A Constraint-Based Interactive Train Rescheduling Tool

C.K. Chiu, C.M. Chou, J.H.M. Lee, H.F. Leung, and Y.W. Leung

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
Email: {ckchiu,cmchou,jlee,lhf,ywleung}@cs.cuhk.edu.hk

Abstract. In this paper, we formulate train rescheduling as constraint satisfaction problem and describe a constraint propagation approach to tackle it. Algorithms for timetable verifications and train rescheduling are designed under a coherent framework. We define two optimality criteria that correspond to minimizing passenger delay and the number of station visit modifications respectively for rescheduling. Two heuristics are then proposed to speed up and direct the search towards the optimal solutions. The feasibility of our proposed algorithms and heuristics are confirmed with experimentation using real-life data.

Keywords: Rescheduling, Constraint Propagation, Variable and Value Ordering

1 Introduction

The **PRaCoSy** (**P**eople's Republic of China **Ra**ilway **Co**mputing **Sy**stem) project [10] is undertaken by the International Institute for Software Technology, United Nations University (UNU/IIST). The aim of the project is to develop skills in software engineering for automation in the Chinese Railways. A specific goal of the project is the automation of the preparation and updating of the *running map*[1], for dispatching trains along the 600 kilometer long railway line between Zhengzhou and Wuhan in the People's Republic of China. The Zhengzhou to Wuhan section has been chosen as a case study because it is along the busy Beijing-Guangzhou line, the arterial north-south railway in China. The rate of running trains, both goods and passengers, of this section is high and present management procedures are not adequate with the dramatic development of domestic economy.

A running map [9] contains information regarding the topology of the railway, train number and classification, arrival and departure time of trains at each station, arrival and departure paths, *etc*. A computerized running map tool should read in stations and lines definition from a descriptor file, allow segments (subsets of all stations) and time intervals to be defined, allow train timetable to

[1] A running map is a method of monitoring the movement of trains and rescheduling their arrivals and departures to satisfy operational constraints.

be read, and finally display graphically the projection of the timetable against a given segment and a given interval. A sample running map is shown in figure 6. Train dispatchers, users of the tool, have to modify the timetable when trains in some sections cannot run according to the map, possibly due to accidents and/or train delays. The modification to the map should be performed in such a way that certain *scheduling rules* (laid down by the local railway bureau) are not violated. Therefore, a computer running map tool should check users' modifications against possible violation of scheduling rules, and warn users of such violations. In addition, the tool should also assist the user in repairing, either automatically or semi-automatically, an infeasible timetable so that the least train service disruption is made. We call this process *rescheduling*. Scheduling and rescheduling are different in two aspects. First, while scheduling creates a timetable from scratch, rescheduling assumes a feasible timetable and user modifications, which may introduce inconsistencies to the timetable, as input. Second, optimality criteria used in scheduling, such as minimum operating cost, are usually defined in the absolute sense. In rescheduling, however, the quality of the output is measured with respect to the original timetable.

The **PRaCoSy** project has resulted in a running map tool capable of train timetable verification [7]. Our task at hand is to enhance the **PRaCoSy** tool to perform automatic rescheduling, which can be considered as constraint resatisfaction. A major problem with the **PRaCoSy** implementation is that constraints are used only passively to test possible violation of scheduling rules. In view of this limitation, we decided to re-create the running map tool from scratch using a constraint programming approach. In this paper, we give algorithms for timetable verification and train rescheduling used in our tool, and show that constraint programming allows us to perform constraint checking and solving (or propagation) in a coherent framework. We study two notions of optimality for the rescheduled timetable with respect to the original timetable. These notions provide a measure of the quality of the rescheduling operation. We also present two heuristics that direct and speed up the search towards optimal rescheduled timetables.

The rest of the paper is organized as follows. Section 2 defines basic terminology and discusses related work. Section 3 explains the timetable verification algorithm. In section 4, we show how to formulate train rescheduling as a constraint satisfaction problem and give an associated algorithm. We also discuss two heuristics that help to direct and speed up the search towards optimal solutions. In section 5, we describe our prototype implementation and sample runs of the tool. We summarize our contribution and shed light on future work in section 6.

