Alcom-FT Technical Report Series
ALCOMFT-TR-01-176

Time-dependent networks as models to achieve fast exact time-table queries

Gerth Stølting Brodal* Riko Jacob*

5th September 2001

Abstract

We consider efficient algorithms for exact time-table queries, i.e. algorithms that find optimal itineraries. We propose to use time-dependent networks as a model and show advantages of this approach over space-time networks as models.

1 Introduction

Finding the optimal itinerary for a traveler using public transportation is an algorithmic challenge. Using standard algorithms to solve itinerary queries on timetables of interesting detail and size easily leads to unacceptably slow answers, even on modern computers. In practice this problem is overcome by using a heuristic solver, that is a solver that quickly computes results, that are not guaranteed to be optimal. Our focus here is instead to produce optimal itineraries.

The nature of an optimal itinerary query is that of a shortest path question. The natural approach to solve it is to transform the timetable into some weighted graph (modeling as a space-time graph [PS97, PS98]), on which a shortest path corresponds to an optimal itinerary. This seems to be a standard approach, taken for example in [SWW99, MHW01]. In this setting we propose to use as a model time-dependent graphs instead of weighted graphs. This kind of model is closer to the time-table itself and allows the algorithm to immediately disregard large portions of the time-table.

The key idea in a time-dependent network is that the time-delay of a link depends on the point in time the link is used. This model is natural in several other situations, for example data-packets on the Internet or cars on a road network. This models the phenomenon, that the delay a link induces depends on the path that is used to reach this link. In general shortest path questions on such networks are hard to answer, but important special cases allow for

*BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: {gerth,rolf,rjacob}@brics.dk. Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

fast algorithms. These kind of networks have been considered in the literature, see [OR90, OR91] for a survey.

In [SWW99] the authors address the question, whether it is feasible to compute optimal itineraries for the German railway system. They propose several heuristic running time improvements that allow for sufficiently quick answers. Our modeling is compatible to these heuristics, they can immediately be applied in the time-dependent algorithms as well. We expect that our modeling and algorithmic approach will lead to significant running time improvements that allow for a bigger or more detailed network (including local busses, extending to all of Europe). For our lack of real world data the analysis of our approach is a theoretical comparison against [SWW99].

We also consider an extensions of the algorithmic problem, that allows to restrict transfer.

Some simple calculation about problem sizes and modern hardware shows that it can be feasible to precompute the answers to all possible queries and store the result on a hard-disk. If the network size allows this, it can be a viable alternative to an algorithmic solution. This should be the case if the network is not too big. Of course this approach still needs to find fast itinerary in a preprocessing step.

1.1 The algorithmic problem

As already stated, we want to be able to produce optimal itineraries for passengers. That is, given the time-tables of all trains in the country, we want to pre-process this data in a way, that allows us to answer queries fast. We consider queries of the form "given that the traveler is at station A at time t, when is the earliest time he can be at station B, and how can this be achieved?" For simplicity we assume that changing trains has no induced costs, i.e. takes no time and can be done as often as necessary.

We assume that we have a set $\mathbb{T}$ that represents time. We only assume that we have a total ordering of the elements in $\mathbb{T}$. Sometimes we additionally assume that ther is an addition operation defined over $\mathbb{T}$.

A *time-table* T is a set of train-connections. It is valid for a certain time-period, its (time-)*horizon*. This is an interval of $\mathbb{T}$. An *event* is a pair (a, t) where $a \in S$ for some finite set S of stations, and $t \in \mathbb{T}$. A train-connection is a pair (e_1, e_2), $e_1 = (a_1, t_1)$ the departure event and $e_2 = (a_2, t_2)$ the arrival event. We have that the time of the arrival event is not before the time of the departure event, i.e. $t_1 \leq t_2$.

An *itinerary* in T is a sequence of events, more precisely an alternating sequence of train departure and arrival events $e_1, e_2, \ldots, e_k$. Two events e_{2i-1}, e_{2i} have to be consecutive departure and arrival events of some train in T, two events $e_{2i}, e2i + 1$ have to be arrival and departure events at the same station.

