Efficient Timetable Information in the Presence of Delays*

Matthias Müller–Hannemann[1] and Mathias Schnee[2]

[1] Martin-Luther-University Halle, Computer Science,
Von-Seckendorff-Platz 1, 06120 Halle, Germany
muellerh@informatik.uni-halle.de
[2] Darmstadt University of Technology, Computer Science,
Hochschulstraße 10, 64289 Darmstadt, Germany
schnee@algo.informatik.tu-darmstadt.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 about trains (and uses this information within search queries) can provide valid alternatives in case a connection does not work. 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 an invalid 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 policies for delays 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 the 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

* A preliminary version of this paper has appeared in Proceedings of ATMOS 2008 [1].

runs smoothly in scheduled traffic and delays are the norm rather than the exception.

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, floods, and landslides, to name just a few. On a typical day of operation in Germany, an online system has to handle about 6 million forecast messages about (mostly small) changes with respect to the planned schedule and the latest prediction of the current situation. Note that this high number of changes also includes cases where delayed trains catch up some of their delay.

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 executed as planned),
- they can be affected such that the arrival at the destination is delayed, or
- they 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 status information is very helpful. In the positive case – all planned train changes are still possible – passengers can be reassured that they do not have to worry about missing their connecting train(s). To learn that one will arrive 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. 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 options to react. This situation is clearly preferable to missing a connecting train and then using any information system (ticket machine, service counter) to request an alternative.

Because up to now commercial systems do not take the current situation into account (even though 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.

From static to real-time 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 [2] describes the models and suitable algorithms in detail. Previous research on timetable information systems has focused on the static case, where the timetable is considered as fixed.

Here we start a new thread of research on dynamically changing timetable data as a consequence of disruptions. Our contribution is

- the development of a first prototypal but yet completely realistic timetable information system that incorporates current train status information into a multi-criteria search for attractive train connections. Modeling issues have been discussed in the literature on a theoretical level [3] but no true-to-life system with real delay data has been studied and, to our knowledge, no such system that guarantees optimal results (with respect to even a single optimization criterion) exists. We provide results of implementing such a system for a real world scenario with no simplifying assumptions.
- We propose a system architecture 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, a *real-time information server* and one or several *search servers*. The real-time information server receives a massive stream of status messages about delayed trains. Its purpose is to integrate schedule changes into the "planned schedule". Moreover, it has to compute from the received messages (primary delays) all so-called secondary delays which result from trains waiting for each other according to certain waiting policies. The new overall status information is then sent to the search servers which incorporate all changes into their internal model. Search servers, in turn, are used to answer customer queries about train connections.
- Both servers require a specific graph model as the underlying basic data structure. We here introduce the concept of a *dependency graph* as a model to efficiently propagate primary delay information according to policies for delays in the real-time information server. Our dependency graph (introduced in Section 4) is similar to a simple time-expanded graph model with distinct nodes for each departure and arrival event in the entire 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 server we use a *search graph*. Here, we are free to use either the time-expanded or the time-dependent model. In this work, we have chosen to use the time-expanded model for the search graph, since our previous work, the timetable information server MOTIS [4], 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 $17\mu s$ per update message on average.
- To store a full timetable over a typical period of a year, static timetable systems are usually built on a compact data structure. For example, they identify the same events on different days of operation and use bitfields to specify valid days. This space saving technique does not work in a dynamic environment since the members of such an equivalence class of events have to be treated individually, as they will have, in general, different delays. We will show how a static time-expanded graph model can be extended to a dynamic graph model without undue increase in space consumption.

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 timestamps. 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. Due to a number of low-level optimizations we achieve a considerable speed-up over the preliminary work in Frede et al. [1].

A related field of current research is disposition and delay management. Gatto et al. [5, 6] 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 policies for delays have been discussed, for example by Ginkel and Schöbel [7]. Schöbel [8] also proposed integer programming models for delay management. Stochastic models for the propagation of delays are studied, for example, by Meester and Muns [9]. Policies for delays in a stochastic context are treated in [10].

Overview. The remainder of this paper is organized as follows. In Section 2, we will discuss primary and secondary delays. We introduce our system architecture in Section 3, and describe its two main components afterwards. First, we explain our dependency graph model and the propagation algorithm for delays (in Section 4). Then, we briefly review the time-expanded graph model and our search server MOTIS (Section 5). Afterwards, we present the update of the search graph (in Section 6). A major issue for a real system, the correct treatment of days of operation, will be discussed in Section 7. Afterwards, we provide our experimental results in Section 8. Finally, we conclude and give an outlook.

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 trains (specified by 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 policies for delays are:

- *never wait* In this policy, no secondary delays occur at all in our model. 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 AG 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 its direction, the waiting time is increased, or during peak hours, when trains operate more frequently, the waiting time may be decreased.

