Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
87(7) 616–629

DOI: 10.1177/0037549710373571
sim.sagepub.com
SAGE SCS

Online railway delay management: Hardness, simulation and computation*

André Berger[1], Ralf Hoffmann[2], Ulf Lorenz[3] and Sebastian Stiller[4]

Abstract

Delays in a railway network are a common problem that railway companies face in their daily operations. When a train is delayed, it may either be beneficial to let a connecting train wait so that passengers in the delayed train do not miss their connection, or it may be beneficial to let the connecting train depart on time to avoid further delays. These decisions naturally depend on the global structure of the network, on the schedule, on the passenger routes and on the imposed delays. The railway delay management (RDM) problem (in a broad sense) is to decide which trains have to wait for connecting trains and which trains have to depart on time. The offline version (i.e. when all delays are known in advance) is already NP-hard for very special networks. In this paper we show that the online railway delay management (ORDM) problem is PSPACE-hard. This result justifies the need for a simulation approach to evaluate wait policies for ORDM. For this purpose we present TOPSU–RDM, a simulation platform for evaluating and comparing different heuristics for the ORDM problem with stochastic delays. Our novel approach is to separate the actual simulation and the program that implements the decision-making policy, thus enabling implementations of different heuristics to "compete" on the same instances and delay distributions. We also report on computational results indicating the worthiness of developing intelligent wait policies. For RDM and other logistic planning processes, it is our goal to bridge the gap between theoretical models, which are accessible to theoretical analysis, but are often too far away from practice, and the methods which are used in practice today, whose performance is almost impossible to measure.

Keywords

online railway delay management, Web-based simulation, discrete event simulation, PSPACE

1. Introduction

Delays in a railway network are one of the biggest problems for the daily operations of a railway company. Delayed trains and missed connections lead to dissatisfied customers and possibly refunds that have to be paid to delayed passengers.

When a train does get delayed, it may be beneficial to let another train wait so that passengers in the delayed train do not miss their connection. However, passengers in the waiting train will then get delayed and may in turn miss their connections. Owing to the complexity of both the network and the schedule, one decision may have a large impact on the propagated delays later during the day.

Even today, the decision whether a train should wait or not, is still made by a human dispatcher, mainly based on a lot of training and experience. However, to decrease the delays incurred by making bad decisions, it may be favorable to have these decisions made by an algorithm or at least support a dispatcher by making proposals. Algorithmically, this becomes the problem of finding a good wait policy, a mechanism that decides at any point

[1]Maastricht University, Department of Quantitative Economics, PO Box 616, 6200 MD, Maastricht, The Netherlands.
[2]Technische Universität Berlin, Department of Mathematics, Berlin, Germany.
[3]Technische Universität Darmstadt, Department of Mathematics, Darmstadt, Germany.
[4]Massachusetts Institute of Technology, Sloan School of Management, 77 Massachusetts Avenue, Cambridge, MA 02139, USA.

*A preliminary version of this article appeared in Berger et al.[1]

Corresponding author:
André Berger, Maastricht University, Department of Quantitative Economics, PO Box 616, 6200 MD, Maastricht, The Netherlands
Email: a.berger@maastrichtuniversity.nl

in time which trains should depart and which trains should wait. Different objective functions may be defined for this problem. In this paper we consider the problem of minimizing the total delay of all passengers.

It has been shown that even the offline version, i.e. the case when all delays are known in advance, is NP-hard even for very special networks.[2,3] However, a branch-and-bound algorithm has been developed to solve the problem optimally by using an integer programming formulation for the problem.[4]

This IP formulation is based on a model of railway delay management (RDM) that uses an event–activity network, where the nodes represent arrival or departure events, and the edges represent driving, transfer, or waiting activities. The only network for which an optimal (polynomial time) algorithm is known for the online problem is the line.[2]

Moreover, simulation models of railway networks and the corresponding operations and scheduling policies have been developed before in other contexts.[5,6,7,8,9,10,11,12]

In this paper our main focus lies on the algorithmic aspects as well as the simulation of the online delay management problem. We use the above-mentioned description of the event–activity networks in Section 2 to show that the online railway delay management (ORDM) problem is PSPACE-hard. Thus, for practical applications, heuristics have to be developed. It lies in the nature of such heuristics that their performance is really hard to measure. In particular, for a PSPACE-hard problem, it is infeasible to obtain good bounds on the optimal solution. Moreover, it is difficult to compare different heuristics due to differences in the model, the objective and the implementation. For the case of RDM, it is also important that the source delays are in some sense comparable when different heuristics are evaluated.

In order to overcome the above-mentioned problems, we have developed the simulation platform TOPSU–RDM for RDM problems, on which different heuristics can be applied to different instances and on which their performance can be evaluated. The framework and the simulation platform are described in Sections 3 and 4.

In Section 5 we present some examples of wait policies that we have implemented as well as computational results. In Section 6 we also discuss how our approach can be used for other logistics and production planning problems as well.

2. Online railway delay management is PSPACE-hard

It is widely assumed that the complexity class PSPACE is not contained in NP. In other words, there are problems in PSPACE for which there is no polynomial-time checkable certificate. If this holds, e.g., for the RDM problem, then one may not evaluate a delay management strategy in polynomial time and one could not decide for every value k in polynomial time whether a certain strategy scores in expectation better or worse than k. In general, this would also inhibit the comparison of different strategies. In the following we show that the ORDM problem is indeed PSPACE-hard. This result justifies the simulation-based evaluation of wait policies.

Definition 2.1. *The* complexity class *PSPACE is the set of all decision problems that can be decided on a deterministic Turing machine using space limited by a polynomial in the input size. A problem P is said to be PSPACE*-hard*, if there is a Karp reduction from every problem in PSPACE to P. A problem in PSPACE that is PSPACE-hard is called PSPACE*-complete.

We prove that the following simple version of the ORDM problem is already PSPACE-hard, i.e. at least as hard (by polynomial-time reduction) as any problem in PSPACE.

For the reduction we use the following PSPACE-complete problem.

Definition 2.2. *Deciding whether a logical expression of the following type is true*

$$\exists x_1 \forall x_2 \ldots \exists x_{n-1} \forall x_n : \bigwedge_i \bigvee_j z_{ij},$$

where z_{ij} are literals in the variables $\{x_1, \ldots, x_n\}$ and their negations, is called the quantified Boolean formula (QBF) *problem.*

We reduce the QBF to a simple version of the online delay management problem.

2.1. The basic online delay management problem

We present the delay management problem for this reduction on an event (nodes)–activity (arcs) network, whereas our simulation contains an infrastructure graph. It will become clear that the model used for the reduction is even slightly simpler than that of the simulation. This means that every instance of the reduction model can be described as an instance for our simulation tool. Yet, some further complicating aspects such as single tracks, which are included in the simulation, are not needed for the reduction, and thus not modeled in this section.

An instance of the *basic online delay management* (BODM) problem

$$(G, T, C, \pi, S, \mathcal{D})$$

consists of an infrastructure graph G, a set of trains T, a directed graph C with a vertex set $V(C)$ of relevant events and an arc set $A(C)$ representing precedence constraints among those events, a timetable $\pi : V(C) \to \mathbb{R}_{\geq 0}$, a set S containing functions $\tau : A(C) \to \mathbb{R}_{\geq 0}$ (which we call *scenarios*) for the minimal time distances of two events connected by a precedence constraint, and finally some mathematical object $\mathcal{D}$ expressing a cost model for the delay management.

