, but not always in the same order as can be seen in the apparent horizontal lines. Since the departure and arrival vertices are randomly choses it can also be the case that little CPU Time is needed to get a result, since the found path can be short. The same holds for the memory usage.

If we look closely at the plots we can also see the worst case performance is quadratic is size with both algorithms. This does not come as a surprise as we have seen Dijkstra's algorithm operates in $O\left(|V|^2\right)$ in section 3.2.1.

It is also clear that the Bellman-Ford Algorithm performs worse. The power of Bellman-Ford lies in the ability to handle edges with negative costs, but these do not appear in the test datasets. In these test graphs $|V| < |E|$ and since Bellman-Ford runs in $O(|V||E|)$ it's bigger than the $O\left(|V|^2\right)$ of Dijkstra.

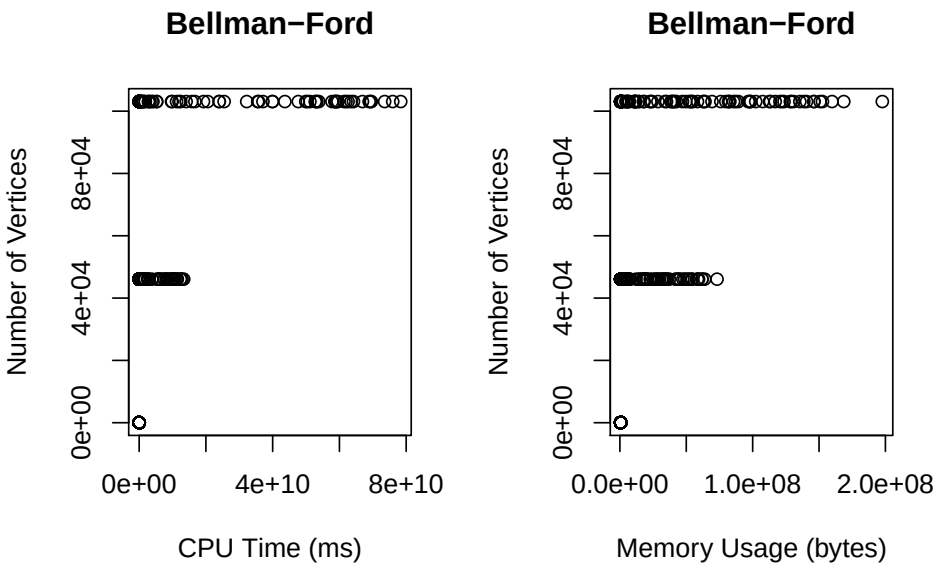


Figure 17: CPU Time and Memory usage of Dijkstra's algorithm run on the Example, Quarter and Half datasets.

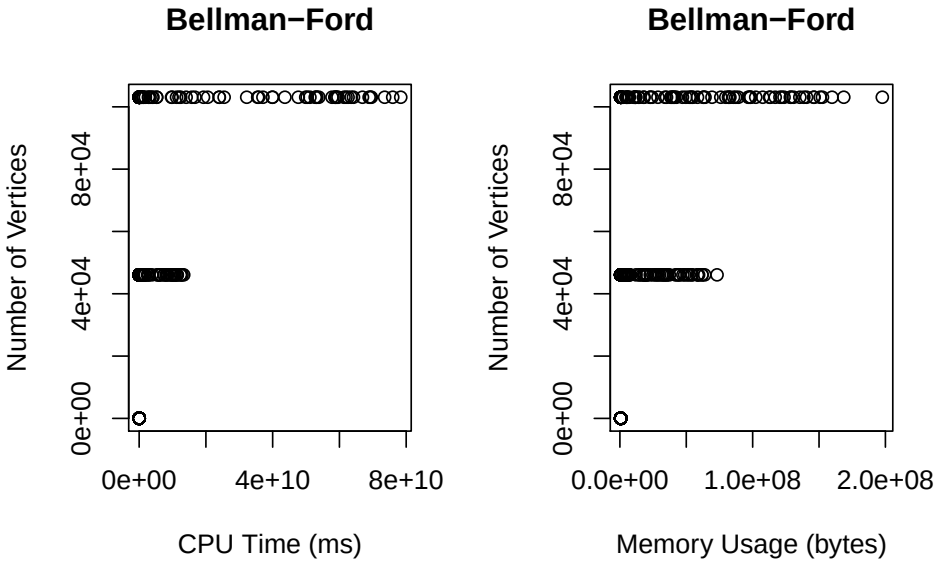


Figure 18: CPU Time and Memory usage of Bellman-Ford algorithm run on the Example, Quarter and Half datasets.