The *waiting time* $wt_s(t, f)$ is the maximum delay acceptable for train t at station s waiting for a feeding train f. Let $dep_s^{sched}(t)$ 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. Train t waits for train f at station s if

$$arr_s(f) + minct_s(f, t) - dep_s^{sched}(t) < wt_s(t, f).$$

In this case, train t will incur a secondary delay. Its new departure time is determined by the following equation

$$dep_s(t) = \begin{cases} arr_s(f) + minct_s(f, t) & \text{if } t \text{ waits} \\ dep_s^{sched}(t) & \text{otherwise.} \end{cases}$$

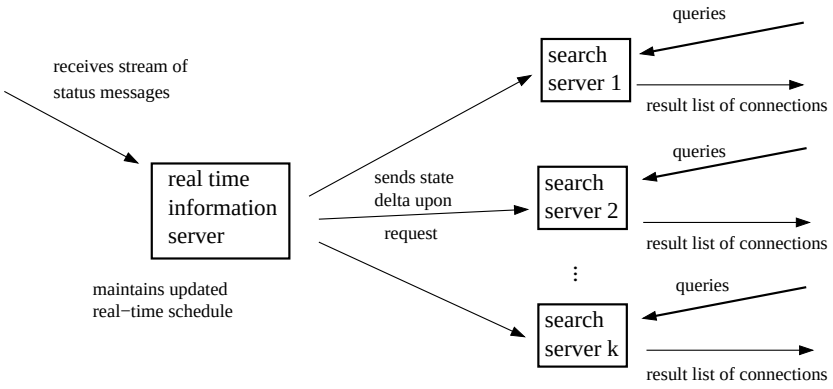

Fig. 1. Sketch of the system architecture.

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

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, see Figure 1 for a sketch. 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, our *real-time information server*, is a *dependency graph* which models all the dependencies between different trains and between the stops of the same train and is used to compute secondary delays (in Section 4 we introduce in detail the dependency graph and propagation algorithm). The dependency graph stores the obtained information needed to update the search servers and transmits this information in a suitable format to them. The search servers in turn update their internal graph representation whenever they receive these changes. 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 running as the real-time information server that continuously receives new status information and broadcasts it. We will present some details in the following subsection. 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.

Multi-Server Approach

The *search server* mainly consists of a search graph, an update component for the search graph, and a query algorithm.

In a multi-server environment, updates of a search server are either triggered by a load balancer or an internal clock after a maximum amount of time without update. The data it receives (called *state delta* for the remainder of this work) are lists of changed departure and arrival times as well as meta-information about additional and canceled trains and connection status decisions. Subsequently, it adjusts the search graph accordingly and thereafter the graph looks exactly as if it were constructed from a schedule with all these updated departure and arrival times. Thus, the search algorithm does not need to know whether it is working on a graph with updated times or not.

The *real-time information server* receives all the up-to-date status information, uses its internal *dependency graph* to compute updated departure and arrival times (cf. Section 4) and stores these and the meta-information in a data structure `UDS` (*update data structure*). `UDS` maintains for every event with a changed timestamp a 3-tuple consisting of (1) a reference to the event itself, (2) the latest updated timestamp of this event, and (3) the release time when the last update of this event took place. Whenever a search server requests an update, it receives all events with a release time later than the last update of that server. If the timestamp of an event (or node in the graph model) changes, we call the necessary update a *(node) shift*.

For a true multi-server architecture with multiple search servers we basically have two update scenarios:

- An additional search server joins in and has to be initialized to the current time: We iterate over all existing event entries in `UDS` and transmit all those with times differing from the scheduled time.
- A search server has answered a number of queries and now enters update mode: We could simply transmit all events with release time greater than the last update time of the search server (referenced as *iterator version*). As this requires iterating over all stored events even to calculate a small delta, we can do more efficiently utilizing a map (referenced as *map version*).

In the map version a map of all changed events and their previous event time is maintained for each search server individually. Whenever a new event time is released, we look for that event in the map. Only if it is not already present, we store the event itself and its event time before the last change. This is the current timestamp of the event in the search server. To answer an update request we simply return all events in this map, whose new event time differs from the event time in the map (and thus the time in the current server), and clear the map afterwards. Using this technique we not only save iterating over all entries to determine the set of changed events (our state delta) but also avoid transmitting events that have been changed more than once and do not require a shift, since their new event time is the same as in the last update.

Our `UDS` data structure enables us to transmit only consistent state deltas on demand. Thereby, we can decrease both the time spent in communication and updating the graphs (e.g. if between two update phases more than one information for a single event is processed in the dependency graph, it is not required to transmit the intermediate state and adjust the graph accordingly).

4 Dependency Graph

4.1 Graph Model

Our *dependency graph* (see Figure 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 timestamp which can dynamically change. Departure and arrival nodes are in one-to-one correspondence with departure and arrival events. Their timestamps 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 timestamp of a forecast node is the current external prediction for its 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 timestamp 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 timestamp 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 disembarking of passengers). Thus, for a standing edge e, we set $\tau(e) = arr_s(t) + stand_s(t)$. Standing edges are always active.