The vertex set of C is the set of relevant events. Each relevant event is characterized by a triple (t, a, b), with $t \in T$ a train, and $a, b \in V(G) \cup E(G)$ are either vertices or edges of the infrastructure graph. The triple (t, a, b) represents the event that train t changes from infrastructure vertex (edge) a to infrastructure edge (vertex) b. The timetable entry $\pi((t, a, b))$ states the time for which this event is scheduled.

2.2. Disposition timetable

The goal is to give a non-anticipative strategy that constructs a feasible *disposition timetable* in every scenario. A disposition timetable is a vector $\pi' : V(C) \to \mathbb{R}_{\geq 0}$ that respects the timetabling condition $\pi' \geq \pi$. It is feasible in a scenario τ, if for all precedence constraints $a = (i, j) \in A(C)$ we have $\pi(j) - \pi(i) \geq \tau(a)$.

As the strategy is required to be non-anticipative, the data $\tau((x, y))$ are only available after the time when $\pi'(x)$ took place.

2.3. Cost model

Different ways to define the cost model $\mathcal{D}$ are possible. We use the following. The cost model contains a set of origin–destination pairs with a certain weight, i.e. we know how many passengers want to travel from a certain starting station to a certain final destination. Their paths through the infrastructure network G are fixed. They may follow that path on different trains, but the sequence of stations they pass is fixed. In each scenario the total delay of the passengers in π' compared with π, plus a certain fixed cost for those passengers, who will not reach their destination at all, defines the cost.

An alternative way to define the costs specifies a certain cost for each transfer that is broken and for each arrival which is delayed in π'. The two models are not equivalent, but can be translated into each other in many cases. For the reduction we use the model described above. However, we sometimes refer to the costs as the costs of breaking a transfer or delaying a train, because these terms are more convenient, and in the specific case can be translated into the original cost model.

The decision problem, which we show to be PSPACE-hard, is the following question.

Definition 2.3. *The following question is called the* BODM decision problem. *Given a BODM instance and a budget B, is there a non-anticipative strategy for constructing a disposition timetable, that achieves a cost value lower than the budget B in every realization of $\tau \in S$?*

2.4. Reduction of QBF to the BODM decision problem

For a given Boolean formula in conjunctive normal form, $\bigwedge_i \bigvee_j z_{ij}$, with literals in the set of variables $\{x_1, \ldots, x_n\}$ and their negations, we construct an instance of the BODM decision problem. Below we show that for this BODM instance there exists a strategy that achieves a cost lower than the budget B, if and only if the quantified Boolean formula, $\exists x_1 \forall x_2 \ldots \exists x_{n-1} \forall x_n : \bigwedge_i \bigvee_j z_{ij}$, is true. This will imply that the BODM decision problem is PSPACE-hard.

In our construction we use *fixed* and *non-fixed* trains. A train is fixed in the sense that delaying this train would automatically exceed the budget by yielding a cost $M_0 > B$. Nevertheless, we use fixed trains, that are *a priori* fixed to be late. Such a late fixed train has an initial delay prior to the decisions of the strategy, but may neither be delayed any further, nor does it have a buffer time to compensate for the delay. We introduce the late, fixed trains to explain transfers that are *a priori* broken, i.e. lead from an arrival (of a late, fixed train) to an earlier departure. The costs for the initial delays of those trains are constant for all further unfolding of the scenario and all disposition timetables. Therefore, we can neglect them.

The non-fixed trains fall into two different groups. Each train of the first group, the *variable trains*, corresponds to a variable x_i of the Boolean formula. If such a train is delayed, we interpret the corresponding variable x_i as being false, and true if the train is on time. The trains of the second group are called *modeling trains*, as they serve some technical purpose in the reduction. They can also be delayed or run on time while the strategy is carried out. However, the reduction will be constructed such that their delay is entirely dependent on the delay of the variable trains.

For modeling reasons we want that for every non-fixed train the decision about running delayed or on time must be taken at the start of the train‘s ride and kept until the final destination. We enforce this by an incoming transfer from a late, fixed train at the

beginning of the ride and an outgoing transfer to an on time, fixed train at the end. The non-fixed train cannot meet both transfers. Thus, it always incurs the cost for breaking one of these transfers, M_1. Let m be the number of non-fixed trains, then the total budget $M_1 m + C < B < M_1(m+1) + C$ is set such that none of these trains may break both transfers. (The constant C is the constant cost of all *gadgets*, as explained below.)

With these ingredients (on-time fixed trains, late fixed trains, variable trains, modeling trains, and the rule that any of the latter two types of trains must be scheduled either late in the whole disposition timetable or on time in the whole disposition timetable) below we devise a gadget for a logical NON operator and a gadget for a logical, multiple AND operator. The NON gadget will yield that a certain modeling train is delayed if and only if a certain other train is on time. The AND gadget has an out train that is on time, if and only if all trains of a certain set are on time. Before we describe the mechanism of these gadgets, we first show how they are used to reduce the QBF, $\exists x_1 \forall x_2 \ldots \exists x_{n-1} \forall x_n : \bigwedge_i \bigvee_j z_{ij}$. Actually, we use an alternative way to write the Boolean formula namely, $\bigwedge_i \bigvee_j z_{ij} = \bigwedge_i \neg(\bigwedge_j \neg z_{ij})$.

The time horizon of the constructed BODM instance is split into five phases (cf. Figure 1).

1. In the first phase all decisions about whether a variable train is delayed or not are taken one after the other. The incoming transfer from the fixed train determines the order by which these decisions must be taken. Recall, that because of the transfers to the fixed trains, these decision cannot be changed later. In this way we reflect the consecutive mechanism in the QBF. For those variable trains that correspond to an all-quantified x_i the decision whether they run late or on time, shall be taken by the adversary. The scenario set S is restricted such that the adversary may produce exactly one of the two following situations: delay the train i before the incoming transfer from a late, fixed train at the beginning of the ride of train i, or (exclusive or) delay train i immediately before the outgoing transfer to a punctual, fixed train at the end of the ride of train i.
   In principal, the dispatcher can also delay a train of an all-quantified variable. However, this would allow the adversary to delay this train a second time and thereby exceed the cost limit. Therefore, we can assume that the dispatcher will never make an all-quantified train wait.
2. Then each variable train runs through a NON gadget, producing its negated train, which is late if and only if the variable train was on time.
3. The variable trains and their negations pass through the AND gadgets for each of the re-written clauses. A re-written clause i, $\bigwedge_j \neg z_{ij}$, is modeled by an AND gadget with those variable trains as in trains that correspond to the negations of the literals in the original clause.
4. Each out train of those AND gadgets is negated.