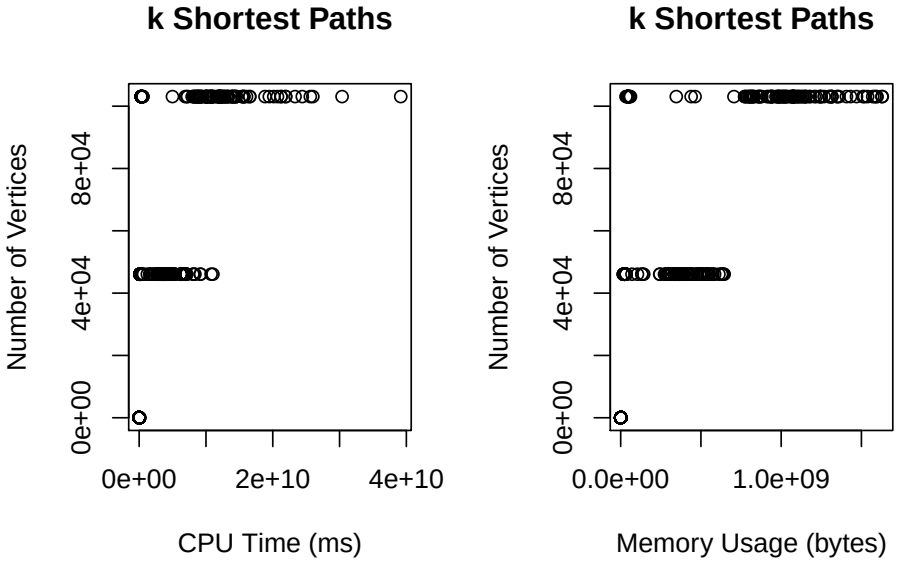


Figure 19: CPU Time and Memory usage of Eppstein's algorithm run on the Example, Quarter and Half datasets.

The results of the k shortest paths algorithm of Eppstein are a bit different. The test is run as often as possible issuing a restart whenever an OutOfMemoryError occurred. It may very well be the case that 'hard' paths were not found and the results are thus not really representable.

If we compare the results of the k Shortest Paths algorithm with the other two algorithm we see that it performs much worse than Dijkstra and equivalent to Bellman-Ford in CPU Time. In the memory case it perform worse than both algorithms.

It is clearly not the case that Eppstein outperforms Dijkstra although according to it's worst case complexity is better. This difference can have multiple causes. Firstly the implementation of Eppstein constructed here could be implemented incorrectly and not really be of the suggested worst case complexity .

Secondly it could outperform Dijkstra and Bellman on bigger datasets. This case could not be tested because the amount of memory usage is going out of bounds by using even moderately sized data sets.

Thirdly the class of graphs tested here, the time table representation graphs, are an issue for this algorithm, although this is very unlikely.

Of course the greatest gain in using Eppstein is the possibility to get the k shortest paths. Both Dijkstra and Bellman-Ford are single shortest path algorithms and will never give the second shortest path. Eppstein can give without much extra cost even more slightly suboptimal paths which can be extremely interesting in this kind of problem. The passenger requesting the trip advice might select slightly unoptimal parameters and the best result the algorithm provides might not be the best result for the passenger. In this case of alternatives he can select the most optimal trip advice for him.

5 Conclusion

The main focus of this thesis was to produce a graph representation of a train time table to be able to run shortest path algorithms on to provide travel advise.

This representation is discussed in section 2. The algorithms are discussed in section 3 and the implementation in section 4.

The Representation part is a model for a graph to represent train time tables. It can represent tranfers, waits in trains and supports multiple kinds of cost function that can be tailored to the travellers requirements. An optimisation is made with station lines, which reduce the number of edges from quadratic to linear with only a slight increase in number of vertices.