2 Preliminaries

In the following, we provide informal definitions of necessary terminology according to [9] to facilitate subsequent discussions. Names not defined should be self-explanatory or clear from the context. Interested readers can refer to [9] for formal definitions of the same terminology.

The *topology* of a railway system is defined by a collection of named stations and identified lines. We differentiate between the lines within a station and those connecting two stations by referring the former as *lines* and the latter as *tracks*. A train *journey* is a sequence of visits to connected stations. Each *visit* to a station is represented by the arrival time and the departure time of the train. A *timetable* is an association from trains to journeys to be made. Scheduling rules is a set of temporal constraints to restrict the arrival time and departure time of each visit in order to prevent such undesirable events as train crash. A timetable is *valid* (or *feasible*) if no scheduling rules are violated under the associations. Otherwise, the timetable is *invalid* (or *infeasible)*.

Given a feasible timetable with n station visits, which is represented by a set of assignments (or equality constraints) of the form $AT_i = t_i^a$ and $DT_i = t_i^d$, where AT_i and DT_i denote the arrival and departure time respectively, for $1 \leq i \leq n$, the *modifications* that we can make to the timetable are to replace some assignments $AT_j = t_j^a$ (or $DT_j = t_j^d$) by $AT_j = t_j'^a$ (or $DT_j = t_j'^d$) where $t_j^a \leq t_j'^a$ (or $t_j^d \leq t_j'^d$) for $j \in \{1 \ldots n\}$. In other words, we can only delay the arrival time or departure time of visits. Given an infeasible timetable, *rescheduling* is the process of modifying the timetable so as to make the timetable feasible.

2.1 Problem Statement

There are six types of scheduling rules [9] in our railway system: the speed rule, the station occupancy rule, the station entry rule, the station exit rule, the line time rule, and the stopover rule. Let there be two trains 1 and 2 and two adjacent stations A and B. The variables AT_{XY} and DT_{XY} denote train Y's arrival and departure time at/from station X respectively. The above scheduling rules can be formulated as the following types of *scheduling constraints*.

The Speed Constraint

$$(lng/(AT_{A1} - DT_{B1})) \leq sp$$

The constant lng denotes the distance between station A and station B. This constraint enforces that the average train speed traveling between the two stations cannot exceed sp.

The Station Occupancy Constraint

$$(DT_{A2} + ctr \leq AT_{A1}) \vee (DT_{A1} + ctr \leq AT_{A2})$$

This constraint enforces that there is at least ctr time units between two trains occupying a track.

The Station Entry Constraint

$$(AT_{A1} - AT_{A2} \geq cen) \vee (AT_{A2} - AT_{A1} \geq cen)$$

This constraint enforces that there is at least cen time units between two trains entering a station via a line.

The Station Exit Constraint

$$(DT_{A1} - DT_{A2} \geq cex) \vee (DT_{A2} - DT_{A1} \geq cex)$$

This constraint enforces that there is at least cex time units between two trains departing from a station via a line.

The Line Time Constraints

$$((DT_{B1} < DT_{B2}) \wedge (AT_{A1} < AT_{A2})) \vee ((DT_{B1} > DT_{B2}) \wedge (AT_{A1} > AT_{A2}))$$
$$AT_{B1} < (DT_{B2} - cenx) \vee AT_{A2} < (DT_{A1} - cenx)$$

The line time rule is split into two constraints. The first constraint enforces that no train overtakes another train if they are traveling in the same direction on a line. The second constraint enforces that if there are two journeys on a line in opposing directions, the line must be unoccupied for at least $cenx$ time units.

The Stopover Constraint

$$DT_{A1} - AT_{A1} \geq cst$$

This constraint enforces that a train will stay in a station for at least cst time units.