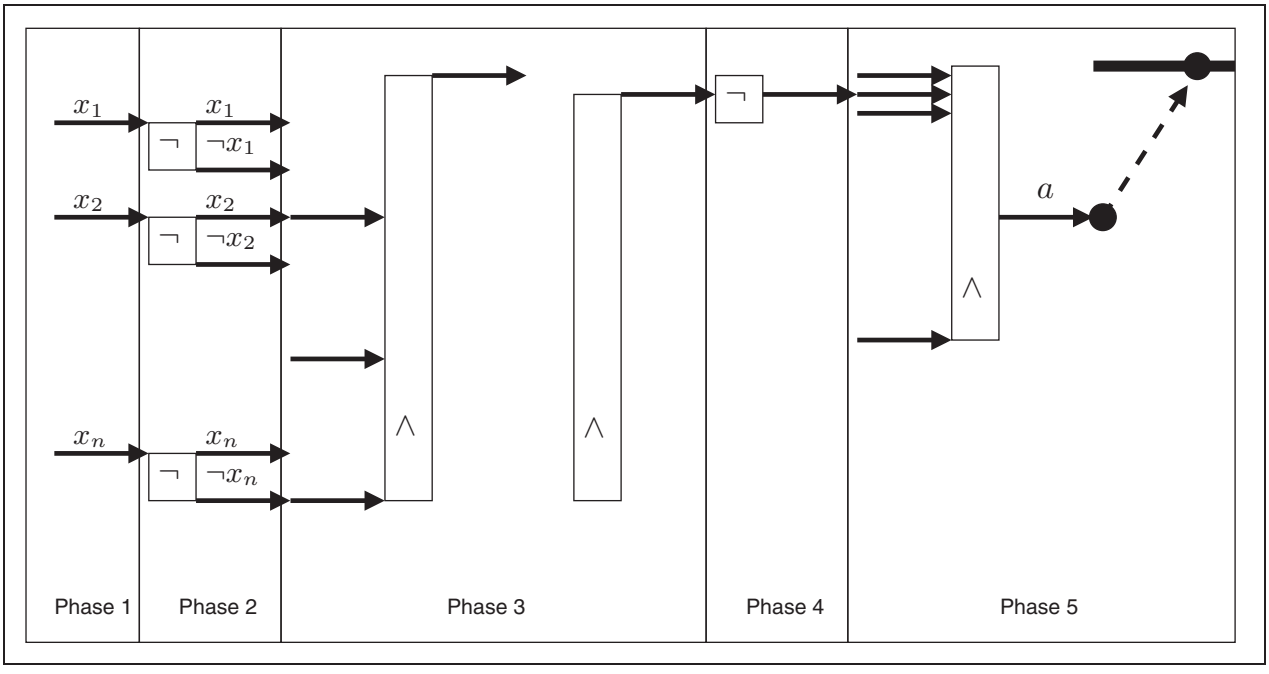

Figure 1. Using the gadgets.

5. Finally all of those negations enter the central AND gadget. The out train a of this gadget has a tight transfer to a fixed train. If that out train is late, it yields a cost of M_2.

We make sure that every AND and NON gadget yields a fixed cost in every scenario. Thus, we can choose B and M_2 such that the total cost is below B, if the train a is on time, and the cost exceeds B, if a is late. In this way the BODM instance is feasible with cost limit B, if and only if the quantified Boolean formula is true. This completes the reduction.

2.5. The NON gadget

The initial state of a NON gadget is depicted in Figure 2. There is a fixed train (drawn as a bold line) and two non-fixed trains. The lower of the non-fixed trains, the in-train is always late for its transfer to the other non-fixed train. We draw a rhombus to symbolize some fixed delay that should explain this fact. The upper train can wait for the lower train and thus keep the connection (Figure 4). But, if the lower train is additionally delayed before the rhombus, the upper train would have to wait so long, that is has to break a transfer to a fixed train. The costs for breaking this transfer are M_0, i.e., would immediately exceed the budget. Thus, the strategy will break the transfer from the in-train to the out-train, and the latter will leave the gadget on time (Figure 3). A delayed in-train yields an on time out-train, and vice versa.

Still the gadget would not work, because we cannot guarantee that the transfer is not broken if the in train is on time or the out train is delayed although it breaks the transfer from a late in train. To exclude these cases, we have to make sure that a NON gadget yields a fixed cost in both dispositions we desire (in train on time and out train late and vice versa), and exceeds this cost in any other disposition. To this end, let c_w be the cost of delaying the out train, c_b the cost of breaking the transfer from the in train, and $c_b - c_w = c_g$ a positive number. We introduce an *a priori* broken transfer from a late, fixed train to the in train, which to break costs c_g. This transfer is broken, if and only if the in train is on time. In other words, the desired dispositions are the only two dispositions by which the gadget has a cost of less than or equal to c_b, and those yield cost equal to c_b.

Note that we use NON gadgets, which output the out train and the in train, and NON gadgets that only output the out train.

2.6. The AND gadget

The AND gadget is fairly simple: all in-trains have to be on time for the out train to be on time. Therefore, all in trains have a tight connection of breaking cost M_0 to the out train. Again, we have to make sure that the out train is not scheduled late although all in trains are on time. To this end, all in trains run along the same track for some distance. There are some passengers that want to travel this distance, but come from a late, fixed train. Only if at least one of the in trains is late, these passengers will reach their destination. The cost c_h of not serving these passengers equals the cost of delaying the out train. Thus, the gadget at least costs c_h, and

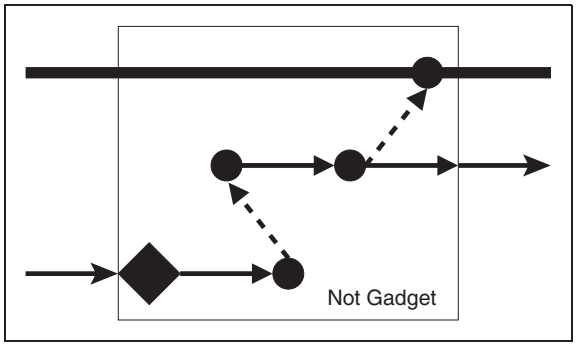

Figure 2. The initial situation.

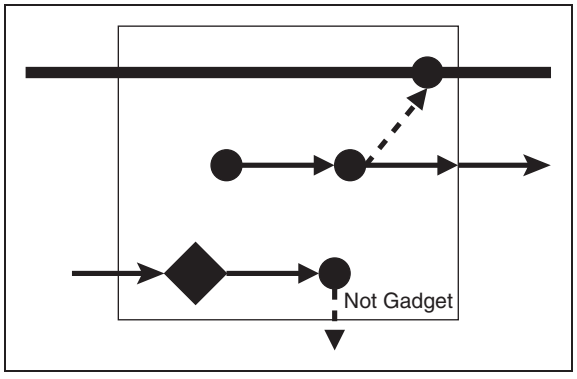

Figure 3. Delayed yields on time.

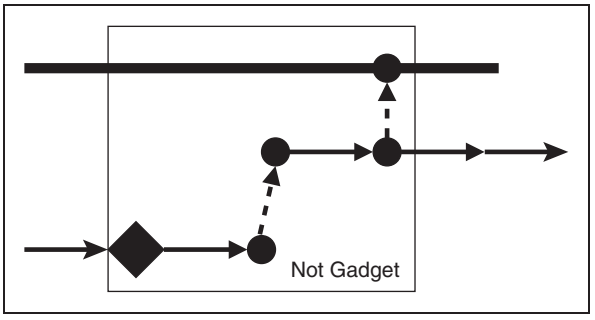

Figure 4. On time yields delayed.