Several algorithms are described in the Shortest Path Algorithms part. The most important one is k shortest paths problem and a solution of it by Eppstein. His implementation is described in detail and later implemented.

The Implementation part contains an implementation of the Train Time Table Representation discussed and an implementation of Eppstein's algorithm in Java. Also provided is a performance test of both the representation as the algorithms and an analysis of it.

It turns out this implementation performs worse than expected. The performance tests turn out worse than Dijkstra's in both space and time. This could be due to a number of reasons. The most propable is that this implementation is suboptimal and not in the same time coplexity as Eppsteins algorithm.

6 Future Work

Possible future work on this application can be divided in two areas. The graph representation and the algorithms.

6.1 Graph Representation

It would be interesting to see in which ways the representation could be made more compact while retaining the needed information, but lowering the number of vertices and edges needed.

Extra appliances could be to add support for different travel forms besides trains. Walks, trips with own vehicles or other forms of public transport such as busses and airplanes could be added. This should be possible, though it should be kept in mind that the representing graph must not grow too quick.

The limited set of cost functions which determine the travel preferences of the passenger could be extended to contain more possibilities besides the now provided wait time and number of hops, as discussed in section 2.1.2.

This is similar to the problem that the graph needs to be rebuild from almost scratch when a new departure or arrival vertex is inserted into the graph. This results in the need to regenerate the edges of the graph. It would be nice to see a way to do this without recreating the graph. This is extremely useful if used in a public facing service in which the processing time to present a trip advice is critical.

Another neat feature would be to see how the graph can be pruned for two different cases. One is if certain train connections are unavailable. Another is to see if there are certain graph properties which result in the ability to prune the graph without losing the optimal paths for the current required trip advice, but will speed up the search for it.

6.2 Algorithm

This section mostly concerns Eppstein's k shortest paths algorithm.

The current implementation provided with this thesis is still not optimal in all locations. Although it follows the paper of Eppstein closely there are still enough places where optimisations could be made. It is unclear if Eppstein's time complexity is the same. Getting this clear is important.

Also Eppstein provides a way to even more improve the algorithm in his paper, which is not yet taken into account in this implementation. This could be added to improve space and time complexity.

It would also be nice to see more of a comparison with different k shortest paths algorithms instead of single shortest path algorithms such as Dijkstra and Bellman-Ford. The full potential of the algorithm is not shown in this way of testing.

References

[Bel56] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1956.

[Dij59] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[EGK+02] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz - open source graph drawing tools. In *Graph Drawing*, pages 594–597. Springer, 2002.

[Epp94] D. Eppstein. Finding the k shortest paths. 1994.

[Fau06] Kjell Magne Fauske. dot2tex - a graphviz to latex converter. Online, 2006.

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[GT06] M.T. Goodrich and R. Tamassia. *Data structures and algorithms in Java*. Wiley, 2006.

[gua11] Guava: Google core libraries for java 1.5+. Online, 2011.

[jod11] Joda time - java date and time api. Online, 2011.

[NC09] Barak Naveh and Contributors. Jgrapht, java graph library, 2009.

[Sch08] L. Schrijver. Wiskunde achter het Spoorboekje. *Pythagoras*, 48(2):8–12, 2008.

[Wik10] Wikipedia. Dijkstra's algorithm — Wikipedia, the free encyclopedia, 2010. [Online; accessed 20-July-2010].

[Wik11] Wikipedia. Bellman–ford algorithm — Wikipedia, the free encyclopedia, 2011. [Online; accessed 30-December-2010].

