Efficient On-Trip Timetable Information in the Presence of Delays

Lennart Frede[1], Matthias Müller–Hannemann[2] and Mathias Schnee[1]

[1]Darmstadt University of Technology, Computer Science,
Hochschulstraße 10, 64289 Darmstadt, Germany
{frede,schnee}@algo.informatik.tu-darmstadt.de
[2]Martin-Luther-University Halle, Computer Science,
Von-Seckendorff-Platz 1, 06120 Halle, Germany
muellerh@informatik.uni-halle.de

Abstract. The search for train connections in state-of-the-art commercial timetable information systems is based on a static schedule. Unfortunately, public transportation systems suffer from delays for various reasons. Thus, dynamic changes of the planned schedule have to be taken into account. A system that has access to delay information of trains (and uses this information within search queries) can provide valid alternatives in case a train change breaks. Additionally, it can be used to actively guide passengers as these alternatives may be presented before the passenger is already stranded at a station due to a broken transfer. In this work we present an approach which takes a stream of delay information and schedule changes on short notice (partial train cancellations, extra trains) into account. Primary delays of trains may cause a cascade of so-called secondary delays of other trains which have to wait according to certain waiting policies between connecting trains. We introduce the concept of a dependency graph to efficiently calculate and update all primary and secondary delays. This delay information is then incorporated into a time-expanded search graph which has to be updated dynamically. These update operations are quite complex, but turn out to be not time-critical in a fully realistic scenario. We finally present a case study with data provided by Deutsche Bahn AG showing that this approach has been successfully integrated into our multi-criteria timetable information system MOTIS and can handle massive delay data streams instantly.

Keywords: timetable information system, primary and secondary delays, dependency graph, dynamic graph update

1 Introduction and Motivation

In recent years the performance and quality of service of electronic timetable information systems has increased significantly. Unfortunately, not everything runs smoothly in scheduled traffic and the presence of delays is the norm rather than the exception.

ATMOS 2008
8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems
http://drops.dagstuhl.de/opus/volltexte/2008/1584

Delays can have various causes: Disruptions in the operations flow, accidents, malfunctioning or damaged equipment, construction work, repair work, and extreme weather conditions like snow and ice, floodings, and landslides to name just a few. A system that incorporates up-to-date train status information (most importantly information about future delays based on the current situation) can provide a user with valid timetable information in the presence of disturbances.

Such an on-line system can additionally be utilized to verify the current status of a journey:

- Journeys can either be still valid (i.e., they can be followed as planned),
- can be affected such that the arrival at the destination is delayed,
- or may no longer be possible.

In the latter case a connecting train will be missed, either because the connecting train cannot wait for a delayed train, or the connecting train may have been canceled. In a delay situation, such a status information is very helpful. In the positive case that all planned train changes are still possible, passengers can be reassured that they do not have to worry about potential train misses. To learn that one arrives x minutes late with the planned sequence of trains may allow a customer to make arrangements, e.g. inform someone to pick one up later accordingly. In the unfortunate case that a connecting train will be missed, this information can now be obtained well before the connection breaks and the passenger is stranded at some station. Therefore, valid alternatives may be presented while there are still more possibilities to act. This situation is clearly preferable over missing a connecting train and than going to a service point to request an alternative.

As up to now the commercial systems do not take the current situation into account (although estimated arrival times may be accessible for a given connection, these times are not used actively during the search), their recommendations may be impossible to use, as the proposed alternatives already suffer from delays and may even already be infeasible at the time they are delivered by the system.

Static timetable information systems. The standard approach to model static timetable information is as a shortest path problem in either a time-expanded or time-dependent graph. The recent survey [1] describes the models and suitable algorithms in detail. We developed our timetable information system MOTIS which performs a multi-criteria search for train connections in a realistic environment using a suitably constructed time-expanded graph. Our underlying model ensures that each proposed connection is indeed feasible, i.e. can be used in reality. The criteria considered are travel time, number of interchanges, ticket cost, and reliability of all interchanges of a connection. The system is able to present many attractive alternatives to customers [2].

Our contribution and overview. Previous research on timetable information systems has focused on the static case where the timetable is considered as fixed. Here we start out a new thread of research on dynamically changing timetable

data due to disruptions. We extended our timetable information system MOTIS to use current train status information. Modeling issues have been discussed on a theoretical level but no true to life system with real delay data has been studied in the literature and to our knowledge no such system that guarantees optimal results (with respect to even a single optimization criterion) exists.