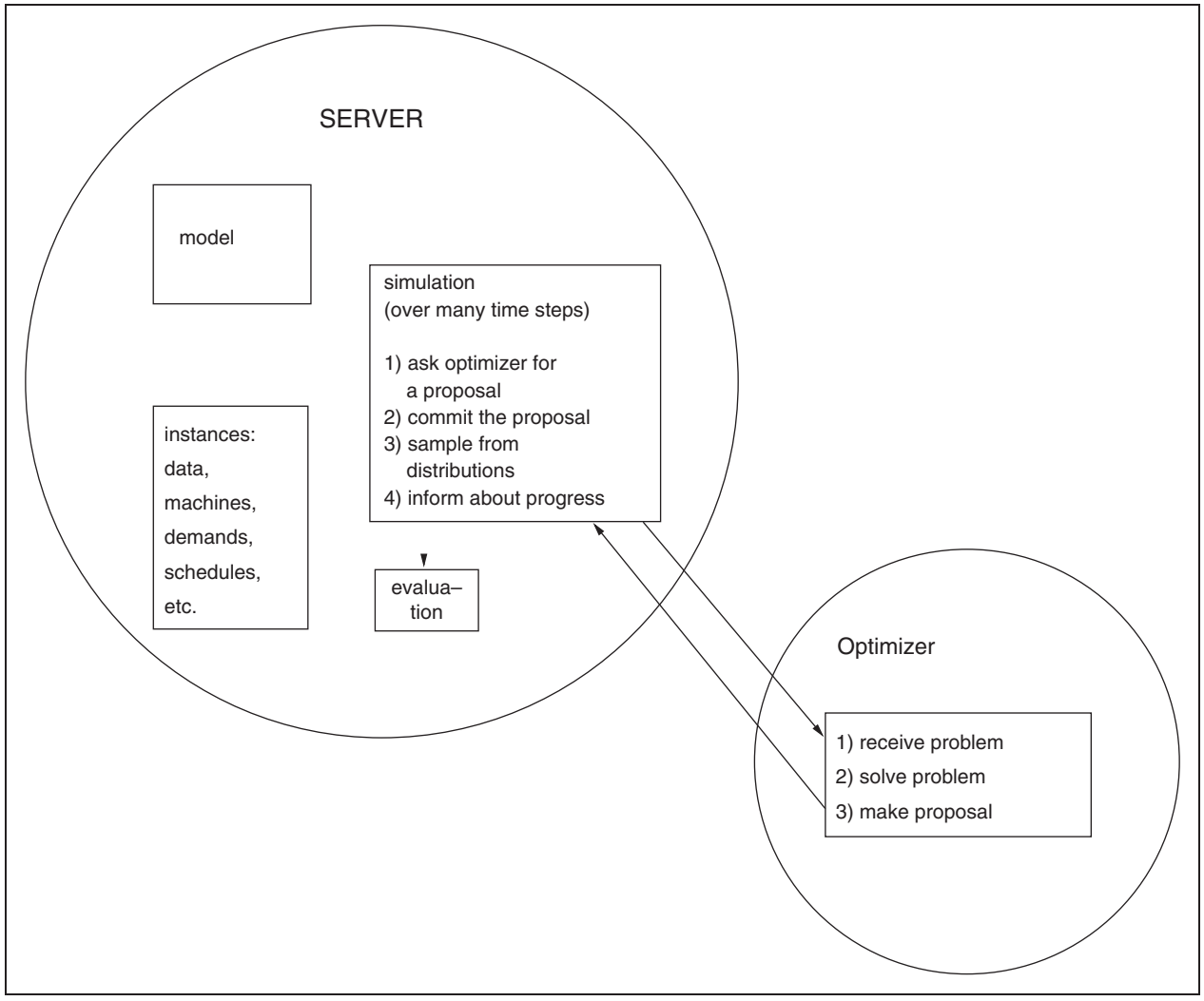

Figure 5. The TOPSU idea: splitting the tasks.

will exceed this cost in cases where the out train is late even though all in trains are punctual.

3. The TOPSU framework

TOPSU–RDM has been developed within the TOPSU framework (Tournaments for Optimal Planning and Control under Uncertainty[1]). TOPSU is an interactive framework for optimal planning and control of production or other control processes under uncertainty. It divides an optimization task into three parts: model building, an algorithm for solving the problem, which is induced from the model, and the experimental evaluation of the algorithm inside the model (cf. Figure 5).

One most crucial point of TOPSU is that the framework does not only demand this partition, but also allows the distribution of these three tasks to different people. Thus, it can be considered as a platform for the competition of algorithms. The second crucial point is the fact that TOPSU supports the influence of uncertainty within its implicit optimization model. We decided to incorporate this feature for two reasons. First, practitioners often claim that production processes have massively to deal with several kinds of uncertainty. Second, production systems are typically so large that optimization must focus on a certain part of this system or, in other words, we have to optimize parts of a supply chain. We think that it will be advantageous for the optimization of a supply chain if its components are aware of uncertain boundaries.

Technically, TOPSU is realized with the help of the Internet. On one site, we have a so-called server, where the ultimate simulation proceeds. On this server, all necessary data are available for download. The control of the simulation, however, is remotely executed at the so-called clients.

[1]TOPSU is the abbreviation of the German translation.

The field of optimization under uncertainty, especially with probability-based uncertainties, is a fast-growing and increasingly important area. Just think of planning tasks for railways or aircraft, and remember your own experiences with disruptions. Disruptions reflect the fact that, at planning time, not all information is available. Optimization problems, considering these uncertainties, often become PSPACE-hard. In some cases it will be possible to extract easy subproblems which can be solved in polynomial time, and that simultaneously serve as good solutions for the real-world application. However, if this is not possible, simulation experiments promise a lot of gain in insights. Experimental work for the evaluation of optimization processes often has the disadvantage that results of different authors cannot be compared with each other. One reason is that each author examines slightly modified problems, and another reason is that measuring is not standardized. In TOPSU, the methodical tasks ''finding a problem for examination'', ''algorithms and heuristics'', and ''measuring'' are split to different persons. Moreover ''finding a problem for examination'' and ''measuring'' are centralized. That increases comparability and, simultaneously, the credibility of the experiments.

We now give a brief overview of the contents of the paper. In Section 4 we present the model and the details of the RDM simulation platform. We show that ORDM is PSPACE-hard in Section 2, and in Section 5 we present our computational results. In Section 6 we consider a more generic modeling tool and we discuss future work and give some conclusions in Section 7.

4. The simulation platform

In this section we describe in more detail the model that is used in our simulation and the specifics of the simulation. The simulation platform consists of three parts: a *server* program that implements the model, a program that implements the wait policy (called an *engine*), and a *graphical user interface* (GUI) which enables the communication between the server and the engine and which provides a visualization of the simulation.

4.1. The server

We start with a description of the model that is implemented on the server. In our model stations are the nodes and tracks are the edges of a directed graph on which the trains can move. Physical tracks that can be used in both directions are modeled as two distinct directed edges, and the server makes sure that only one of these two edges is used at any time. Each station and each track has a capacity, the maximum number of trains that can be in that station or on that track, respectively, at any time. Moreover, each edge has a timeslack, i.e. the minimum time that has to pass between two trains entering or leaving that track. The edges have the first in first out (FIFO) property, i.e. the trains leave an edge in the same order that they entered that edge. There is also a minimum halt time in the stations, the minimum time that trains have to stay in a station before they continue their scheduled route.

In addition to stations and tracks, the server has information about the trains, the schedule, and the passengers. Each entry in the schedule consists of a train, an edge, a departure time, an arrival time, and a pointer to the delay distribution for this entry. Passenger flows are called origin–destination pairs, each having a weight (corresponding to the number of passengers), a start time, and a list of edges that these passengers are going to traverse during their travel. Note that rerouting of trains and passengers is not allowed in our model. In addition, there is a (global) constant, the minimum change time, which is the time needed for a passenger to transfer between one train and another. It is assumed, that each passenger always uses the first train going towards his/her next intermediate destination. This may be the train he/she is currently in, or another train that is heading in the same direction and not leaving before the minimum change time has passed.

The simulation running on the server is discrete time and event based. During initialization, for each train, an event for its first entry in the schedule is created and inserted into a (time-sorted) priority event queue. As the trains move along the edges, events will be taken out from the event queue and new events will be generated. An event consists of a time, a train, an edge, and an indicator whether this event means the train wants to enter or leave that edge at the specified time. The server will then run the following loop until the end of the schedule has been reached:

1. Collect queries at current time from the event queue.
2. Send a query message to the engine.
3. Receive a result message from the engine.
4. Commit the answers from the result message, if feasible.
5. Sample delays for trains that in fact have left a station.
6. Send message about committed decisions and sampled delays to the engine.