It will be common that T is not given explicitly, but for example as the time-table for one day and the additional information that this day is repeated throughout a certain period. Note that in a repetitive schedule there will be over-night connections that are not possible if the horizon is restricted to one day.

These definitions have some severe restrictions. At a station we can not distinguish between continuing in the same train and transfer. We also disallow the traveler to use any other means of transportation like walking 200 meters from one station to another. In Section 5 we will address these issues, allowing us to impose restrictions on the maximum number of transfers or accounting for the time it takes to walk inside the station as part of a transfer.

2 Using a space-time network

The approach we describe is straightforward given how we made the algorithmic problem precise. It is a direct transformation of a time-table into a directed weighted graph. Paths in this graph are almost itineraries. To calculate the weights it is important that $\mathbb{T}$ allows for addition and subtraction.

Given a time-table T we construct the graph $G = (V, E)$ in the following way: V is the set of all events of T. There is an edge from event e at station a to the first event at station a after e. If the schedule is repetitive, this can be reflected by having a link from the last event of the day at station a to the first event of the day at station a. If a pair of events (e_1, e_2) is in T, there is a directed edge from e_1 to e_2. The weight of an edge in the weighted graph model is given by the time-difference of its endpoints, i.e. the traveling time or the waiting time at a station.

This type of model can be seen as a space-time network as described in [PS97]. It is the model used in [SWW99] and therefore important for us because we use it for a running time comparison we will present in Section 4.

In [SWW99] this model is used and augmented with different heuristics that improve the running time. These heuristics immediately transfer to the time-dependent approach, and achieve running times that allow exact time-table queries for the German railway system in a reasonable time. As we can strictly improve over this model this do-ability statement remains valid. The detailed comparison of the two approaches is given in Section 4. We have also reason to believe that our approach scales better, so we would expect it to be fast enough on even bigger or more detailed networks, e.g. including local buses and train as well and/or being precise about changing trains.

In [SWW99] the authors observe that "CPU time looks linear to the number of nodes (and thereby edges) explored." This statement can be explained by cache behavior. If we assume that the priority queue stays small enough to fit into the CPU-cache, whereas the graph is to big for that, we get that the CPU-times are dominated by the cache-misses that correspond to accessing the graph.

Be aware that there are some obvious alternatives to this model. For example, one could deviate from the cycle of edges at one station and explicitly have an edge if a passenger can actually change trains as suggested by a combination of arrival/departure events. This model uses in general a lot more edges, but it allows for a much more detailed modeling. We could for example make sure that the walking inside the station actually allows the transfer.

3 Time-dependent networks

In this section we develop the terminology for time-dependent networks (or graphs, which is precisely the same) and proof the correctness of a fastest path algorithm, that works in the situation we have here. We omit a discussion of the feasibility of shortest path questions in the more general setting, allowing negative delays and/or non-monotonic functions. In this more involved setting, it is important to specify a waiting policy. See [OR90, OR91] for a discussion of this type of network.

The domain of time is again a linearly ordered set $\mathbb{T}$. In this section we do not assume that there is an addition operation defined on $\mathbb{T}$. Typical examples for the set $\mathbb{T}$ are the real numbers and the integers, but also finite sets might be interesting.

A time-dependent network is a directed graph $G = (V, E)$, where every edge e has an associated *link-traversal function* $f_e : \mathbb{T} \to \mathbb{T}$.

Let $f : \mathbb{T} \to \mathbb{T}$ be a function. If f satisfies $f(t) \geq t$ for all t, we say that f has *non-negative delay*. If for $t \leq t'$ we have $f(t) \leq f(t')$, we say that f is *monotonic*.

In general it is important to be precise about the waiting policy for a traveler in such a network. This is not an issue here, as we are only interested in shortest/fastest path in the network and encode all the necessity and possibility to wait into the functions. More precisely, if we consider only monotonic functions waiting never pays off, the imposed restrictions on waiting is irrelevant. The more general setting and the influence of different waiting policies is discussed in [OR90, OR91].