We give first results of implementing such a system for a real world scenario with no simplifying assumptions at all. The architecture we propose is intended for a multi-server environment where the availability of search engines has to be guaranteed at all times. Our system consists of two main components, the *dependency graph* and the *search graph*. The dependency graph is used to efficiently propagate primary delay information according to waiting policies. The overall new status information is then incorporated into the search graph which is used for customer search queries. Our dependency graph is similar to a simple time-expanded graph model with distinct nodes for each departure and arrival event of the whole schedule for the current and following days. This is a natural and efficient model since every event has to store its own update information. For the search graph, however, we are free to use either the time-expanded or the time-dependent model. In this paper, we have chosen to use the time-expanded model for the search graph since MOTIS is based on this. Although update operations are quite complex in this model, it will turn out that they can be performed very efficiently, in less than a millisecond per update message on average.

We will also discuss the difference between searches in an *on-trip* scenario, where a passenger is either stranded at a station or in a train whose connecting train will be missed, to classical *pre-trip* searches.

The rest of this paper is organized as follows: In Section 2, we will discuss primary and secondary delays. We introduce our architecture in Section 3 and its two components, the update of the search graph (in Section 4) and the propagation algorithm on our dependency graph model (in Section 5). In Section 6, we present our approach to perform on-trip as opposed to pre-trip searches. Afterwards, we provide our experimental results in Section 7. Finally, we conclude and give an outlook.

Related work. Delling et al. [3] independently of us came up with ideas on how to regard delays in timetabling systems. In contrast to their work we do not primarily work on edge weights, but consider nodes with time stamps. The edge weight for time follows, whereas edge weights for transfers and cost do not change during the update procedures. This is important for the ability to do multi-criteria search.

A related field of current research is disposition and delay management. Gatto et al. [4,5] have studied the complexity of delay management for different scenarios and have developed efficient algorithms for certain special cases using dynamic programming and minimum cut computations. Various waiting policies have been discussed, for example by Ginkel and Schöbel [6]. Schöbel [7] also proposed integer programming models for delay management. Stochastic models

for the propagation of delays are studied, for example, by Meester and Muns [8]. Waiting policies in a stochastic context are treated in [9].

2 Up-To-Date Status Information

2.1 Primary Delay Information

First of all, the input stream of status messages consists of reports that a certain train departed or arrived at some station at time τ either on time or delayed by x minutes. In case of a delay, such a message is followed by further messages about predicted arrival and departure times for all upcoming stations on the train route.

Besides, there can be information about additional special trains (a list of departure and arrival times at stations plus category, attribute and name information). Furthermore, we have (partial) train cancellations, which include a list of departure and arrival times of the canceled stops (either all stops of the train or from some intermediate station to the last station).

Moreover, we have manual decisions by the transport management of the form: "Change from train t to t' will be possible" or "will not be possible". In the first case it is guaranteed that train t' will wait as long as necessary to receive passengers from train t. In the latter case the connection is definitively going to break although the current prediction might still indicate otherwise. This information may depend on local knowledge, e.g. that not enough tracks are available to wait or that additional delays are likely to occur, or may be based on global considerations about the overall traffic flow. We call messages of this type *connection status decisions*.

2.2 Secondary Delays

Secondary delays occur when trains have to wait for other delayed trains. Two simple, but extreme examples for waiting policies are:

- *never wait*
  In this policy, no secondary delays occur at all. This causes many broken connections and in the late evening it may imply that customers do not arrive at their destination on the same travel day. However, nobody will be delayed who is not in a delayed train.
- *always wait as long as necessary*
  In this strategy, there are no broken connections at all, but massive delays are caused for many people, especially for those whose trains wait and have no delay on their own.

Both of these policies seem to be unacceptable in practice. Therefore, train companies usually apply a more sophisticated rule system specifying which trains have to wait for others and for how long. For example, the German railways Deutsche Bahn employ a complex set of rules, dependent on train type and local specifics.

In essence, this works as follows: There is a set of rules describing the maximum amount of time a train t may be delayed to wait for passengers from a feeding train f. Basically, these rules depend on train categories and stations. But there are also more involved rules, like if t is the last train of the day in that direction, the maximum delay time is increased, or during peak hours, when trains operate more frequently, the maximum delay time may be decreased.