We now describe some details of the points above.

Collect queries. In this step all events from the top of the event queue, whose event time is equal to the current simulation time, are checked for feasibility and inserted into a query collection that will be sent to the engine.

This means that no query is generated for an event which cannot be implemented at that time, e.g. a train wants to enter an edge that is full (i.e. the number of trains on that edge equals the edge's capacity). In this case, a new event for that train is created at the earliest possible time at which the "infeasibility reason" may disappear, e.g. the time of the next event of the first train on that edge.

Sending queries and receiving results. All queries that have been collected in the previous step will be sent to the engine in a single message. The simulation on the server stops until a corresponding result message is received from the engine.

Committing decisions. Similar to the method used while collecting queries, only those queries will be committed which are feasible and were answered positively by the engine. Whenever a query is committed, e.g. a train is entering a track, a new event for that train to leave the edge is created at the expected arrival time: the scheduled travel time plus the sampled delay. If all queries were denied by the engine, the simulation jumps to the next point in time in the event queue, and the events corresponding to the current queries are postponed to that time.

Sampling delays. Delays are sampled whenever a train actually leaves a station. A delay is sampled from the distribution linked to the corresponding entry in the schedule and added to the travel time for that train on that edge.

Sending information to the engine. After committing the decisions and sampling the delays, a message is sent to the engine to inform about the decisions and the delays.

The actual arrival and departure times are stored during the simulation. After the end of the simulation, the objective value of the simulation is computed. This is actually the only time when the passenger data is used. For each origin–destination pair the actual travel time is computed and the scheduled travel time is subtracted. The sum of these (weighted) delays is the objective value. It may happen, that a passenger does not reach his/her final destination due to large delays or a bad wait policy. In this case, the actual travel time is replaced by a large constant (e.g. the costs to pay for accommodation for that passenger).

For each instance and each user that runs an engine on that instance, an entry is written to the highscore list of that instance. For several runs by the same user on the same instance, average scores can be seen in the GUI.

4.2. The graphical user interface

The GUI (cf. Figure 6) enables the communication between the server and the engine. It is also used to connect engine and server, display highscores, and for visualizing the network and the simulation.

The steps in using the GUI to run a simulation are as follows:

- Login (username and password can be obtained from the authors).
- Choosing RDM as the "game".
- Choosing an instance.
- Getting the pre-specified parameters.
- Connecting an engine.
- Starting a simulation.

If necessary and recognized by the engine, parameters can be passed to the engine via the GUI. The user can also specify a subinterval of the pre-defined timeframe on which the simulation should run, or change parameters such as the cost that is added to the objective for passengers who do not reach their final destination. However, highscores will be only written when the full timeframe with the pre-specified parameters is simulated.

In the visualization panel (cf. Figure 7) of the GUI, the user can stop and continue the simulation, and proceed stepwise. This may be helpful for the analysis, at least for smaller instances.

4.3. The engines

An engine for RDM, i.e. a program implementing a certain wait policy, basically just has to say yes or no to the queries sent by the server. It may do so, of course, without keeping any information about the network, the schedule, or the passengers. "Intelligent" engines, however, will need such information.

This may be information such as previous arrivals and departures, the current location of a train, or the next event of a train in the event queue on the server. For algorithm/wait policy developers, a set of Java classes is available that take care of keeping up to date all of the necessary data during a simulation. An engine can use these classes and just has to implement the method that determines the answers to the queries posed by the server. Instructions to implement an engine are available at the TOPSU–RDM Webpage.[13] Sample engines that implement the "always yes", "wait for all connections" and "wait randomly" are available for testing purposes.

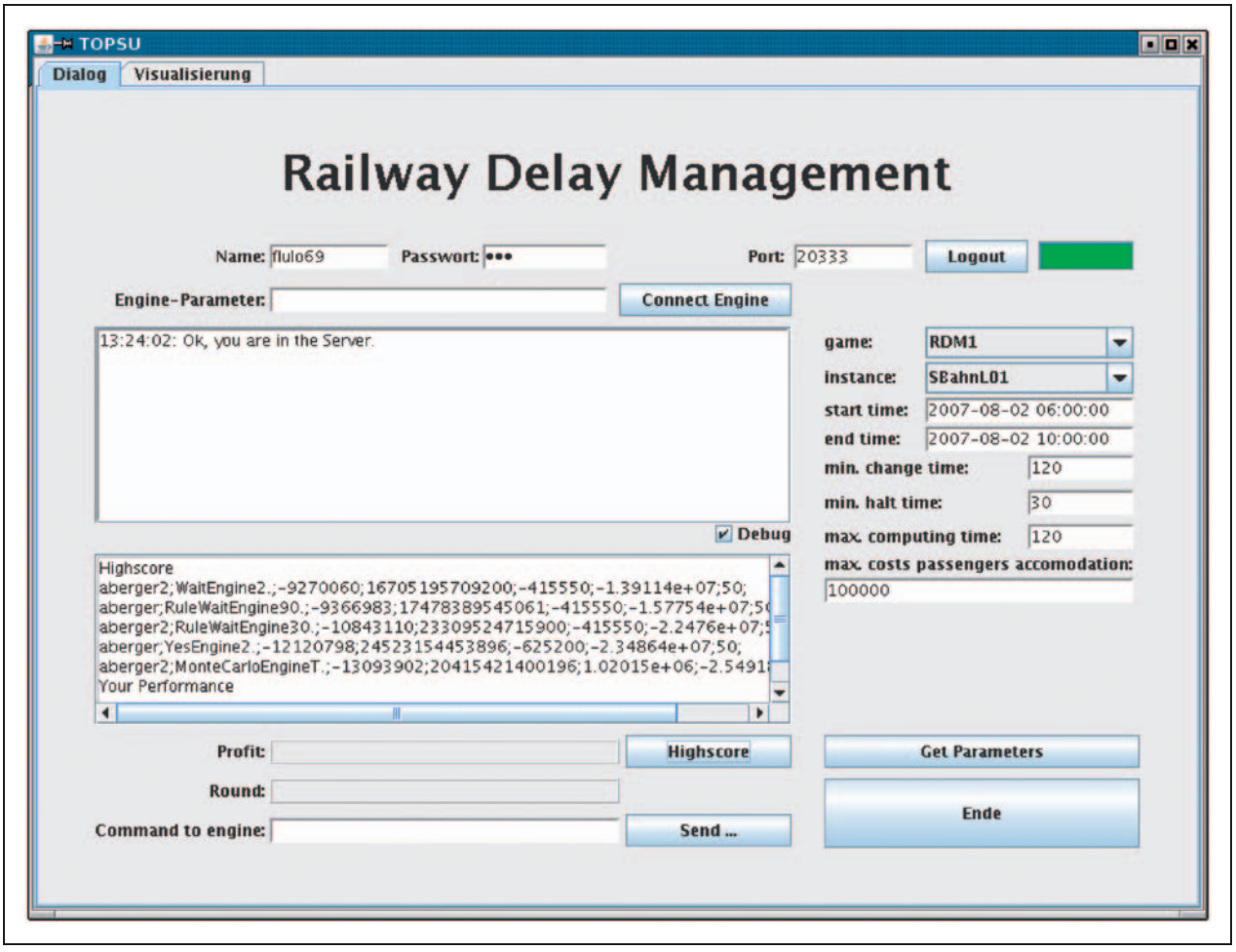

Figure 6. The graphical user interface of TOPSU–RDM.

5. Computational results

In this section we report our computational results. We consider an instance derived from a simplified network of the Berlin S-Bahn and a schedule running from 06:00 until 10:00. The instance contains 37 trains, 24 stations, 60 edges, 782 entries in the schedule, and 378 origin–destination pairs. The origin–destination pairs and the probability distributions for the delays are estimates and resemble realistic values.