Here we use the following simple definition of a timed path in G: A sequence $v_1, v_2, \ldots, v_k \in V$ of nodes in G and a sequence $t_1, t_2, \ldots, t_k \in \mathbb{T}$ of times form a *timed path* if $e = (v_i, v_{i+1}) \in E$ is an edge of G and $t_{i+1} = f_e(t_i)$. The vertex v_1 is called the departure location or source of the path, t_1 the departure time, v_k the destination and t_k the arrival time.

The *earliest arrival* question for a source node s, a destination d, and a departure time t asks for a timed path p from s to d of which the arrival time is minimal. Similarly the *latest departure* question is well defined if we fix an upper bound for the arrival time, i.e. we ask for a path that has the latest departure time at s under the constraint that the arrival time is not after the specified arrival time.

Assume that we can (efficiently) evaluate the link-traversal functions of a graph and (efficiently) perform comparisons of elements in $\mathbb{T}$. Then it is reasonable to consider a variant of Dijkstra's algorithm to solve the earliest arrival question. It is natural to analyze its running time as a number of function evaluations, comparisons and operations of a (comparison based) priority queue over $\mathbb{T}$.

Lemma 3.1 (All earliest arrival paths are simple) *Let G be a finite time-dependent network with monotonic and non-negative delay edge traversal functions. Let p be a timed path in G departing at time t at node s and arriving at time t' at node d. Then there exists a simple timed path in G departing at time t at node s and arriving at time t'' at node d such that $t'' \leq t'$.*

Corollary 3.2 (Earliest arrival path are well defined) *Let G be a finite time-dependent network with monotonic and non-negative delay edge traversal functions. Let s and d be two nodes of G, and a departing time $t \in \mathbb{T}$. Then there exists $t' \in \mathbb{T}$ achieving the minimum arrival time at d over all timed paths from s to d.*

Label setting algorithm

Let Q be a priority queue over pairs from $V \times \mathbb{T}$, that allows to extract a pair with a minimal element from $\mathbb{T}$, i.e. a priority queue of points in time annotated with nodes.

Static Input: $G = (V, E)$, and for every $e \in E$ a monotonic, non-negative delay link-traversal function $f_e \colon \mathbb{T} \to \mathbb{T}$.
Dynamic Input: source node $s \in V$, destination node $d \in V$, departure time $t \in \mathbb{T}$
$a[v] := \infty$ for all $v \in V$
$a[s] := t$
$S := \emptyset$
$Q := \{(s,t)\}$
while $Q \neq \emptyset$ **do**
(u, t''):= extract_min(Q)
$S := S \cup \{u\}$
break while-loop if $u = d$
for each edge $(u,v) \in E$ such that $v \notin S$ **do**
$t' := f_{uv}(a[u])$
if $a[v] > t'$ **then**
if $a[v] = \infty$ **then**
insert(Q, v, t')
else
update(Q, v, t')
end if
$a[v] := t'$
pred$[v] := u$
end if
end for
end while
$v := d$
$p := (v, a[v])$
while $v \neq s$ **do**
$v :=$ pred$[v]$
$p := (v, a[v]).p$
end while
Output: p

Correctness of the algorithm

The above algorithm is correct, really for the same reason as Dijkstra's algorithm is correct for non-negatively weighted graphs. But as we did not find a proof of correctness that would immediately apply in our situation, we give a complete proof.

Define $\delta(s,t,u)$ to be the earliest possible arrival time at node u given that we departed at time t at node s, i.e. the minimum over all paths from s to u departing at time t.

The correctness of the algorithm follows from the fact that the invariant $a[u] = \delta(s,t,u)$ for all nodes $u \in S$ holds before (and after) every execution of the body of the while loop.

The way we update the estimates, we maintain the invariant that

- pred$[\cdot]$ defines a tree R in G that is directed towards s
- Let p be a path that traverses only edges of R in the reverse direction.

Then $a[\cdot]$ gives the timestamps that form together with p a timed path departing at s at time t. For every node $u \in R$ such a path exists.