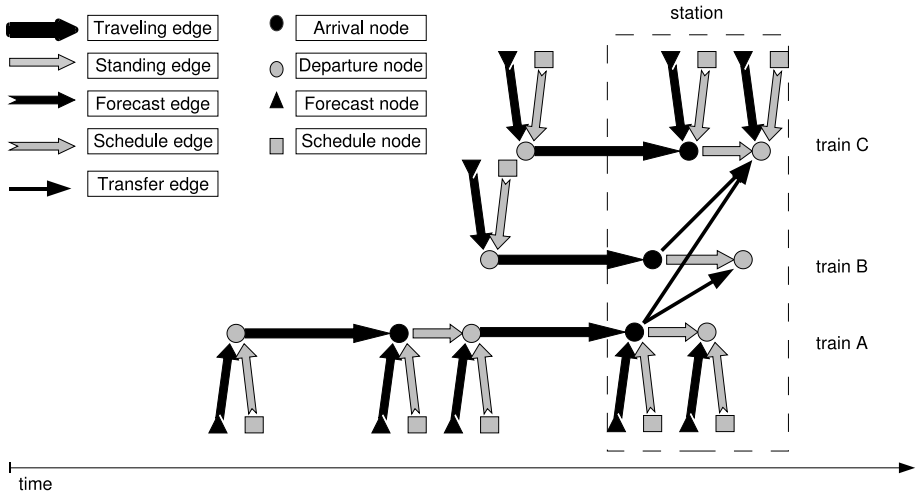

Fig. 2. Illustration of the dependency graph model.

– *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).

  Due to various, mostly unknown factors determining the travel time 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. However, if a more sophisticated, but efficiently computable oracle for $tt(s, s', t)$ taking the mentioned factors into account were available, it could be used without changing our model.

– *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 $\tau(e) = wait_s(t, f)$, where

$$wait_s(t, f) = \begin{cases} arr_s(f) + minct_s(f, t) & \text{if } t \text{ waits for } f \\ 0 & \text{otherwise} \end{cases}$$

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

$$arr_s(f) + minct_s(f,t) - dep_s^{sched}(t) < wt_s(t,f)$$

or if 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 set $\tau(e)$ to the resulting delayed departure time or to zero, respectively.

4.2 Computation on the Dependency Graph

The current timestamp 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 $f_1, \ldots, f_n$ we have $dep_s(t) =$

$$\max\{dep_s^{sched}(t), dep_s^{for}(t), arr_s(t) + stand_s(t), max_{i=1}^{n}\{wait_s(t,f_i)\}\}.$$

For an arrival we have

$$arr_s(t) = \max\left\{arr_s^{sched}(t), arr_s^{for}(t), dep_{s'}(t) + tt(s',s,t)\right\}$$

with the previous stop of train t at station s'. Inactive edges do not contribute to the maximum in the preceding 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 timestamps) of all nodes whose timestamps have changed since the last computation was finished. Whenever we have new forecast messages, we update the timestamps of the forecast nodes and, if they have changed, insert them into the queue. For a connection status decision we modify the corresponding transfer edge and update its head node. If its timestamp changes, it is inserted into the queue. As long as the queue is not empty, we extract a node from the queue and update the timestamps of the dependent nodes (which have an incoming edge from this node). If the timestamp 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 timestamp 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 timestamps have been computed and the nodes with changed timestamps can be sent to the search graph update routine or, in the multi server architecture, to the `UDS` data structure.

A note on the implementation. For ease of exposition we have introduced all kinds of nodes and edges in the dependency graph as being real nodes and edges. Of course, in our implementation we do not use a node and an edge to encode nothing more than a single timestamp for schedule and forecast times. Only arrival and departure nodes are real nodes with entering and leaving edges plus two integer variables representing the scheduled and forecast time. The latter is set to some predefined value to specify "no real-time information available (yet)". An arrival node has a container of leaving transfer edges, one entering traveling edge and one leaving standing edge. Analogously, a departure node has a container of entering transfer edges, one entering standing edge and one leaving traveling edge. Iterators over incoming dependencies and markers for the current input determining the timestamp of the node (the incoming edge or schedule or forecast time with maximum timestamp) have to be able to traverse resp. point to the different representations. We deemed the much more elegant version of the update routines - pretending the existence of nodes and edges for schedule and forecast times as well - better suited for presentation.

5 Time-Expanded Search Graph Model

5.1 The Static Model

Let us briefly describe the realistic time-expanded search graph model used in this work. Its basic idea is — as in the dependency graph before — to model each departure and arrival event of some train as a node with a timestamp. Each timestamp here represents the time after midnight in minutes.

Again, for each departure event of some train there is a *traveling edge* to its very next arrival event. With each traveling edge we associate a number of additional attributes: a bitfield representing traffic days, with one bit for each day of the schedule period, and several train attributes (train category, train number and name, availability of extra services, and the like).

The difference between the dependency graph and the search graph comes from the need to model the transfer between trains more explicitly in the latter case so that a Dijkstra-like shortest path algorithm can be used. Non-constant transfer times between pairs of connecting trains are modeled with the help of additional *change nodes*. For every departure time at a station there is a change node which is connected via *entering edges* to all departure nodes at that time. Change nodes at the same station are ordered increasingly by their timestamp, and subsequent change nodes (in this order) are interconnected with *waiting*

edges. Moreover, at each station the last change node before midnight is linked to the first change node after midnight. For each arrival node there is a *leaving edge* connecting it to the corresponding first change node which is reachable in the time needed for a transfer from this train to any other. All possible shorter transfer times (e.g. for trains at the same platform) are realized using *special transfer edges*. The subgraph formed by all edges incident to change nodes of a certain station will be referred to as the *change level* of this station.