Appendix

A Source files

A.1 Station Abbreviations source file

```
Name;Abbreviation
Aalten;ATN
Abcoude;AC
Akkrum;AKM
Alkmaar;AMR
Alkmaar Noord;AMRN
Almelo;AML
Almelo de Riet;AMRI
Almere Buiten;ALMB
Almere Muziekwijk;ALMM
Almere Oostvaarders;ALMO
Almere Parkwijk;ALMP
Almere Poort;ALPO
Almiere Centrum;ALM
Alphen aan den Rijn;APN
Amersfoort;AMF
Amersfoort Schothorst;AMFS
Amersfoort Vathorst;AVAT
Amsterdam Amstel;ASA
Amsterdam Bijlmer ArenA;ASB
Amsterdam Centraal;ASD
Amsterdam Holendrecht;HLDT
Amsterdam Lelylaan;ASDL
Amsterdam Muiderpoort;ASDM
Amsterdam RAI;RAI
Amsterdam Sloterdijk;ASS
Amsterdam Sloterdijk Hemboog;ASSH
Amsterdam Watergraafsmeer;WGM
Amsterdam Zuid;ASDZ
Anna Paulowna;ANA
Apeldoorn;APD
Apeldoorn De Maten;APDM
Apeldoorn Osseveld;APDO
Appingedam;APG
Arkel;AKL
Arnemuiden;ARN
Arnhem;AH
Arnhem Presikhaaf;AHPR
Arnhem Velperpoort;AHP
Arnhem Zuid;AHZ
Assen;ASN
Baarn;BRN
Baflo;BF
Barendrecht;BRD
Barneveld Centrum;BNC
Barneveld Noord;BNN
Bedum;BDM
Beek-Elsloo;BK
Beesd;BSD
Beilen;BL
Bergen op Zoom;BGN
Berkel en Rodenrijs;RDR
Best;BETH
Beverwijk;BV
Bilthoven;BHV
Blerick;BR
Bloemendaal;BLL
Bodegraven;BDG
Borne;BN
Boskoop;BSK
Bovenkarspel Flora;BKF
Bovenkarspel-Grootebroek;BKG
Boxmeer;BMR
Boxtel;BTL
Breda;BD
Breda Grens;BDGR
Breda Prinsenbeek;BDPB
Breukelen;BKL
Brummen;BMN
Buitenpost;BP
Bunde;BDE
Bunnik;BNK
Bussum Zuid;BSMZ
Capelle Schollevaar;CPS
Castricum;CAS
Chevremont;CVM
Coevorden;CO
Cuijk;CK
Culemborg;CL
Daarlerveen;DA
Dalen;DLN
Dalfsen;DL
De Vink;DVNK
Deinum;DEI
Delden;DDN
Delft;DT
Delft Zuid;DTZ
Delfzijl;DZ
Delfzijl West;DZW
Den Dolder;DLD
Den Haag Centraal;GVC
Den Haag HS;GV
Den Haag Laan van NOI;LAA
Den Haag Mariahoeve;GVM
Den Haag Moerwijk;GVMW
Den Haag Ypenburg;YPB
Den Helder;HDR
Den Helder Zuid;HDRZ
Deurne;DN
Deventer;DV
Deventer Colmschate;DVC
Didam;DID
Diemen;DMN
Diemen Zuid;DMNZ
Dieren;DR
Doetinchem;DTC
```