- $a[u] \geq \delta(s,t,u)$
- For all $u \in S$ we have $a[u] = \delta(s,t,u)$

Now assume that we have $e = a[u] > \delta(s,t,u) = \delta$ for some node $u \in S$. Without loss of generality we can assume that u is the first (in the execution of the algorithm) such node, i.e. that for all other nodes v in S we have $a[v] = \delta(s,t,v)$.

In this situation let p be some some earliest arrival path from s at time t to u. Let $p(v)$ denote the time path p visits node v. Let y be the first vertex of p that is outside of S and x its predecessor on p (inside S). By assumption we have $a[x] = \delta(s,t,x)$ and $\delta(s,t,x) \leq p(x)$ follows as p induces a subpath from s to x that departs at time t and arrives at time $p(x)$. When x was inserted into S the algorithm updated $a[y]$, and as no update increases an estimate, we have $a[y] \leq f_{xy}(\delta(s,t,x))$. As f_{xy} is monotonic, we can conclude that $f_{xy}(\delta(s,t,x)) \leq f_{xy}(p(x)) = p(y)$. As all link-traversal functions on the remaining path from y to u have non-negative delay, we have $p(y) \leq p(u) = \delta$. Putting these inequalities together, we get $a[y] < a[u]$, which is a contradiction in the case $u = y$. In all other cases, the algorithm would have chosen node y instead of node u as the next node to add to S, which contradicts the definition of u.

This argument also shows that we do not need to check $v \in S$ at Line 13, because we will never find a node inside S where we can still improve $a[u]$. This is the only use of the set S. Depending on how expensive it is to evaluate a link-traversal function, it might or might not pay off to not maintain the set S in the algorithm.

3.1 Modeling a time-table lookup with a time-dependent network

Let T be a time-table. We describe how to construct a time-dependent network $G = (V,E,f)$. Then we chose the set of nodes V to be the set of stations appearing in T. Let u and v be two stations. Define

$$C_{uv} = \{(t,t') \mid ((a,t),(b,t') \in T\}$$

that is for every direct connection from u to v there is a pair of times in C_{uv}. If C_{uv} is empty, there is no arc from u to v. Otherwise we define

$$f_{uv}(t) = \min\{t'' \mid (t',t'') \in C_{uv} \text{ and } t \leq t'\}.$$

This function is monotonic and of non-negative delay.

Let I be an itinerary. Then we find a timed path p in G that is never later at a certain station than I is. The path p is therefore a timed path that does not arrive later at the destination than I does.

Let p be a timed path in G. By the definition of the link traversal functions we know, that every link traversal of p corresponds to an entry of T. Concatenating those entries yields a valid itinerary.

All in all we have that we can answer an itinerary query from s at time t to d on T by solving the earliest arrival problem on G with source s at time t and destination d.

4 Implementation issues

There are several speed-up techniques known for Dijkstra's algorithm. Two classes of such techniques are on-the-fly potential and pruning, as for example used in [SWW99].

The most well known potential technique is to modify the value that is used in the priority queue by a lower bound on the remaining path-length to the destination. If the network is embedded into the plane, this can be an appropriate multiplicative of the Euclidean distance to the destination. One way to convince oneself about the correctness of this method is to think of modifying the edge weights of the graph according to some potential, i.e. $w'_{uv} = w_{uv} + p(v) - p(u)$. Adding any potential to a graph does not change the relative weight of paths in the graph – if path p is longer than path q in G, then path p is also longer than path q on G', the graph with modified weights. Adding a potential that does not create negative weights does not influence the correctness of Dijkstra's algorithm. If we cleverly choose the potential, we can improve the running time of the algorithm significantly. If we by chance manage to use the shortest path distances to d as the potential p, we get that precisely all edges that are on some shortest path towards d have weight 0. If the shortest path happens to be unique, the algorithm only looks at the nodes on the shortest path and the outgoing edges from these nodes. This situation is of course not what we expect to find, if we already have the shortest path potential we do not even need Dijkstra's algorithm to determine shortest paths. But if we have a good, conservative approximation of such a potential we can hope to significantly reduce the part of the graph that is inspected by the algorithm. Of course this is only useful if this potential can easily be computed. This is for example true for the Euclidean distance to the destination.