The *waiting time* $wt(t, s, f)$ is the maximum delay acceptable for train t at station s waiting for a feeding train f. Let $dep_{sched}(s, t)$ and $dep(s, t)$ be the departure time according to the schedule resp. the new departure time of train t at station s, $arr(s, t)$ the arrival time of a train and $minct(s, f, t)$ the minimum change time needed from train f to train t at station s. Note that in a delayed scenario the change time can be reduced, as guides may be available that show changing passengers the way to their connecting train. If the following equation holds

$$arr(s, f) + minct(s, f, t) - dep_{sched}(s, t) < wt(t, s, f)$$

train t will incur a secondary delay because it waits for f at station s. Its new departure time is determined by the following equation

$$dep(s, t) = \begin{cases} arr(s, f) + minct(s, f, t) & \text{if } t \text{ waits} \\ dep_{sched}(s, t) & \text{otherwise .} \end{cases}$$

In case of several delayed feeding trains, the new departure time will be determined as the maximum over these settings.

During day-to-day operations these rules are always applied automatically. If the required waiting time of a train lies within the bounds defined by the rule set, trains will wait. Otherwise they will not. All exceptions from these rules have to be given as connection status decisions.

3 System Architecture

Our system consists of two main components. One part is responsible for the propagation of delays from the status information and for the calculation of secondary delays, while the other component handles connection queries. The core of the first part is a *dependency graph* which models all the dependencies between different trains and between the stops of the same train (in Section 5 we give more details on that). The obtained information is then sent to the search graph which is updated accordingly. This decoupling of dependency and search graph allows us to use any graph model for the search graph.

In a distributed scenario this architecture can be realized with one server for the dependency graph that continuously receives new status information and broadcasts the update information to a number of servers on which the query algorithms run. Load balancing can schedule the update phases for each server. If this is done in a round robin fashion, the availability of service is guaranteed.

Our design decision to work with two separate components also gives us additional flexibility when to broadcast the update information. In this paper, we

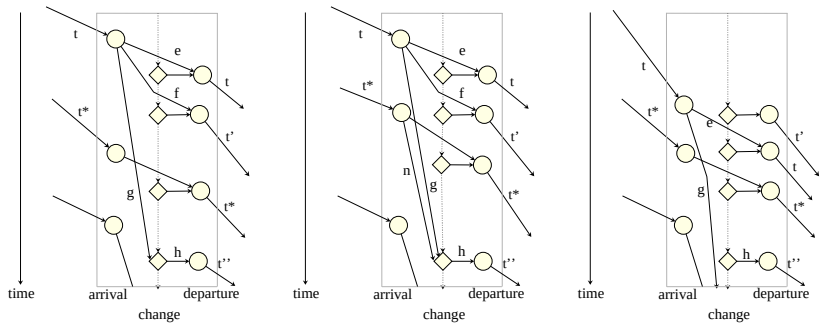

Fig. 1. The change level at a station (left) and changes if train t^* arrives earlier (middle picture) or train t arrives later (right).

broadcast the update information immediately when it becomes available. However, a reasonable alternative is to broadcast a consistent update state only every Δ minutes, for some small Δ. This option may save many update operations in the search graph which, in particular, result from small oscillations in forecasts of trains with frequent comparisons of actual and scheduled times.

4 Updating the Search Graph

Time-Expanded graph model. Let us briefly recall the time-expanded graph model. The basic idea is to introduce a directed search graph where every node corresponds to a specific event (departure, arrival, change of a train) at a station.

A connection served by a train from station A to station B is called *elementary*, if the train does not stop between A and B. Edges between nodes represent either elementary connections, waiting within a station, or changing between two trains. For each optimization criterion, a certain length is associated with each edge.

Traffic days, possible attribute requirements and train class restrictions with respect to a given query can be handled quite easily. We simply mark train edges as *invisible* for the search if they do not meet all requirements of the given query. With respect to this visibility of edges, there is a one-to-one correspondence between feasible connections and paths in the graph.

More details of the graph model can be found in [2].

Modeling interchanges in a time-expanded graph. To model non-constant change times between pairs of trains, additional nodes and edges are required besides the ones for arrival and departure events. In forward search (when the desired departure is specified), for every departure time at a station there is a change node connected via *entering edges* to all departure nodes at that time. The change nodes are interconnected with *waiting edges*. *Leaving edges* link to

the first change node which is reachable in the time needed for a transfer from this train to any other. All possible shorter change times (e.g. for trains at the same platform) are realized using *special transfer edges*. Additionally, we have *stay-in-train edges*. Only entering edges carry the cost of a train change.

In Figure 1 (left) it is possible to change from train t to all trains departing not earlier than t'' using leaving edge g, any number of waiting edges and an entering edge (e.g. h to enter t''). A change to train t' on the same platform is also feasible using special interchange edge f and, of course, to stay in train t via stay-in-train edge e. However, it is impossible to change to train t^* although it departs later than t', because it requires more time to reach it.