We have run the simulation 50 times for each of the tested wait policies. In the following we describe the wait policies whose performance we have tested (our implemented wait policies are called *engines*):

1. YesEngine (YE): this engine answers "YES" to all wait or leave questions posed by the server.
2. WaitEngine (WE): this engine initially computes all connections for all origin–destination pairs. During the course of the simulation, it only answers "YES".
3. RuleWaitEngineX (RWEX): this engine does the same as the WaitEngine, except it says "YES" to a leave query if the train is currently already X seconds late, even if not all connecting trains have arrived at the current station yet.
4. MonteCarloEngine (MCE): a wait policy based on ideas from Monte Carlo tree search, alpha–beta pruning and bounded lookahead. It considers the wait decision problem as a game between the *decision maker* and the *delay maker*. The quality of the answer mainly depends on the number of possible nodes in the game tree that the algorithm can visit in a certain amount of allowed time, i.e. the more time we allow the engine to compute its answer, the better the answer should be.

The results depend heavily on the penalty that is imposed for passengers who do not make it to their final destination within the given timeframe due to bad wait decisions. In a future version it may therefore be useful to give a multi-criteria score for each run of

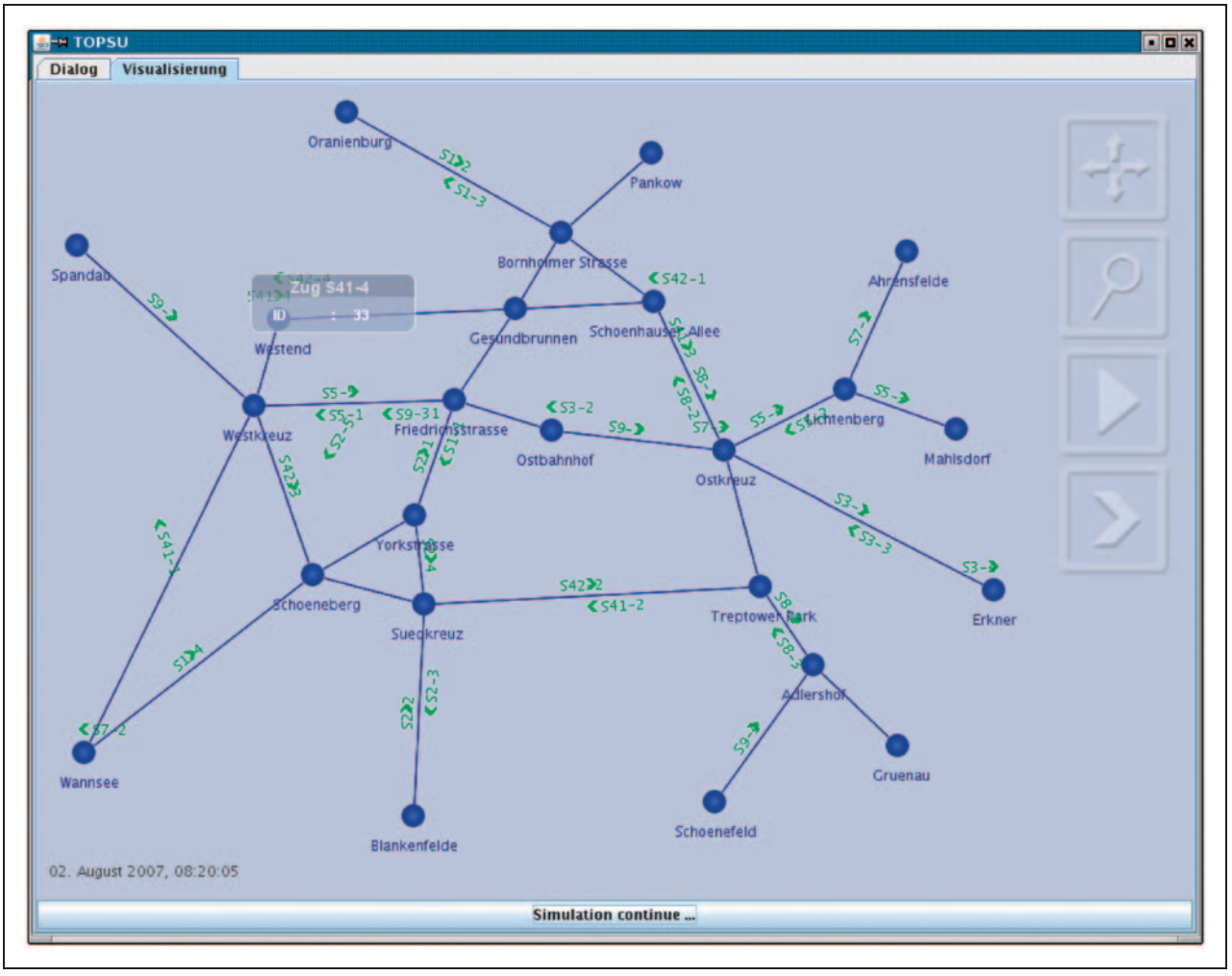

Figure 7. A visualization of the simulation for a (simplified) Berlin S-Bahn schedule.

a wait policy, e.g. the total delay of all arrived passengers and the number of "stranded" passengers.

Tables 1, 2 and 3 show the results when this penalty is either 1000, 10,000 or 100,000, respectively. The results are normalized so that in each run the YesEngine has an objective value of 100%. The first line in each table shows the average performance of the other engines, and the bottom shows how often each engine was ranked first, second, etc., among all five engines over the 50 runs (the different number of total first and second places is due to ties).

It can be seen that the most trivial wait policy, the YesEngine, still performs best if the penalty is very low. However, more realistically we should assume that this penalty is rather high, as this would imply that a good wait policy should in any case try to avoid stranded passengers. For the highest tested penalty (cf. Table 3), it turns out that the WaitEngine, which ensures no passengers miss any connection, performs best on average, as well as ranking first most often.

Table 1. Penalty 1000

	YE	WE	RWE30	RWE90	MCE
Objective	100%	114%	102%	113%	229%
1st place	18	25	19	26	0
2nd place	3	11	11	10	1
3rd place	10	13	12	13	0
4th place	17	1	7	0	3
5th place	2	0	1	1	46

This shows that considerable effort should be put into an intelligent wait policy if it should perform better than any of the rather trivial wait policies. Note that the first four wait policies consider neither the passenger routes nor the delay probabilities when making their decision.

In contrast, the MonteCarloEngine does use these parameters. Owing to its complexity the time it needs

to answer a query is quite long. In our experiments we did not give the engine much time to find an answer to a query, and on average, the objective value of the MonteCarloEngine is three times as bad as that of the WaitEngine. However, as can be seen from Table 3, it performs best on 12 of the 50 days. This is very promising, and we hope that when this engine is developed further, it will be able to compete with or even beat the other more simple wait policies.

Our data do not show that one of our engines is significantly superior to another. Such a statement, however, is not necessarily the intention of our transparent, tournament-driven environment. Instead, the transparent tournaments have the task to help distinguishing good from bad procedures, when statistics fail or developers are ought to have no chance for manipulation. The probability spaces of our problems are enormously large, especially due to the interplay of random events and active decisions.