Another, more direct way to reduce the size of the inspected part of the graph is to remove edges and nodes that are not relevant, i.e. not on a shortest path for the current s and d. In the extreme, again, we might have precomputed a shortest path for all possible choices of s and d, and stored the resulting path. Now we can on the fly disregard all the edges that are not on the current path. The algorithm will then only explore the path itself. Again, in this situation there is no need to run an algorithm anymore. The interesting versions of this method are those where we do not need a lot of memory to store the result of the precomputation, and where evaluating whether a link can be disregarded is easy. A good example is the pruning technique used in [SWW99]. There we have for every edge some geometric information for which kind of s and d the edge is relevant.

In [SWW99] the point is made that a combination of these speed-up techniques reduces the running time of Dijkstra's algorithm in a space-time network enough to compute optimal itineraries on a complete German railway time-table on-line. We observe that these speed-up techniques are just as powerful on the time-dependent network, and that using the time-dependent approach can significantly reduce the size of the inspected part of the network and by this the size of the priority queue.

In the following we will analyze the running time of different time-table query algorithms. This is as a comparison the space-time graph based algorithm and then different implementations of the link-traversal functions. The parameters determining the running time are

- The size of the inspected part of the graph in nodes, this is where the algorithm annotates with estimates and tree-edges, that is nodes v in Line 19 of the algorithm.
- The size of the inspected part of the graph in edges. This are memory accesses/function evaluations.
- Size of the priority queue
- Number of extract min operations. This is the number of times we *explore* a node u, i.e. we have that the node is u in Line 9 of the algorithm.

Here we do not consider the time the algorithm spends maintaining the priority queue. There are several results in the literature discussing how to obtain fast priority queues in this context, for example using the fact that edge weights are integers. This kind of priority queue can be used for the time-dependent networks as well, if we restrict time to be integers. We note that even a more substantial change to the shortest path algorithm as proposed in [Gol01] can easily incorporated into the time-dependent algorithm.

We consider the time-table query from s at time t to d. Let t' be the arrival time at d of a fastest itinerary.

4.1 The space-time network

The first observation we make is, that we never really update the $a[v]$ for any node v. This is because the node has a time build in, we reach it equally fast on all possible paths. This can be exploited to speed up the priority queue. Consider a station a. The algorithm explores all events at a, starting from the earliest arrival time at a up to t'. How many events these actually are heavily depends on the station. But we can see that for a long itinerary, i.e. for big t', there is the possibility of futile work: Assume s itself is a busy station that has hourly non-stop connections to 10 cities. Assume t' is 4 hours after t. Then we usefully explore the events for the 10 connections in the first hour after t. But afterwards we continue exploring nodes at s where the updated node on the other side of the arc is already known to be reachable (but presumably not yet in Q).

The other algorithms avoid these futile exploration steps at some other cost. This trade-off is what we are investigating in the following.

4.2 General piecewise linear functions

If we take a modular approach, we will implement the link-traversal functions completely independent of each other. That is, we will have a procedure that produces the value of $f_{uv}(t)$. This can easily be achieved with $\log_2(k)$ comparisons if k is the number of connections from u to v, i.e. $k = |C_{uv}|$.

Now we do not unnecessarily explore nodes, but we might consider arcs that in the space-time network were not considered, because the departure event is after t'. Additionally we waste time by doing independent searches for the different outgoing arcs.

Note that this approach can easily handle large time-horizons when the schedule is basically repetitive, but has many exceptions arbitrarily spread over the timetable. The important feature of this situation is that the link-traversal functions have a small (in terms of space) representation, that still allows for fast evaluation.

4.3 Combining searches

To avoid the repetition of searches in the for-loop of Line 11–22, one can combine all the events into one data-structure. More precisely we have at every node a balanced search tree that has as leafs the times of all outgoing events. In a preprocessing step we annotate these leafs with a list of outgoing events, for every arc the next in time. Then every exploration of one node costs one binary search plus the update operation on the other side of the arcs.