Updates The update in the search graph does not simply consist of setting new time stamps for nodes (primary and secondary delays), insertions (additional trains) and deletions (cancellations) of nodes and resorting lists of nodes afterwards. Furthermore, all the edges present to model the changing of trains at the affected stations have to be recomputed respecting the changed time stamps, additional and deleted nodes, and connection status information. The following adjustments are required on the change level (see Figure 1):

- Inserting change nodes or unhooking them from the waiting edges chain at times where a new event is the only one or the only event is moved away or canceled.
- Updating the leaving edges pointing to the first node reachable after a train change.
- Updating the nodes reachable from a change node via entering edges.
- Recalculating special interchange edges from resp. to arrival resp. departure nodes with a changed time stamp (either remove, adjust or insert special interchange edges).

The result of the update phase is a graph that looks and behaves exactly as if it was constructed from a schedule describing the current situation. Additionally it contains information about the original schedule and reasons for the delays.

Next we give two examples for updating the search graph[1]: Suppose train t^* manages to get rid of some previous delay and now arrives and departs earlier than previously predicted (see Figure 1, middle part). In the new situation it is now possible, to change to train t'' using the new leaving edge n and the existing entering edge h.

In our second example let train t arrive delayed as depicted in Figure 1 (right). As it now departs after t', it is not only impossible to change to t' (special interchange edge f is deleted), but also the change departure nodes for the departures of t' and t are in reverse order. Therefore, the waiting edges have to be relinked. Furthermore, a change to t'' is no longer possible, so the leaving edge h points to a node later than the departure of t''.

[1] Note that we increased the station dependent interchange time from the middle to the right extract to make this example work.

5 Dependency Graph

5.1 Graph Model

Our *dependency graph* (see Fig. 2) models the dependencies between different trains and between the stops of the same train. Its node set consists of four types of nodes:

- departure nodes,
- arrival nodes,
- forecast nodes, and
- schedule nodes.

Each node has a time stamp which can dynamically change. Departure and arrival nodes are in one-to-one correspondence with departure and arrival events. Their time-stamps reflect the current situation, i.e. the expected departure or arrival time subject to all delay information known up to this point.

Schedule nodes are marked with the planned time of an arrival or departure event, whereas the time stamps of forecast nodes is the current external prediction for their departure or arrival time.

The nodes are connected by five different types of edges. The purpose of an edge is to model a constraint on the time stamp of its head node. Each edge $e = (v, w)$ has two attributes. One attribute is a Boolean value, signifying whether this edge is currently active or not. The other attribute $\tau(e)$ denotes a point in time which basically can be interpreted as a lower bound on the time stamp of its head node w, provided that the edge is currently active.

- *Schedule edges* connect schedule nodes to departure or arrival nodes. They carry the planned time for the corresponding event of the head node (according to the published schedule). Edges leading to departure nodes are always active, since a train will never depart prior to the published schedule.
- *Forecast edges* connect forecast nodes to departure or arrival nodes. They represent the time stored in the associated forecast node. If no forecast for the node exists, the edge is inactive.
- *Standing edges* connect arrival events at a certain station to the following departure event of the same train.
  They model the condition that the arrival time of train t at station s plus its minimum standing time $stand(s, t)$ must be respected before the train can depart (to allow for boarding and deboarding of passengers). Thus, for a standing edge e, we set $\tau(e) = arr(s, t) + stand(s, t)$. Standing edges are always active.
- *Traveling edges* connect a departure node of some train t at a certain station s to the very next arrival node of this train at station s'. Let $dep(s, t)$ denote the departure time of train t at station s and $tt(s, s', t)$ the travel time for train t between these two stations. Then, for edge $e = (s, s')$, we set $\tau(e) = dep(s, t) + tt(s, s', t)$. These edges are only active if the train currently has a secondary delay (otherwise the schedule or forecast edges provide the necessary conditions for its head node).