Finally, we have *stay-in-train edges* each of which connects the arrival node of some train to the corresponding departure node at the same station, provided the latter exists. 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 traveling 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 train connections and paths in the graph. More details of the graph model can be found in [4].

5.2 Search Server MOTIS

Over the last years, the authors developed the timetable information system MOTIS (multi-objective train information system) which performs a multi-criteria search for train connections in a realistic environment based on the above described time-expanded graph. To be more precise, MOTIS uses an extension of this model to incorporate additional features from practice which we omit here for clarity of the presentation.

Our underlying model ensures that each proposed connection is indeed feasible, i.e. can be used in reality. MOTIS is parameterized to search with respect to a selection of optimization criteria, most importantly travel time, number of interchanges, ticket cost, and reliability of all interchanges of a connection. The search algorithm used within MOTIS is a generalized multi-criteria Dijkstra-like algorithm enhanced with some additional speed-up techniques like goal-directed search [4]. The system is designed not only to present the true Pareto-optima, but more generally "all attractive" connections to customers, see also [4]. MOTIS has successfully been extended to search for low-cost connections [11] and night trains [12].

5.3 The Dynamic Model

The static time-expanded graph model has been slightly adapted for the dynamic scenario. Compared to the standard search graph we have to store additional information, namely status decisions, a second timestamp for each node to report actual and scheduled time in query results, additional strings containing reasons for the delays, and the like. Moreover, we need a slightly different representation of trains with identical schedules on multiple days. We defer details of this modification to Section 7.

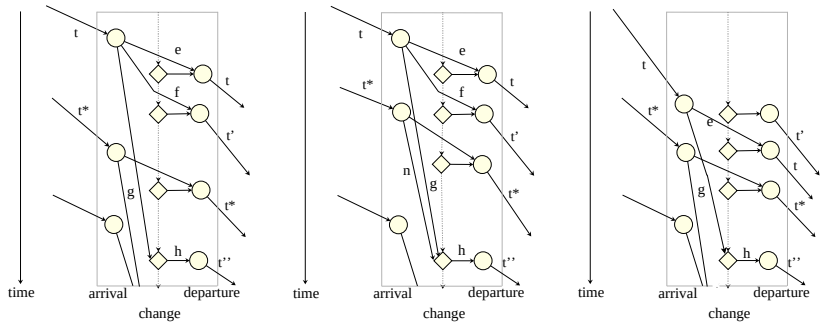

Fig. 3. The change level at a station (left) and necessary changes if train t^* arrives earlier (middle) or train t arrives later (right).

6 Updating the Search Graph

The update in the search graph does not simply consist of setting new timestamps for nodes (primary and secondary delays), insertions (additional trains) and deletions (cancellations) of nodes and resorting lists of nodes afterwards. All the edges modeling the changing of trains at the affected stations have to be recomputed respecting the changed timestamps, additional and deleted nodes, and connection status information. The following adjustments are required on the change level (see Figure 3):

- 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.
- Inserting additional change nodes or unhooking them from the chain of waiting edges at times where a new event is the only one or the only event is moved away or canceled.
- Recalculating special interchange edges from resp. to arrival resp. departure nodes with a changed timestamp (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. In Figure 3 (left) it is possible to change from train t to all trains departing not earlier than t'' using leaving edge g, any number of consecutive 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. Suppose train t^* manages to get rid of some previous delay and now arrives and departs earlier

than previously predicted (see Figure 3, middle part). In the new situation it is now possible, to change from t^* 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 3 (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 departure nodes for the departures of t' and t are in reverse order. Therefore, the waiting edges have to be re-linked.

7 Traffic Days

A common simplification in theoretical work on timetable information systems is the assumption that trains operate periodically. Often even a periodicity of one hour is used. In real schedules, however, there is a considerable difference between peak hours, late evenings and "quiet" nights. For our timetable server MOTIS we take time modulo a single day in order to have a better manageable graph size as opposed to full time expansion. Recall that traveling edges carry traffic day flags (stored in bitfields) to model the days of operation, e.g. trains operating only on weekdays, or weekends, different schedules for school days and non school-days, trains operating on public holidays according to the weekend schedule etc.

In our scenario with delay information we have to take care of multiple traffic days as well. To be able to present the customer with updated information we need to model not only "today" (the current day) but also tomorrow as some connections might pass the midnight border (have a so-called "*night jump*"), especially if we query with a departure in the afternoon or evening. Resulting alternative journeys, requested after a delay on a journey, may even end on the next day due to delays, although no night jump was present in the original connection.

Therefore, we chose a schedule length of two days. In our time-expanded graph we represent all the trains operating today and tomorrow. However, trains that have the same schedule on both days can no longer be represented just once with two traffic day flags set. To be able to shift today's train without affecting the version of tomorrow, thus not incorrectly cloning delays, or vice versa, we need two distinct versions of such trains.