This approach still performs one search per node and uses additional space, at one node we have a multiplicative blow-up of its out-degree.

4.4 Avoiding the space-overhead

Let d be the out-degree of the node in the time-dependent network corresponding to a. To avoid the space-overhead, we can put all the outgoing events of a station a into one array A sorted by their departure time. We can do this in the following way: we place d such events (so called primary entries), then we leave d empty spaces, then the next d events and so on. Let t'' be the time of the last event before some empty spaces. Then we put for every outgoing direction the next event into the so far empty entries of A (secondary entries). We put all primary entries into one balanced search tree.

When we explore node a, we find the next primary entry of A and scan the next $2d$ entries of A. Then we are sure to have all the next outgoing events in all outgoing directions.

This approach uses asymptotically the same space as a space-time graph. It still performs one search per exploration of a node.

4.5 Avoiding the search

If we watch the algorithm we see that when we perform an update operation, we have our hands on an event-pair e of the time-table. In particular we have one particular arrival time t'' at one particular node a. In addition to e we can store a pointer into the array of outgoing events at a, namely to the first primary element after time t''. If we make this pointer part of the data structure stored in the priority queue, there is no need for a search to explore a node.

4.6 Avoiding to consider connections that depart after t'

There is still one aspect in which the space-time graph might be superior to the time-dependent network approach: It will never consider a link whose departure time is after the arrival time t' at the destination station.

We can also achieve this in the time-dependent network. Assume we did not do the space-saving, that is we have a list of outgoing events for every relevant time at a node. We slightly change this list such that it is no longer ordered by the time of the departure event, but by the time of the arrival event on the other side of the arc. Instead of inserting all the arrival events of the list into the priority queue at once, we only insert the first event and a pointer to the next event of the list. Only when we extract this first event, we insert the second event of the list and so on. With this we inspect less entries of the time-table than the space-time graph algorithm does, we asymptotically never perform more work than that algorithm.

4.7 Cache behavior

Even so the last presented algorithm is the fastest algorithm if we consider asymptotic worst-case running time on the unit cost RAM, it might not be the fastest in practice. One thing is that the possibility to save constant factors in the running time is hard to foresee and an important factor in actual running time. More importantly we should also consider that the running time on a real machine can be heavily influenced by cache-faults. In this respect the last algorithm might not really be a good idea: assuming that the priority queue is small enough to fit into cache, it voluntarily jumps around in memory.

The actual cache behavior seems to be too hard to predict to really make a statement about what should be the fastest algorithm without performing experiments. We leave this with the statement that there should be some possibilities to adapt the above algorithms to a specific cache size.

Of course there is also the priority queue to be considered in tuning the algorithm. This might not be an issue if the time-table happens to be such, that the priority queue is always very small and the overall running time is dominated by the cache misses stemming from accessing the network.

4.8 Concluding remarks on algorithms

Instead of using an array with primary and secondary entries we can consider the list of next events as a persistent linked list that (in the preprocessing) changes as time progresses. See [DSST86] for details on this idea. This construction gives the same asymptotic behavior. It has the additional advantage that it can easily carried over to the exploration saving idea of Section 4.6.

We can see all the algorithms presented here as some clever way to on-demand create the important part of some space-time graph. Again this is only a different point of view.

Note that we can solve as well latest departure questions for a given arrival time. For this we just invert the direction of time and departing and arriving in the timetable.

All the algorithms presented here can additionally perform the online network pruning techniques suggested in [SWW99]. The theoretical analysis carried out here suggests, that the algorithms presented here should be faster, on realistic networks presumably significantly faster, than the explicit space-time graph algorithm used in [SWW99]. The statement that these algorithms are fast enough to produce optimal answers for time-tables of interesting size, should therefore carry over to even bigger (or more detailed) networks, if we use the algorithms presented here. Of course only experiments with real world data can give a conclusive answer whether or not these statements really hold.