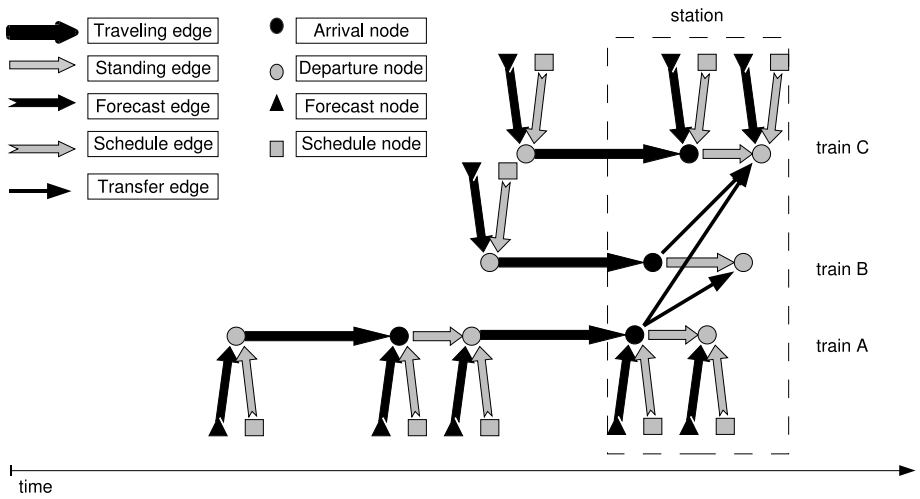

Fig. 2. Illustration of the dependency graph model.

Due to various, mostly unknown factors determining the speed of trains in a delayed scenario, e.g. speed of train, condition of the track, track usage (by other trains and freight trains that are not in the available schedule), used engines with acceleration/deceleration profiles, signals along the track etc. we assume for simplicity that $tt(s, s', t)$ is the time given in the planned schedule.

– *Transfer edges* connect arrival nodes to departure nodes of other trains at the same station, if there is a planned transfer between these trains. Thus, if f is a potential feeder train for train t at station s, we set $\tau(e) = wait(t, s, f)$, where

$$wait(t, s, f) = \begin{cases} arr(s, f) + minct(s, f, t) & \text{if } t \text{ waits for } f \\ 0 & \text{otherwise} \end{cases}$$

(cf. Section 2.2) if we respect the waiting rules. Recall that t waits for f only if the following equation holds

$$arr(s, f) + minct(s, f, t) - dep_{sched}(s, t) < wt(t, s, f)$$

or we have an explicit connection status decision that t will wait.
By default these edges are active. In case of an explicit connection status decision "will not wait" we mark the edge in the dependency graph as not active and ignore it in the computation.
For an "always wait" or "never wait" scenario we may simply always return the resulting delayed departure time or zero, respectively.

5.2 Computation on the Dependency Graph

The current time stamp for each departure or arrival node can now be defined recursively as the maximum over all deciding factors: For a departure of train t at station s with feeders $f_1, \dots f_n$ we have $dep(s,t) =$

$$\max\{dep_{sched}(s,t), dep_{for}(s,t), arr(s,t) + stand(s,t), max_{i=1}^{n}\{wait(t,s,f_i)\}\}.$$

For an arrival we have

$$arr(s,t) = \max\{arr_{sched}(s,t), arr_{for}(s,t), dep(s',t) + tt(s',s,t)\}$$

with the previous stop of train t at station s'. Inactive edges do not contribute to the maximum in the preceeding two equations.

If we have a status message that a train has finally departed or arrived at some given time dep_{fin} resp. arr_{fin}, we do not longer compute the maximum as described above. Instead we use this value for future computations involving this node.

We maintain a priority queue (ordered by increasing time stamps) of all nodes whose time stamps have changed since the last computation was finished. Whenever we have new forecast messages, we update the time stamps of the forecast nodes and, if they have changed, insert them into the queue. As long as the queue is not empty we extract a node from the queue and update the time stamps of the dependent nodes (which have an incoming edge from this node). If the time stamp of a node has changed in this process, we add it to the queue as well.

For each node we keep track of the edge e_{max} which currently determines the maximum so that we do not need to recompute our maxima over all incoming edges every time a time stamp changes. Only if $\tau(e_{max})$ was decreased or $\tau(e)$ for some $e \neq e_{max}$ increases above $\tau(e_{max})$ the maximum has to be recomputed.

- If $\tau(e)$ decreases and $e \neq e_{max}$ nothing needs to be done.
- If $\tau(e)$ increases and $e \neq e_{max}$ but $\tau(e) < \tau(e_{max})$ nothing needs to be done.
- If $\tau(e)$ increases and $e = e_{max}$ the new maximum is again determined by e_{max} and the new value is given by the new $\tau(e_{max})$.

When the queue is empty, all new time stamps have been computed and the nodes with changed time stamps can be sent to the search graph update routine.

6 Search Types

Most timetable information systems consider a pre-trip scenario: The user is at home and requests a connection from station s_1 to s_2 departing or arriving around some time τ or inside an interval $[\tau_1, \tau_2]$. In such a scenario, it is important that the search delivers all attractive connections with respect to several criteria which suit the query. Even if you use information systems at a station or click "Right-now" in an online system you will usually be offered several alternatives.