7.1 Memory Consumption Issues

The simplest version to attain separate nodes for today's and tomorrow's events is to use full time expansion on all our schedule days and not take time modulo 1440 and use traffic day flags on the train edges. Unfortunately, this would not only increase the number of event nodes and edges as well as the change edges, it would also significantly increase the number of change nodes and waiting edges. Whereas there is no way to avoid the increase for the former type of nodes and edges, we found a way to keep the size of the rest the same: We only use full time

expansion for departure and arrival events and link all events to a change level with only one node per necessary timestamp, regardless of the day of that event, i.e. the number of change nodes and waiting edges remains the same, only the number of adjacent edges to the change nodes increases. Three different models for the search graph arise.

- Model (A) is the static model where the same events on two subsequent days are represented only once but two traffic day flags are set.
- Model (B) treats each arrival and departure event individually and uses the sparse change level implementation as described above.
- Model (C) also treats each arrival and departure event individually but uses full change level expansion.

Note that in the dependency graph we opted for full time expansion. There is no change level with waiting edges and all the change representation is between the trains itself and only necessary to decide whether trains wait for others or not and compute the resulting changes. In this model a source delay propagation may or may not delay events on the following day. There is no need for case distinctions due to day changes.

Test data. To study the effect of these models on the space consumption, we use the train schedule of Germany in 2008. The schedule contains 68,300 trains for the whole year with over 5,000 distinct bitfields for the days of operation. We look at the graphs prepared for two subsequent days, either two weekdays, Wednesday and Thursday (We & Th) with 38,600 trains with distinct schedules or for Sunday and Monday (Su & Mo) with 46,600 trains with distinct schedules.

Comparison of models. In Table 1, we compare our three different models for the search graph. For the more homogeneous case of two weekdays version (C) requires double the amount of space while for our variant (B) we manage to increase the number of nodes by two thirds and the number of edges by four fifths. The tremendous increase of (C) is due to the large number of trains operating identically each weekday. If we look at the graph for Sunday and Monday the increase is much smaller as many of the trains operate either on Monday or on Sunday, therefore the increase in nodes and edges for the trains is below 50%. Still our model improves the additional required memory space from nearly one half to about one third.

During the actual search for train connections, variant (B) has a slight running time overhead in comparison with full time expansion (C). This overhead turns out to be negligible if a look-ahead in the search process categorizes entering edges as not allowed if they lead to a departure node for a train not operating on the required day.

	model	unit	event nodes	train/std edges	change nodes	change edges	waiting edges	total nodes	total edges
We & Th	(A)	(in k)	988	950	459	988	459	1447	2397
We & Th	(B)	(in k)	1956	1878	459	1954	459	2415	4291
We & Th	(C)	(in k)	1956	1878	912	1954	912	2868	4744
increase	(A → B)	(in %)	98.0	97.7	0	97.8	0	66.9	79.0
increase	(A → C)	(in %)	98.0	97.7	98.7	97.8	98.7	98.2	97.9
Su & Mo	(A)	(in k)	1181	1134	498	1180	498	1679	2812
Su & Mo	(B)	(in k)	1702	1634	498	1701	498	2200	3833
Su & Mo	(C)	(in k)	1702	1634	798	1701	798	2500	4133
increase	(A → B)	(in %)	44.1	44.1	0	44.2	0	31.0	36.3
increase	(A → C)	(in %)	44.1	44.1	60.2	44.2	60.2	48.9	47.0

Table 1. Sizes of the search graph for two days, Wednesday and Thursday resp. Sunday and Monday and the increase when changing between the models (A), (B), and (C) as described in the text.

7.2 Moving from One Day to the Next

At midnight we have to change the current day for our real-time information server as well as the search servers. Now, information about yesterday is no longer relevant as tomorrow becomes today and we need to load the "new tomorrow".

The real-time information server loads the dependency graph for tomorrow and "forgets" yesterday. With the fully time-expanded model there is no hassle in doing so. Note that we still have to keep yesterday's events that are delayed to today and have not happened yet, but nothing more about yesterday is needed any longer. Thus, we can delete all information about yesterday's events in the data structure. With our prototype, this whole procedure is finished in less than 35 seconds for the complete German timetable.

The search servers need a longer update phase than usual as they have to be restarted with the now current day and the following day. Afterwards, they request an update for a new server (exactly as described for a new server in Section 3). In this update they receive all information for today currently available. These updates typically take less than ten seconds. Together with the restart time of about 20 seconds a single search server is down for about half a minute. Even a server that has not yet changed days can still be updated after midnight and produce correct search results, as only the information about the next day is missing, not the current day. So there is no problem with the last server updating at say 01:00 a.m. Since midnight is not a peak hour for timetable information systems a number of servers might change days concurrently without compromising the availability of service. In summary, within a multi-server solution down-times of individual servers can easily be hidden from the customer.

8 Evaluation of the Prototype

We implemented the dependency graph and the update algorithm described in Section 4 and extended our time table information system MOTIS to support updating the search graph (cf. Section 6). 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 subsection, we present results for a standard operating day with an average delay profile testing various waiting profiles broadcasting the update information as soon as it becomes available. In the succeeding subsection we will present first results for the multi-server architecture (as described in Section 3) and test different update intervals. All experiments were run on an Intel Xeon 2.6 GHz with 8GB of RAM.