5 Extensions

5.1 More realistic transfer

The algorithmic problem discussed so far made the unrealistic assumption, that transfer between trains is basically disregarded. It seems, that we do not get a more realistic modeling of transfer for free. For a start we try to include some time for the walking inside the station in the itinerary, extending the network by as little as we can. We consider every platform to be a station of its own right. Then we connect the platforms by walking links, either following the geometry of the station or with a star. The link traversal functions for the walking links have constant delay, i.e. they are of the form $f(t) = t + c$ for some constant c. (This assumes that we have addition for our set $\mathbb{T}$.)

5.2 Limiting the number of transfers in an itinerary

The above modeling of train changes can also be used to count the number of changes. Without extending the network further we do not consider a change of trains, if it does not involve moving from one platform to another. If we want to be more precise, we can introduce virtual platforms, one for every train-line. Then we do not capture changing of trains within a train line, which is no restriction, as it can always be avoided. Note that we basically return to the explicit network if every train has to be considered as a train line of its own.

To find all Pareto-optimal solutions, i.e. for every bound on the number of transfers the earliest arrival itinerary, we do the following:

First we find the fastest path without an upper bound on the number of changes. This will have some number k of transfers, and is the solution if the bound on the number of transfers is $\geq k$. It is left to find the earliest arrival itinerary under the restriction, that we have at most $k-1, k-2, \ldots, 0$ transfers. We take k copies (levels) of the original network with the twist that all walking links (i.e. all links that stand for a transfer) connect level i with level $i+1$. On this network we search for a path from s at level 1 to some copy of node d. If we find this, we found the fastest path with at most $k-1$ transfers, the level l of the endpoint of the path tells us how many transfers we actually have. Note that it might be that the fastest connection uses 5 transfers, but insisting on 4 transfers is slower than doing only 2 transfers. Then we delete all levels above and including l and continue our search to some copy of d at the remaining levels. We continue until we found a solution without transfer or we found that we can not reach the destination with less then l transfers. In practice we might want to stop this process as soon as insisting on few transfers results in unreasonably long travel times.

5.3 Connection to regular expressions

The above procedure can be seen as searching for a fastest path under a regular expression constrained. If we think of arcs in the network that stand for using a train being labeled with t and transfer links being labeled with w, the constraining regular expression is $(w^*t^*w^*)^k$. It is easily possible to compute earliest arrival questions under general regular expression constraints (we only need to construct a nondeterministic finite automaton and build the cross product with the network). This might be a powerful tool express different kinds of restrictions on the itinerary. See [BJM00] for a discussion on the complexity of imposing formal language constraints onto (shortest) paths questions.

5.4 Concluding on extensions

The extensions mentioned here are also applicable if the time-table is modeled using a space-time graph. The point we want to make here is that using a time-dependent network as a model actually allows for an easy description.

References

[BJM00] Chris Barrett, Riko Jacob, and Madhav Marathe. Formal language constrained path problems. *SIAM J. Comput.*, 30(3):809–837, 2000.

[DSST86] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 109–121, 1986.

[Gol01] Andrew V. Goldberg. A simple shortest path algorithm with linear average time. In *Algorithms - ESA 2001*, volume LNCS 2161, pages 230–241, 2001.

[MHW01] Matthias Müller-Hannemann and Karsten Weihe. Pareto shortest paths is often feasible in practice. In *Algorithm Engineering, WAE 2001*, volume LNCS 2141, pages 185–197, 2001.

[OR90] Ariel Orda and Raphael Rom. Shortest-path and minimum delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3):607–625, 1990.

[OR91] Ariel Orda and Raphael Rom. Minimum weight paths in time-dependent networks. *Networks: An International Journal*, 21, 1991.

[PS97] Stefano Pallottino and Maria Grazia Scutellà. Shortest path algorithms in transportation models: classical and innovative aspects. Technical Report TR-97-06, University of Pisa, 14, 1997.

[PS98] Stefano Pallottino and Maria Grazia Scutellà. Shortest path algorithms in transportation models: classical and innovative aspects. In P. Marcotte and S. Nguyen, editors, *Equilibrium and Advanced Transportation Modelling*, pages 245–281. Kluwer Academic Publishers, 1998.

[SWW99] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. In *Algorithm Engineering*, volume LNCS 1668, pages 110–123, 1999.