In an *on-trip* scenario one is much closer to an earliest arrival problem. We differentiate two cases of the on-trip search:

1. A customer is at a certain station and wants to travel right now. Either he comes without a travel plan (for example, he was unable to plan the end of some meeting) or he may have just missed a connecting train.
2. The customer sits already in a train and wants to search for alternatives, for example, because he has been informed that a connecting train will be missed.

In both cases travelers want to reach their destination as fast and convenient as possible. In case of delays many railways even remove restrictions on train-bound tickets, so it might be possible to completely forget about ticket costs, since the ticket is already paid and the passenger may use any means of transportation available. If there is a restriction like "no high speed train" (like the German ICE or French TGV) which is not revoked, an on-trip search with train category restrictions should be supported.

On-trip search at a station. In the example above one would not want to spend too much time at a station to shorten the traveling time measured from the departure with the first used train to the arrival at the destination (as calculated in the pre-trip scenario), instead the total travel time counting from "now" is one of the optimization goals. However, in the presence of delays it may become more important to search for reliable connections.

On-trip search in a train. In case the user currently travels in a train the on-trip search is different from the scenario at a station. Instead of leaving the train and standing at a station with the connecting train long gone (or canceled), we can do much better if we know of this problem in advance. Interesting alternatives may either leave the train before arriving at the station where the connection breaks, or stay longer in the train to change trains at a subsequent station.

Realization. Both on-trip searches can be realized in our timetable information system using different starting events. Instead of creating start labels for all departures in the departure interval (for forward search), we either

- create only a single start label at the change level of the source station and count time including the waiting time before taking any train (on-trip station), or
- create only a single start label at the arrival station of the train edge the traveler uses when receiving the information about a connecting train that will be missed (on-trip train).

Note that in the on-trip train case, using the arrival node of the train instead of any of the departure nodes, the modeling of interchanges in the time expanded graph guarantees that only valid train changes at the first stop after receiving the information are used. It would not be feasible to solve the on-trip train case with

a single departure at that station, because we need to ensure that the departure of the train with which one arrives, all departures below the station dependent change time (through special interchange rules) and all later departures are considered and all but the first case are counted as an additional interchange. Thus quite a lot of different departures with differing values for elapsed time at start and number of interchanges used so far would have to be considered.

7 Evaluation of the Prototype

We implemented the dependency graph and the update algorithm described in Section 5 and extended our time table information system MOTIS to support updating the search graph (cf. Section 4). Although these update operations are quite costly, we give a proof of concept and show that they can be performed sufficiently fast for a system with real-time capabilities.

Our computational study uses the German train schedule of 2008. During each operating day all trains that pass various trigger points (stations and important points on tracks) generate status messages. There are roughly 5000 stations and 1500 additional trigger points. Whenever a train generates a status message on its way, new predictions for the departure and arrival times of all its future stops are computed and fed into a data base. German railways Deutsche Bahn AG provided delay and forecast data from this data base for a number of operation days. The simulation results for these days look rather similar without too much fluctuation neither in the properties of the messages nor in the resulting computational effort. In the following, we present results for a standard operating day with an average delay profile.

To test our system, we used five sets of waiting profiles. Basically, the train categories were divided into five classes: high speed trains, night trains, regional trains, urban trains, and class "all others." Waiting times are then defined between the different classes as follows:

- *standard* High speed trains wait for each other 3 minutes,
  other trains wait for high speed trains, night trains, and trains of class "all others" 5 minutes,
  night trains wait for high speed and other night trains 10 minutes, and 5 minutes for class "all others."
- *small* All times of scenario standard are halved, but night trains do not wait for train class "all others."
- *double* All times of scenario standard are doubled.
- *all5* All times of scenario standard are set to five minutes, in addition regional trains wait 5 minutes for all but urban trains.
- *extreme* All times of the previous scenario are doubled.

It is important to keep in mind that the last two policies are far from reality and are intended to strain the system beyond the limits it was designed to handle. For each of these different waiting profiles we tested different maximum distances of feeding and connecting trains $\delta \in \{5, 15, 30, 45, 60\}$, with one hour

search graph	
event nodes	1.0 mil
change nodes	0.8 mil
edges	2.2 mil

dependency graph	
events	977,324
standing edges	449,575
driving edges	488,662

Table 1. Properties of our search graph (left) and dependency graph (right).

being the periodicity for most types of trains, and compare them to a variant without waiting for different trains (policy *no wait*). In this reference scenario it is still necessary to propagate delays in the dependency graph to correctly update the train runs. Thus the same computations as with waiting policies is carried out, only the terms for feeding trains are always zero.