As no system with the capabilities of our prototype exists, we cannot compare our results to others. To ensure the correctness of our approach we used automated regression tests continuously checking the status of a large number of connections and determining alternatives, collecting meta-information about the encountered delays in the process. Furthermore, we intensively investigated isolated test cases (e.g. explicit search for trains known to us that they were delayed, search for trains departing next to a delay, searches for which the off-line optimum was affected by a delay).

8.1 Overall Performance and Waiting Profiles

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."
- *half* All times of scenario standard are halved.
- *double* All times of scenario standard are doubled.
- *all5* All times of scenario standard are set to five minutes, and in addition regional trains wait 5 minutes for all but urban trains.

search graph	
event nodes	0.99 mil
change nodes	0.46 mil
edges	2.40 mil

dependency graph	
events	0.97 mil
standing edges	0.45 mil
traveling edges	0.49 mil

Table 2. Properties of the search graph (left) and dependency graph (right) for one day.

transfer edges	5min	15min	30min	45min	60min
std / half / double	7.1k	54.7k	123.8k	207.8k	267.8k
all5 / all 10	14.6k	168.3k	399.6k	665.4k	874.3k

Table 3. The number of transfer edges depending on the waiting policy and the maximum allowed time difference δ between feeding and connecting train.

– *all10* 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.

Our dependency graph model assumes that we know at each station which pairs of trains have potentially to wait for each other, i.e., which transfer edges are present. In our implementation we use the pragmatic rule, that if the difference between the departure event of train t_1 and the arrival event of another train t_2 at the same station does not exceed a parameter δ, then there is a transfer edge between these two events.

For each of these different waiting profiles we tested different maximum distances (in minutes) of feeding and connecting trains $\delta \in \{5, 15, 30, 45, 60\}$, and compare them to a variant without waiting for other 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 policies for delays is carried out, only the terms for feeding trains are always zero.

We constructed search and dependency graphs from the real schedule consisting of 36,700 trains operating on the selected day. There are 8,817 stations in the data. The number of nodes and edges in both graphs are given in Table 2. The number of standing and traveling edges are in one-to-one correspondence to the stay-in-train and traveling edges of the search graph. The number of transfer edges depends on the waiting policy and parameter δ and can be found in Table 3. Note that, whether a transfer edge exists or not, depends on the classes that wait for each other and not on the actual number of minutes they wait. Therefore, the number of edges are identical for the policies *half*, *standard*, and *double* as well as for the policies *all5* and *all10*. There is a monotonous growth in the number of transfer edges depending on the parameter δ. Additionally, the number of these edges increases as more trains wait for other trains because of additional rules.

In Table 4, we give the results for our test runs for the different policies and values of δ. Running times are averages over 25 test runs. 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. 336,840 sequences are handled. In total we had 6,340,480 forecast messages, 562,209 messages of the type "this train is now here" and 4,926 connection status decisions. Of all forecast messages 2,701,277 forecasts are identical to the last message already processed for the corresponding nodes. The remaining messages either trigger computations in the dependency graph or match the current timestamp 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 4. Depending on the policy we have a different number of nodes that were shifted and stations that have at least one delayed event (last two columns of the table).

The key figures for the computational efficiency (required CPU times in seconds, operation counts for the number of touched stations and node shifts in multiples of thousand) 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. We notice a significant growth in all key criteria when increasing δ from 5 to 15. All policies behave rather similarly for $\delta = 5$, whereas the differences between the realistic policies and the extreme versions and even from *all5* to *all10* for higher values of δ is apparent.

Amongst the plausible policies there is only a 16% difference in the number of moved nodes. It little more than doubles going to policy *all5* and even increases by a factor of 3.8 towards policy *all10*. Roughly 40 seconds of our simulation time are spent extracting and preprocessing the messages from the forecast stream. This IO time is obviously independent of the test scenario. The increase in running time spent in the search graph update is no more than 3 seconds for $\delta > 5$ for all policies except *all10* with 7 seconds and differs by at most 10 seconds or 17% among the realistic scenarios. The running time scales with the number of shifts. An increase of factor 1.9 resp. 3.4 of node shifts results in a factor of 1.8 resp. 3.3 in running time (compare policies *double* to *all5* and *all10* with $\delta = 60$). The time spent in the dependency graph differs by at most 1 second (about 16%) for realistic scenarios and stays below 30 seconds even for the most extreme policy.

Even for the most extreme scenario a whole day can be simulated in less than 5 minutes. The overall simulation time for realistic policies lies around 2 minutes. For the policy *standard* with $\delta = 45$, we require on average $17\mu s$ reconstruction work in the search graph per executed node shift. By incident, also the overall runtime per computed message is $17\mu s$.