Table 2. Penalty 10,000

	YE	WE	RWE30	RWE90	MCE
Objective	100%	85%	92%	85%	143%
1st place	4	45	15	44	0
2nd place	0	2	4	2	0
3rd place	11	3	26	4	0
4th place	33	0	5	0	2
5th place	2	0	0	0	48

Table 3. Penalty 100,000

	YE	WE	RWE30	RWE90	MCE
Objective	100%	77%	89%	78%	225%
1st place	1	35	9	34	12
2nd place	3	12	8	11	1
3rd place	9	1	19	3	6
4th place	20	2	13	2	3
5th place	17	0	1	0	28

Nevertheless, descriptive statistics can give us additional information. For example, the Friedman test is a non-parametric statistical test which is based on variance rank analysis. Its task is to compare several non-independent samples concerning their error of central tendency. If we compare the 50 samples that we have available for each of our engines (for the case where the penalty is 100,000), the p-value of the Friedman test is nearly zero. Thus, it is nearly certain that the distributions over those random variables which show the profitability of the different engines, are different. This at least means that the different engines do not have the same profitability.

The reason that we chose a non-parametric test is due to the fact that our data are not normally distributed, as the quantile plot for the WaitEngine shows (cf. Figure 8).

The box plot in Figure 8 shows medians inside boxes which contain half of all sample points. There is one box for each of the five engines. Also from this perspective, the WaitEngine has an advantage over the other engines.

Last, but not least, the scatter plots in Figure 9 show that the YesEngine, the WaitEngine and the WaitEngine30 have statistical similarities, i.e. their outputs are linearly correlated. In contrast to this

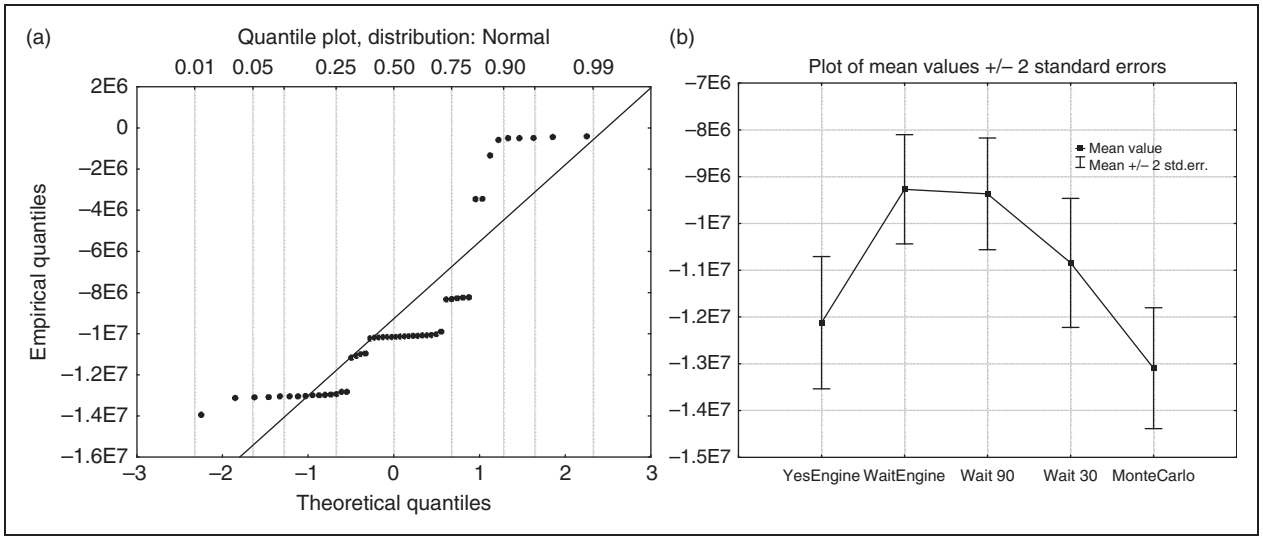

Figure 8. Quantile plot for the WaitEngine (left) and box plot for average values (right).

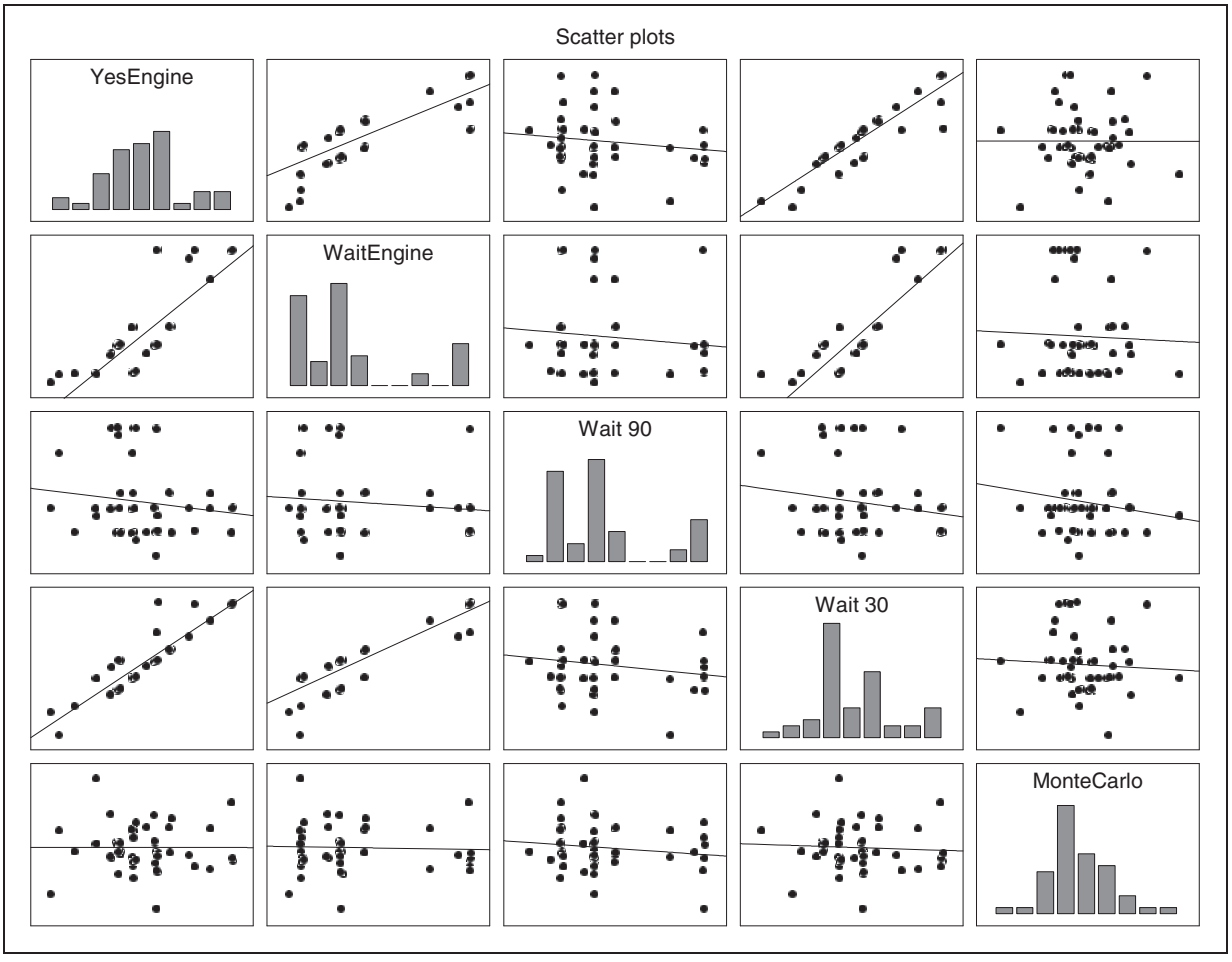

Figure 9. Scatterplot matrix.

observation, the MonteCarlo engine and the WaitEngine90 are not correlated with the other three engines.