Given the topology of a railway system with a valid train timetable, due to unexpected events, the users of the running map tool may want to modify the timetable. Our work is to first check the feasibility of the modified timetable. If it is feasible, the previous timetable is replaced by the modified one. Otherwise, we reschedule the infeasible timetable to generate a new feasible timetable. Note that efficiency should be a critical concern in designing the verification and rescheduling algorithms since in real-life situations, rescheduling must be performed in a timely manner. The notion of "efficiency" may vary according to situations. Ten minutes, however, should be a tolerable bound in general [2].

Optimal solutions are not required usually. In most cases, it is impractical to generate optimal solutions within a given (usually small) time bound. Criteria for optimality, however, should be defined. Such definitions can serve as guidelines for designing various variable-ordering and value-ordering heuristics to generate "good" answers. A precise notion of optimality also enables us to measure the "quality" of the rescheduled timetable. In the following, we present two optimality criteria.

A rescheduled timetable is *minimum-changes optimal* with respect to the original timetable if the least number of station visits are modified. This criterion can be satisfied easily in general since in most cases, we can simply delay the trains in question to the latest possible time. The resulting timetable, however, may introduce unreasonable long delay to some train visits. Thus this criterion should usually be applied with other criteria limiting the maximum delay.

A rescheduled timetable is *minimum-delay optimal* with respect to the original timetable if the longest delay among all train visits is minimum. Let the tuples $\langle AT_1, DT_1, \ldots, AT_n, DT_n \rangle$ and $\langle AT'_1, DT'_1, \ldots, AT'_n, DT'_n \rangle$ denote the infeasible and the rescheduled timetable respectively. The goal of this criterion is to minimize the following expression:

$$\max(AT'_1 - AT_1, DT'_1 - DT_1, \ldots, AT'_n - AT_n, DT'_n - DT_n).$$

The aims of the two criteria could contradict one another and represent the extremes of a spectrum of other possible definitions of optimality.

2.2 Related Work

Rescheduling is different from traditional scheduling in the sense that the possible solutions of rescheduling are restricted by the original schedule. Zweben *et al* [11] tackle this problem using constraint-based iterative repair with heuristics. The resultant GERRY scheduling and rescheduling system is applied to coordinate Space Shuttle Ground Processing. Our work is based on a propagation-based constraint solver.

Somewhat related to our work is train scheduling. Komaya and Fukuda [5] propose a problem solving architecture for knowledge-based integration of simulation and scheduling. Two train scheduling systems are designed in this architecture. Fukumori *et al* [3] use the tree search and constraint propagation technique with the concepts of time belt in their scheduling system. This approach is claimed to be suitable for double-track line and continuous time unit. Recently, Chiang and Hau [1] attempt to combine repair heuristic with several search methods to tackle scheduling problems for general railway systems.

There are two on-going projects that aim at automating train scheduling for real-life railway ministries. Our work is a direct outgrowth of the **PRaCoSy** project [10] at UNU/IIST. The latest **PRaCoSy** running map tool prototype uses constraints only passively to test for constraint violation in their verification engine. The Train Scheduling System (TSS) designed for Taiwan Railway Bureau (TRB) [6] is a knowledge-based interactive train scheduling system incorporating both an automatic and a manual schedulers. Users and the computer system are thus able to bring complementary skills to the scheduling tasks.

3 Timetable Verification

In the following, we describe a timetable verification algorithm, which examines if a given timetable is valid with respect to a set of scheduling constraints. Violated scheduling rules (or constraints) in an invalid timetable will be located and displayed to the user. The algorithm, shown in figure 1, assumes the existence of a propagation-based constraint solver [8] `propagate()`.