Doetinchem De Huet;DTCH
Dordrecht;DDR
Dordrecht Amstelwijk;DDRA
Dordrecht Stadspolders;DDRS
Dordrecht Zuid;DDZD
Driebergen−Zeist;DB
Driehuis;DRH
Dronrijp;DRP
Duiven;DVN
Duivendrecht;DVD
Echt;EC
Ede Centrum;EDC
Ede−Wageningen;ED
Eijsden;EDN
Eijsden Grens;EDNG
Eindhoven;EHV
Eindhoven Beukenlaan;EHB
Elst;EST
Emmen;EMN
Emmen Bargeres;EMNB
Emmen Delftlanden;EMND
Enkhuizen;EKZ
Enschede;ES
Enschede De Eschmarke;ESDE
Enschede Drienerlo;ESD
Ermelo;EML
Etten−Leur;ETN
Eygelshoven;EGH
Franeker;FN
Gaanderen;GDR
Geerdijk;GDK
Geldermalsen;GDM
Geldrop;GP
Geleen Oost;GLN
Geleen−Lutterade;LUT
Gilze−Rijen;GZ
Glanerbrug;GBR
Goes;GS
Goor;GO
Gorinchem;GR
Gouda;GD
Gouda Goverwelle;GDG
Gramsbergen;GBG
Grijpskerk;GK
Gronau Grens;GG
Groningen;GN
Groningen Europapark;GERP
Groningen Noord;GNN
Grou−Jirnsum;GW
Haarlem;HLM
Haarlem Spaarnwoude;HLMS
Haarlem Zuid;HLMZ
Halfweg;HW
Harde 't;HDE
Hardenberg;HDB
Harderwijk;HD
Hardinxveld−Giessendam;GND
Haren;HRN
Harlingen;HLG
Harlingen Haven;HLGH
Heemskerk;HK
Heemstede−Aerdenhout;HAD
Heerenveen;HR
Heerhugowaard;HWD
Heerlen;HRL
Heerlen Kissel;HRLK
Heeze;HZE
Heiloo;HLO
Heino;HNO
Helmond;HM
Helmond Brandevoort;HMBV
Helmond Brouwhuis;HMBH
Helmond 't Hout;HMH
Hemmen−Dodewaard;HMN
Hengelo;HGL
Hengelo Gezondheidspark;HGLG
Hengelo Oost;HGLO
Hertogenbosch 's;HT
Hertogenbosch 's Oost;HTO
Herzogenrath Grens;HZG
Hillegom;HIL
Hilversum;HVS
Hilversum Noord;HVSN
Hilversum Sportpark;HVSP
Hindeloopen;HNP
Hoek van Holland Haven;HLD
Hoek van Holland Strand;HLDS
Hoensbroek;HB
Hollandsche Rading;HOR
Holten;HON
Hoofddorp;HFD
Hoogeveen;HGV
Hoogezand−Sappemeer;HGZ
Hoogkarspel;HKS
Hoorn;HN
Hoorn Kersenboogerd;HNK
Horst−Sevenum;HRT
Houten;HTN
Houten Castellum;HTNC
Houthem−St. Gerlach;SGL
Hurdegaryp;HDG
IJlst;IJT
Kampen;KPN
Kapelle−Biezelinge;BZL
Kerkrade Centrum;KRD
Kesteren;KTR
Klarenbeek;KBK
Klimmen−Ransdaal;KMR
Koog Bloemwijk;KBW
Koog−Zaandijk;KZD
Koudum−Molkwerum;KMW
Krabbendijke;KBD
Krommenie−Assendelft;KMA
Kropswolde;KW
Kruiningen−Yerseke;KRG
Lage Zwaluwe;ZLW
Landgraaf;LG
Leerdam;LDM
Leeuwarden;LW
Leeuwarden Camminghaburen;LWC
Leiden Centraal;LEDN
Leiden Lammenschans;LDL