6. A scheduling and planning simulation generator

Currently, we are working on a generalization of the RDM simulation. Many production, transportation and operations problems resemble some similarities with the delay management problem. It is often the case that some *items* (e.g. trains, parts, goods, etc.) move along a *network* (e.g. railway tracks, road networks, conveyor belts in a production facility) and are processed in some *stationary objects* (e.g. stations, warehouses, machines, etc.).

Our goal is to build a simulation generator that can be used for a wide variety of planning problems occurring in practice. We restrict our efforts to optimization problems, where some passive objects pass some other active objects. We think that a proper entity–relationship description, plus some extra information including message layout between clients and server, suffices to automatically construct a generic simulation block, where the messaging as well as the basic event handling of the simulation are included.

We call a specific planning or production problem in this context a *game*. Although the simulation of a new game on the game server contains only a fair number of lines of code, we had to accept that developing a new simulation kernel is quite a difficult task. Therefore, we are developing TOPSU2.0, which is an extension of the described concept. With TOPSU2.0 we go one step further to more modeling and less programming. Indeed, we are developing a simulation generator for TOPSU.

First, the main idea is that we are mainly interested in more or less classic flow settings. There may be trains which pass stations and go on tracks, or there may be items which pass machines and become products, etc. Many such problems can be modeled with the help of some ''active objects'' such as stations or machines and some ''items'' such as product items or trains. The items move trough the system of active objects in some way,

and a movement from one active object to another is called a transition. After all, for modeling we need mainly SQL tables which describe the active objects, the items, and some meta-information. A METATABLE holds the information, what the main data structure of an object is (e.g. multiset, FIFO etc.), what the role of the object is (e.g. container, item, none, directed FIFO edge), and how it can be identified. A transition describes which item can move from one active object to another. Moreover, we standardized some tables, such as the description of probability distributions.

Second, we have to specify the messages between server, client and GUI. We introduced a simple textfile format for this purpose. Our simulation generator takes the SQL database and the text description of messages as input and generates a simple engine which produces solutions, and the necessary program code for all messaging between GUI, server and engine, and a simulation module which can be easily plugged into the server software. Additional, hand-made, source code can be added to the generated class code such that we can model delays in machines etc.

We hope that this approach will simplify the implementation of other simulation environments for similar planning processes, and that it will underscore the usefulness of the TOPSU concept.

7. Future work

7.1. Refinement of the model

The following refinements can be made to the model to improve the applicability of the simulation tool and to move the model closer to practice. First, different objectives should be implemented and should be made available to the users in the GUI. This could be, for example, minimizing the maximum delay of all passengers, or minimizing the number of missed connections.

Moreover, the minimum change times of passengers do depend on the station where a passenger switches between trains and may also depend on the passengers themselves. Similarly, the halt times in a station may differ during peak periods and may also depend on the train and on the station. Another refinement to the model would also be to model the tracks and their respective capacities inside a station.

7.2. Improving the wait policies

The main algorithmic challenge and one of our future goals is to develop a wait policy that can reduce the delays occurring in actual railway networks. We believe that our approach used in the MonteCarloEngine is very promising. However, other ideas and heuristics may also be exploited, and using our simulation framework it will possible to easily and objectively evaluate and compare new wait policies for the Railway Delay Management problem.

8. Conclusions

In this paper we have presented TOPSU–RDM, a simulation platform for the ORDM problem. We also showed that this problem is PSPACE-hard, justifying that a simulation approach is suitable to verify the performance of different wait policies. Our approach is implemented within a framework that allows a fair and non-manipulable comparison of the wait policies. In particular, we ensure that the initial delays (which are based on probability distributions) that the wait policies have to deal with, are identical for all wait policies.

We also provided results of preliminary computational studies, indicating that with some effort we can improve upon the performance of some trivial wait policies. Moreover, we discussed several extensions for further usage of the ideas and methods presented here for further research in simulating real-world production and operation processes with uncertainties.

Funding

The first three authors have been partially supported by the European Regional Development Fund (ERDF). Sebastian Stiller's work was partially supported by the ARRIVAL project, within the 6th Framework Programme of the European Commission under contract no. FP6-021235-2.

Acknowledgments

We would like to thank two anonymous referees for their helpful comments that greatly improved the presentation of this paper. We would also like to thank Robert Pankrath for developing the visualization for TOPSU–RDM.

References

1. Berger A, Hoffmann R, Lorenz U, Stiller S. TOPSU–RDM—a simulation platform for online railway delay management. In *Simutools '08: Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems and Workshops*. Brussels: Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, 2008, pp. 1–8.
2. Gatto M, Glaus B, Jacob R, Peeters L, Widmayer P. Railway delay management: Exploring its algorithmic complexity. In: *Algorithm Theory—Proceedings SWAT 2004 (Lecture Notes in Computer Science*. Vol. 3111). Berlin: Springer, 2004, pp. 199–211.

3. Gatto M, Jacob R, Peeters L, Schöbel A. The computational complexity of delay management. In: Kratsch D (ed.), *Graph-Theoretic Concepts in Computer Science: 31st International Workshop (WG 2005) (Lecture Notes in Computer Science*. Vol. 3787), Berlin: Springer, 2005.
4. Schöbel A. Integer programming approaches for solving the delay management problem. In: Geraets F, Kroon L, Schoebel A, Wagner D, Zaroliagiis C (eds), *Algorithmic Methods for Railway Optimization (Lecture Notes in Computer Science*. Vol. 4359). Berlin: Springer, 2007, pp. 145–170.
5. Chundi M, Zhao W, Tianyuan X. A platform for simulation of railway network operation scheduling. In *1997 IEEE International Conference on Intelligent Processing Systems, 1997 (ICIPS'97)*, October 1997, Vol. 2, pp. 1342–1346.
6. Rizzoli AE, Fornara N and Gambardella LM. A simulation tool for combined rail/road transport in intermodal terminals. *Math Comput Simul* 2002; 59: 57–71.
7. Vromans MJCM, Dekker R and Kroon LG. Reliability and heterogeneity of railway services. *Eur J Operat Res* 2006; 172: 647–665.
8. Sharma G, Asthana RGS and Goel S. A knowledge-based simulation approach (K-SIM) for train operation and planning. *SIMULATION* 1994; 62: 381–391.
9. Paolucci M and Pesenti R. An object-oriented approach to discrete-event simulation applied to underground railway systems. *SIMULATION* 1999; 72: 372–383.
10. Dessouky MM and Leachman RC. A simulation modeling methodology for analyzing large complex rail networks. *SIMULATION* 1995; 65: 131–142.
11. Kesling GD and Whittaker IC. A simulation model of railroad reliability. *SIMULATION* 1985; 44: 168–180.
12. Eichler J and Turnheim A. A combined simulation of a rapid transit system. *SIMULATION* 1978; 30: 155–167.
13. Berger A, Hoffmann R, Lorenz U and Stiller S. *TOPSU–RDM—A Web-based simulation for railway delay management*. http://wwwcs.uni-paderborn.de/cs/ag-monien/PERSONAL/FLULO/PP/TOPSU_RDM1.html

André Berger is Assistant Professor at Maastricht University, Department of Quantitative Economics, Maastricht, The Netherlands.

Ralf Hoffmann is System Administrator at Technische Universität Berlin, Department of Mathematics, Berlin, Germany.

Ulf Lorenz is Privatdozent (Adjunct Professor) at Technische Universität Darmstadt, Department of Mathematics, Darmstadt, Germany.

Sebastian Stiller is currently a Marie Curie Fellow at the Massachusetts Institute of Technology, Sloan School of Management, Cambridge, USA.