We constructed a search and dependency graphs from the real schedule consisting of 37,000 trains operating on the selected day. The number of nodes and edges in both graphs are given in Table 1. There is one event node, one schedule node and one forecast node per train event in the dependency graph, the number of forecast and schedule edges equals the number of events, too. The number of standing and traveling edges are in one to one correspondence to the stay-in-train and train edges of the search graph. The number of feeding edges depends on the waiting policy and δ and can be found in the eighth column of Table 2. There is a monotonous growth in the number of transfer edges depending on the parameter δ. Additionally, the number of these edges increase as more trains wait for other trains because of the additional rules for scenarios with more rules.

For the chosen simulation day we have a large stream of real forecast messages. Whenever a complete sequence of messages for a train has arrived, we send them to the dependency graph for processing. 340,495 sequences containing a total of 6,211,207 forecast messages are handled. Of all messages 2,471,582 forecasts are identical to the last forecasts already processed for their nodes. The remaining 3,739,625 messages either trigger computations in the dependency graph or match the current time stamp of the node. The latter require neither shifting of nodes nor a propagation in the dependency graph. The resulting number of node shifts is given in the seventh column of Table 2.

At the end of the day 596,496 nodes have received at least one forecast. For 265,544 nodes the forecast differs from the scheduled time although there are 3,287,834 forecasts differing from the scheduled time for the event. Note that the last number is much higher as trains whose prediction changes produce new messages each time. A train with a large number of stops and a long travel time thus can generate a large number of messages.

In Table 2 we give the results for our test runs for the different policies and values of δ. All experiments were run on a standard PC (AMD Athlon 64 X2 4600+ 2.4 GHz with 4GB of RAM). The key figures for required computations, stations with a delayed event and node shifts increase when changing to policies for which trains wait longer or more trains have to wait. Increasing δ yields a higher effect the more trains wait. The overall small impact of changing δ is due to the majority of delays being rather small. Only for the less realistic scenarios

we notice a significant growth in all key criteria when increasing δ from 5 to 15, whereas all policies behave rather similarly for $\delta = 5$.

Amongst the plausible policies there is only a 11% difference in the number of moved nodes. It nearly doubles going to policy *all5* and even increases by a factor of 3.4 towards policy *extreme*. The increase in running time spent in the search graph is equivalent. Of our simulation time roughly 3 minutes are spent extracting and preprocessing the messages from the forecast stream. This time is obviously independent of the test scenario. Interestingly, the time spent in the dependency graph seems to be only minimally affected by exchanging the profiles against those that incur more computations and node shifts. As the

Instance		computation time for				Number of		Delayed at	
policy	δ	SG	DG	IO	total	shifts	feeding	end of day	
	in min	in s	in s	in s	in s	executed	edges	nodes	stations
no wait	-	807	133	177	1118	3,165,614	0	357,972	5,467
	5	819	136	162	1118	3,253,980	8,792	359,105	5,511
	15	860	137	167	1164	3,416,718	55,218	364,961	5,664
small	30	871	137	163	1171	3,430,189	124,141	365,179	5,664
	45	875	139	157	1171	3,432,189	207,855	365,206	5,664
	60	869	137	161	1167	3,434,041	267,638	365,231	5,664
	5	817	135	170	1122	3,254,013	8,792	359,105	5,511
	15	876	136	169	1180	3,426,272	55,284	365,110	5,711
standard	30	880	139	170	1189	3,445,019	124,305	365,353	5,723
	45	878	138	158	1175	3,454,623	208,127	365,395	5,733
	60	917	148	169	1234	3,460,210	268,002	365,452	5,738
	5	813	133	164	1110	3,265,175	8,792	359,254	5,511
	15	917	137	162	1216	3,557,572	55,284	367,171	5,731
double	30	931	136	171	1238	3,617,603	124,305	367,590	5,770
	45	959	136	160	1255	3,646,080	208,127	367,863	5,782
	60	979	137	161	1277	3,661,137	268,002	367,995	5,787
	5	830	137	178	1,145	3,419,161	16,261	366,372	5,815
	15	1,761	141	174	2,076	6,994,379	168,849	404,336	6,541
all5	30	1,776	146	157	2,078	7,095,897	400,114	405,827	6,557
	45	1,796	148	166	2,110	7,112,681	665,811	406,214	6,561
	60	1,793	150	170	2,113	7,121,351	874,649	406,433	6,561
	5	815	137	173	1,124	3,446,965	16,261	367,303	5,818
	15	3,090	159	175	3,424	12,090,373	168,849	422,119	6,648
extreme	30	3,111	164	178	3,453	12,155,547	400,114	434,040	6,676
	45	3,134	170	177	3,480	12,257,936	665,811	438,645	6,684
	60	3,306	178	179	3,663	12,285,623	874,649	440,233	6,684

Table 2. Computation time (propagation in the dependency graph (DG) and update of the search graph (SG), IO and total) and key figures for the number of feeding edges, node shifts in the search graph and the number of nodes and stations with delay at the end of the day with respect to different waiting policies.

overall running time is by far dominated by the reconstruction work in the search graph we would rather try to improve the performance there, if necessary.