Worst-case considerations (based on policy standard with $\delta = 45$). The highest number of messages received per minute is 15,627 resulting in 29,632 node shifts and a computation time of 0.66 seconds for this minute. However, the

Instance policy	δ in min	Computation time for SG in s	DG in s	IO in s	total in s	Node shifts in k	With delay nodes in k	stations
no wait	-	59.8	6.4	39.4	105.6	3,410	396.2	5,385
	5	59.1	6.2	40.0	105.3	3,432	396.6	5,397
	15	60.7	6.4	39.7	106.8	3,525	400.1	5,483
half	30	60.8	6.4	40.4	107.7	3,535	400.4	5,494
	45	61.2	6.5	40.0	107.8	3,539	400.6	5,494
	60	62.3	6.8	39.7	108.8	3,540	400.7	5,496
	5	59.1	6.2	39.3	104.6	3,443	396.8	5,408
	15	62.6	6.5	39.5	108.5	3,614	402.5	5,532
standard	30	63.4	6.7	40.1	110.2	3,636	403.2	5,541
	45	63.6	6.8	39.9	110.2	3,646	403.6	5,541
	60	63.6	6.7	40.3	110.7	3,651	403.7	5,545
	5	58.9	6.3	39.7	104.9	3,447	396.8	5,419
	15	66.4	6.6	40.4	113.4	3,835	406.2	5,590
double	30	67.9	6.9	40.5	115.3	3,908	407.5	5,639
	45	69.4	7.2	40.1	116.7	3,945	408.0	5,642
	60	69.0	7.3	39.9	116.2	3,959	408.1	5,642
	5	60.7	6.4	40.3	107.4	3,623	403.5	5,588
	15	123.1	11.5	40.0	174.6	7,603	440.5	6,051
all5	30	124.9	13.0	40.4	178.3	7,670	442.8	6,064
	45	124.9	14.7	40.6	180.2	7,687	443.4	6,064
	60	126.0	16.5	40.4	182.9	7,689	443.7	6,070
	5	60.7	6.4	40.4	107.5	3,651	404.0	5,608
	15	193.8	19.0	39.8	252.6	13,052	457.9	6,118
all10	30	195.2	21.6	40.9	257.7	13,231	463.0	6,145
	45	198.0	24.6	40.6	263.2	13,346	464.4	6,148
	60	200.7	27.3	40.7	268.7	13,466	465.3	6,162

Table 4. Computation time for the whole day (propagation in the dependency graph (DG) and update of the search graph (SG), IO and total) and key figures (in multiples of thousand) for the executed node shifts in the search graph and the number of nodes and stations with changed status information with respect to different policies for delays.

largest amount of reconstruction work occurred in a minute with 5,808 messages. It required 172,432 node shifts and took 2.38 seconds; this is the worst case minute which we observed in the simulation. Thus, at our current performance we could easily handle 25 times the load without a need for event buffering. This clearly qualifies for live performance.

8.2 Multi-Server Performance

As we have seen in the previous subsection most of the time is spent in reconstructing the search graph. Applying sophisticated software engineering the update process has been sped up considerably. Additionally, a big potential lies in doing less reconstruction work. In a real-time environment it is not necessary to update multiple times per minute as soon as new information is available (as we did in the previous subsection). It clearly suffices to update each minute. Depending on the load balancing scheme every 2 or 3 minutes might still produce results of high quality.

To be able to compare the numbers to the previous section we tested the two servers as introduced in Section 3 "in line", i.e. one waited for the other to finish computation before continuing his own work. We use our waiting profile "standard" with $\delta = 45$ for all versions. The *baseline* version does not use the `UDS` and immediately updates the search graph. The version *split* additionally inserts and retrieves events into/from the `UDS`. Our code spends about 47 seconds on extracting and preprocessing the messages from the forecast stream and propagation in the dependency graph. Pushing all the events through the `UDS` data structure in the split architecture only requires an additional 7.2 seconds.

As we do not see a need for update intervals shorter than one minute, we now read all incoming messages for a particular minute and calculate the resulting changed event times in the dependency graph. These are transmitted to the data structure `UDS` in our real-time information server part. Meanwhile the search graph requests an update every 1, 2, 3, 4, or 5 minutes, using either the *iterator* or *map* version. The results can be found in Table 5. The numbers are averages over 25 runs.

By sending the state delta of the last x minutes as a batch job to the search graph we save a lot of reconstruction work due to mutually interacting messages arriving between two subsequent updates, e.g. oscillating forecasts for trains, or reconstruction is done for a train but later it is shifted again due to a changed arrival time of one of its feeding trains.

With increasing interval size the number of messages to transmit significantly decreases. The resulting time required for updating the search graph is sped up by nearly 10 seconds when changing from immediate update to an interval of one minute. The increase of the interval size by one additional minute within the range of [1-5] reduces the execution time by a few seconds.

The *iterator* version of detecting events to transmit (cf. Section 3) only uses the release time information and cannot detect that an event does not require shifting, therefore it transmits 149k to 404k (depending on the update interval) of these irrelevant messages demanding a node "shift" to the node's current

Instance		Transmissions		Computation time			
		needed	unnec-	SG	UDS		total
Version	interval		essary		ins	ext	
	in min	in k	in k	in s	in s	in s	in s
baseline	-	3646	0	63.5	0.0	0.0	110.3
split	-	3646	0	63.7	3.7	3.5	117.7
iterator	1	3143	0	53.9	3.1	55.6	159.0
	2	2809	149	45.5	2.9	29.3	124.4
	3	2447	284	38.3	2.9	20.4	108.2
	4	2177	360	33.3	2.8	15.8	98.6
	5	1954	404	29.3	2.8	13.1	91.4
map	1	3143	0	54.3	4.9	2.1	107.4
	2	2809	0	45.4	4.9	1.9	98.5
	3	2447	0	38.3	4.8	1.8	91.2
	4	2177	0	33.4	4.7	1.7	86.3
	5	1954	0	29.2	4.7	1.5	81.7