Leidschendam−Voorburg;LDV
Leidschenveen;LVN
Leidschenveen Forepark;LVF
Lelystad Centrum;LLS
Lichtenvoorde−Groenlo;LTV
Lochem;LC
Loppersum;LP
Lunteren;LTN
Maarn;MRN
Maarssen;MAS
Maassluis;MSS
Maassluis West;MSW
Maastricht;MT
Maastricht Randwyck;MTR
Mantgum;MG
Mariënberg;MRB
Martenshoek;MTH
Meerssen;MES
Meppel;MP
Middelburg;MDB
Molenhoek;MHK
Naarden−Bussum;NDB
Nieuw Amsterdam;NA
Nieuw Vennep;NVP
Nieuwerkerk a/d IJssel;NWK
Nieuweschans;NSCH
Nieuweschans Grens;NSCG
Nijkerk;NKK
Nijmegen;NM
Nijmegen Dukenburg;NMD
Nijmegen Heyendaal;NMH
Nijmegen Lent;NML
Nijverdal;NVD
Nunspeet;NS
Nuth;NH
Obdam;OBD
Oisterwijk;OT
Oldenzaal;ODZ
Oldenzaal Grens;ODZG
Olst;OST
Ommen;OMN
Oosterbeek;OTB
Opheusden;OP
Oss;O
Oss West;OW
Oudenbosch;ODB
Overveen;OVN
Pijnacker;PNK
Pijnacker Zuid;PNKZ
Purmerend;PMR
Purmerend Overwhere;PMO
Purmerend Weidevenne;PMW
Putten;PT
Raalte;RAT
Ravenstein;RVS
Reuver;RV
Rheden;RH
Rhenen;RHN
Rijssen;RSN
Rijswijk;RSW
Rilland−Bath;RB
Roermond;RM
Roodeschool;RD
Roosendaal;RSD
Roosendaal Grens;RSDG
Rosmalen;RS
Rotterdam Alexander;RTA
Rotterdam Blaak;RTB
Rotterdam Centraal;RTD
Rotterdam Kleiweg;RTKW
Rotterdam Lombardijen;RLB
Rotterdam Noord;RTN
Rotterdam Wilgenplas;RTWP
Rotterdam Zuid;RTZ
Ruurlo;RL
Santpoort Noord;SPTN
Santpoort Zuid;SPTZ
Sappemeer Oost;SPM
Sassenheim;SSH
Sauwerd;SWD
Schagen;SGN
Scheemda;SDA
Schiedam Centrum;SDM
Schiedam Nieuwland;NWL
Schin op Geul;SOG
Schinnen;SN
Schiphol;SHL
Sittard;STD
Sliedrecht;SDT
Sneek;SK
Sneek Noord;SKND
Soest;ST
Soest Zuid;STZ
Soestdijk;SD
Spaubeek;SBK
Stavoren;STV
Stedum;STM
Steenwijk;SWK
Susteren;SRN
Swalmen;SM
Tegelen;TG
Terborg;TBG
Tiel;TL
Tiel Passewaaij;TPSW
Tilburg;TB
Tilburg Reeshof;TBR
Tilburg West;TBWT
Twello;TWL
Uitgeest;UTG
Uithuizen;UHZ
Uithuizermeeden;UHM
Usquert;UST
Utrecht Centraal;UT
Utrecht Lunetten;UTL
Utrecht Overvecht;UTO
Utrecht Terwijde;UTT
Utrecht Zuilen;UTZL
Valkenburg;VK
Varsseveld;VSV
Veenendaal Centrum;VNDC
Veenendaal West;VNDW
Veenendaal−de Klomp;KLP

```
Veenwouden;VWD
Velp;VP
Venlo;VL
Venlo Grens;VLGR
Venray;VRY
Vierlingsbeek;VLB
Vlaardingen Centrum;VDG
Vlaardingen Oost;VDO
Vlaardingen West;VDW
Vleuten;VTN
Vlissingen;VS
Vlissingen Souburg;VSS
Voerendaal;VDL
Voorburg;VB
Voorburg 't Loo;VBL
Voorhout;VH
Voorschoten;VST
Voorst-Empe;VEM
Vorden;VD
Vriezenveen;VZ
Vroomshoop;VHP
Vught;VG
Waddinxveen;WAD
Waddinxveen Noord;WADN
Warffum;WFM
Weert;WT
Weesp;WP
Wehl;WL
Westervoort;WVT
Wezep;WZ
Wierden;WDN
Wijchen;WC
Wijhe;WH
Winschoten;WS
Winsum;WSM
Winterswijk;WW
Winterswijk West;WWW
Woerden;WD
Wolfheze;WF
Wolvega;WV
Workum;WK
Wormerveer;WM
Zaandam;ZD
Zaandam Kogerveld;ZDK
Zaltbommel;ZBM
Zandvoort aan Zee;ZVT
Zetten-Andelst;ZA
Zevenaar;ZV
Zevenaar Grens;ZVG
Zevenbergen;ZVB
Zoetermeer;ZTM
Zoetermeer Centrum West;ZCW
Zoetermeer Oost;ZTMO
Zoetermeer Voorweg;ZTMV
Zuidbroek;ZB
Zuidhorn;ZH
Zutphen;ZP
Zwaagwesteinde;ZWW
Zwijndrecht;ZWD
Zwolle;ZL
```

A.2 Made up time table source file

```
// TreinNr;AVK;Spoor;Dienstregelpunt;Tijd
1;D;1;ASD;1200
1;A;1;ASA;1210
1;D;1;ASA;1211
1;A;1;HVS;1220
1;D;1;HVS;1221
1;A;1;Ut;1230

2;D;1;Ut;1200
2;A;1;HVS;1210
2;D;1;HVS;1211
2;A;1;ASA;1220
2;D;1;ASA;1221
2;A;1;ASD;1230

3;D;1;ASD;1230
3;A;1;ASA;1240
3;D;1;ASA;1241
3;A;1;HVS;1250
3;D;1;HVS;1251
3;A;1;Ut;1300
```