Even for the most extreme scenario a whole day can be simulated in one hour. The overall simulation time for realistic policies lies around 20 minutes. For the policy *standard* with $\delta = 45$ we are below 1/5ms ($189\mu s$) per message, at a rate of less than 75 messages arriving per second. This clearly qualifies for live performance.

8 Conclusions and Future Work

We have built a first prototypal system which can be used for efficient off-line simulation with massive streams of delay and forecast messages for typical days of operation within Germany.

It remains an interesting task to implement a live feed of delay messages for our timetable information system and actually test real-time performance of the resulting system. Since update operations in the time-dependent graph model are somewhat easier than in the time-expanded graph model, we also plan to integrate the update information from our dependency graph into a multi-criteria time-dependent search approach developed in our group (Disser et al. [10]).

Additionally, we would be interested in looking into speed-up techniques for dynamic scenarios in the multi-criteria case. Since most of the existing speed-up techniques focus on single criterion search, time-less edges, and usually require bidirectional search, this is not at all easy in the multi-criteria static scenario without delays. The recent SHARC algorithm [11,12] is a powerful speed-up technique for uni-directional search which seems to be a promising candidate to generalize to a dynamic scenario and multiple search criteria.

Acknowledgments

This work was partially supported by the DFG Focus Program Algorithm Engineering, grant Mu 1482/4-1. We wish to thank Deutsche Bahn AG for providing us timetable data and up-to-date status information for scientific use, and Christoph Blendinger and Wolfgang Sprick for many fruitful discussions.

References

1. Müller-Hannemann, M., Schulz, F., Wagner, D., Zaroliagis, C.: Timetable information: Models and algorithms. In: Algorithmic Methods for Railway Optimization. Volume 4395 of Lecture Notes in Computer Science, Springer Verlag (2007) 67–89
2. Müller-Hannemann, M., Schnee, M.: Finding all attractive train connections by multi-criteria Pareto search. In: Proceedings of the 4th ATMOS workshop. Volume 4359 of Lecture Notes in Computer Science, Springer Verlag (2007) 246–263
3. Delling, D., Giannakopoulou, K., Wagner, D., Zaroliagis, C.: Timetable Information Updating in Case of Delays: Modeling Issues. Technical report, ARRIVAL (2008)

4. Gatto, M., Glaus, B., Jacob, R., Peeters, L., Widmayer, P.: Railway delay management: Exploring its algorithmic complexity. In: Algorithm Theory — SWAT 2004. Volume 3111 of Lecture Notes in Computer Science, Springer (2004) 199–211
5. Gatto, M., Jacob, R., Peeters, L., Schöbel, A.: The computational complexity of delay management. In: Proceedings of the 31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG 05). Volume 3787 of Lecture Notes in Computer Science, Springer (2005) 227–238
6. Ginkel, A., Schöbel, A.: The bicriteria delay management problem. Transportation Science **41** (2007) pp. 527–538
7. Schöbel, A.: Integer programming approaches for solving the delay management problem. In: Algorithmic Methods for Railway Optimization. Volume 4359 of Lecture Notes in Computer Science, Springer (2007) 145–170
8. Meester, L.E., Muns, S.: Stochastic delay propagation in railway networks and phase-type distributions. Transportation Research Part B **41** (2007) 218–230
9. Anderegg, L., Penna, P., Widmayer, P.: Online train disposition: to wait or not to wait? ATMOS'02, ICALP 2002 Satellite Workshop on Algorithmic Methods and Models for Optimization of Railways, Electronic Notes in Theoretical Computer Science **66** (2002)
10. Disser, Y., Müller-Hannemann, M., Schnee, M.: Multi-criteria shortest paths in time-dependent train networks. In: WEA 2008. Volume 5038 of Lecture Notes in Computer Science, Springer (2008) 347–361
11. Bauer, R., Delling, D.: SHARC: Fast and Robust Unidirectional Routing. In: Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08), SIAM (2008) 13–26
12. Delling, D.: Time-Dependent SHARC-routing. In: ESA 2008. Volume 5193 of Lecture Notes in Computer Science, Springer (2008) 332–343