Table 5. The number of transmitted events, node shifts and execution time for simulating the whole day. We compare version with and without two server architecture using an iterator or a map to determine the relevant events (cf. Section 3) for different update intervals.

position. On the other hand, the *map* version only transmits events with changed timestamp, even if the release time is newer, therefore we do not have unnecessary transmissions. As shifts to the same position are never executed we only have the unnecessary transmission and no extra work, as we can see with the identical run times for the search graph update (column SG).

Inserting the information (column UDS ins) about changed event times into the `UDS` takes between 2.8 and 3.7 seconds, depending on the number of insertions (and thus the interval size). For the *map* version the bookkeeping requires an additional 1.8 to 2.0 seconds for the whole day. While the extraction (column UDS ext) using the *map* version requires 1.5 to 2.1 seconds, iterating for each update over all stored events to find the relevant new information in the *iterator* version is very costly and takes 13.1 to 55.6 seconds. Obviously, these times do not depend on the number of transmissions but on the number of iterations, as we observe that the extraction time is inversely proportional to the interval size.

The improvement in run time of 3 seconds (from 110.3 to 107.4 seconds), when changing from the baseline version to the split version with an interval of one minute, does not seem like much. However, it enabled us to do load balancing and handle updates on demand with our multi-server approach. The update time for the search servers consists of the time for receiving events from the `UDS` plus the time for the search graph update. Therefore, instead of taking 110.3 seconds to read messages, propagate delays and update the search in our baseline version, we only need 56.4 seconds in the split architecture for keeping the search graph up-to-date. Thus, we gain more than 50 seconds of available computation time

per search server (about half the time required by the baseline version that does all the work on its own). Together with the initial startup phase and the first update with all relevant information for today depending on yesterday's data of about half a minute (cf. Subsection 7.2) a search server is only 60 to 90 seconds per day busy with startup and updating. This means that each search server can use 99.9% of its time for answering search queries.

The real-time information server spends about 47 seconds for reading messages and propagation in the dependency graph and additional 3 seconds storing the data. For each registered server (in our tests just one) it takes 2 seconds maintaining the map of relevant events and 2 seconds to extract and transmit those events. Thus, we have by far enough time to update a multitude of search servers.

9 Conclusions and Future Work

We have built a first prototype which can be used for efficient off-line simulation with massive streams of delay and forecast messages for typical days of operation within Germany. Using the presented multi-server solution, the correct handling of all necessary updates is so fast that each search server can use almost all of its time for answering search queries. Stress tests with extreme policies for delays showed that the update time scales linearly with the amount of work. So even for cases of major disruptions we expect a sufficient performance of such a multi-server solution. Compared to typical stream profiles, we are able to handle about 25 times as much reconstruction work.

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. [13]).

A true real-time timetable information system as demonstrated by our prototype opens the door for a new service to passengers who want their travel plans supervised. The provider of such a service would constantly check the validity of planned connections, and in case of necessary changes due to delays inform the affected passenger and propose new alternative connections by sending messages to a mobile phone or an email address.

Acknowledgments

This work was partially supported by the DFG Focus Program Algorithm Engineering, grant Mu 1482/4-1. Thanks go to our students Lennart Frede and Mohammad Keyhani who contributed to the software design and implementation. We wish to thank Deutsche Bahn AG for providing us with timetable data and up-to-date status information for scientific use, and Christoph Blendinger and Wolfgang Sprick for many fruitful discussions.

References

1. Frede, L., Müller-Hannemann, M., Schnee, M.: Efficient on-trip timetable information in the presence of delays. In Fischetti, M., Widmayer, P., eds.: Proceedings of ATMOS 2008 - 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2008)
2. 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
3. Delling, D., Giannakopoulou, K., Wagner, D., Zaroliagis, C.: Timetable Information Updating in Case of Delays: Modeling Issues. Technical report ARRIVAL-TR-0133, ARRIVAL Project (2008)
4. 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
5. 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
6. 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
7. Ginkel, A., Schöbel, A.: The bicriteria delay management problem. Transportation Science **41** (2007) pp. 527–538
8. 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
9. Meester, L.E., Muns, S.: Stochastic delay propagation in railway networks and phase-type distributions. Transportation Research Part B **41** (2007) 218–230
10. 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)
11. Müller-Hannemann, M., Schnee, M.: Paying less for train connections with MOTIS. In Kroon, L.G., Möhring, R.H., eds.: 5th Workshop on Algorithmic Methods and Models for Optimization of Railways, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
12. Gunkel, T., Müller-Hannemann, M., Schnee, M.: Improved search for night train connections. In Liebchen, C., Ahuja, R.K., and Mesa, J.A., eds.: Proceedings of ATMOS 2007 - 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2007). Extended journal version appears in Networks.
13. 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 Verlag (2008) 347–361