The timetable T can be viewed as a set of constraints for all variables in either the form $AT_i = t_i^a$ or $DT_i = t_i^d$, where AT_i is an arrival time and DT_i is a departure time. After all variables become ground (line `5`), we post the scheduling constraints one by one (lines `7-13`). Having all variables ground, the `propagate()` engine performs essentially constraint checking. It is easy to check

```
procedure verify(in C, T, out I)
/* C: scheduling constraints, T: timetable, I: violated constraints */
        /* Initialization */
        I ← {}
        S ← propagate(T)            /* Constraint store S is initialized to T */
        /* Constraint Verification */
        for each c ∈ C
                S ← propagate(S ∪ c)
                if inconsistency found
                        I ← I ∪ {c}
                        S ← S\{c}    /* Retract c from the constraint network */
                endif
        endfor
end
```

Fig. 1. The Timetable Verification Algorithm

that a constraint is violated under the given timetable if inconsistency is found after the constraint is told to the store. Violated constraints are retracted so that the algorithm can proceed to check for other possible constraint violations. Again, the groundness of all variables allows us to retract a constraint by simply removing the undesirable constraint from the constraint store.

4 Rescheduling as Constraint Satisfaction

Scheduling is an instance of constraint satisfaction problem. In the following, we show the same for rescheduling. Given a timetable T. Users modify T by adjusting its arrival and departure times, obtaining T', which can be valid or invalid. If T' is invalid, the *rescheduling* process should attempts to repair T' to make it feasible. By repairing, we mean adjusting the value of the non-modified variables so that (1) the timetable becomes valid again and (2) the new timetable should be reasonably "close" to the original timetable T. By being close to T, we mean that the new timetable should create the least service disruptions. Example optimality criteria are given in section 2. Note that user modified variables must be kept fixed during the rescheduling process since the modifications represent dispatcher requirements.

In order to formulate rescheduling as constraint satisfaction, we have to determine the variables of the problem, the domains associated with the variables, and the constraints of the problem. In rescheduling, the variables are the arrival and departure time of the timetable. Every variable share the integer domain $\{0, \ldots, 1439\}$[2]. There are three types of constraints in the rescheduling problem.

1. Scheduling constraints: The scheduling constraints set forth in section 2.

[2] There are 1440 minutes in 24 hours.

2. Modification constraints: For each arrival or departure time X which is modified by the user to new value t, we have the equality constraint $X = t$. This constraint enforces the user modifications to stay fixed during rescheduling.
3. Forward-labeling constraints: For each non-modified variable X with value t in the original timetable T, we have the constraint $X \geq t$. This constraint is necessary to ensure that we can only *delay* arrival or departure time.

Rescheduling now becomes finding a solution to the above constraint satisfaction problem. A solution is *optimal* if the solution is "closest" to the original timetable.

We are now ready to present the rescheduling algorithm, shown in figures 2 and 3. The algorithm can be divided into three phases. In phase one (line 4), we

```
1   procedure reschedule(in C, T)
2   /* C: scheduling constraints, T: feasible timetable */
3       /* Initialization */
4       S0 ← propagate(C)      /* Save a copy of the constraint store after
5                                  propagating constraints in C */
6       /* Rescheduling */
7       while true
8           Read in user modifications U    /* U is in the same form as T */
9           modify(T, S0, U, R)
10          if R = fail       /* No feasible timetable */
11              Prompt error messages
12          else
13              T ← R    /* Display rescheduled timetable if necessary */
14          endif
15      endwhile
```

Fig. 2. The Train Rescheduling Algorithm

post and propagate all scheduling constraints to prune infeasible values in the variables. The pruned constraint network is then saved in S_0. Since the scheduling constraints are the same for any timetable T and user modifications U, the same store S_0 will be reused in every rescheduling step. In real-life situation, rescheduling has to be performed repeatedly for different timetables and user modifications, this saving operation helps to avoid unnecessary invocations of constraint relaxation.

Actual rescheduling takes place in the procedure `modify()` (lines `16-46`). In the second phase (lines `19-35`) of rescheduling, information is extracted from user modifications and the original timetable to post and propagate the modification constraints (lines `23-27`) and the forward-labeling constraints (lines `28-35`). If inconsistency is found, rescheduling is halted and failure is reported. In the third phase (lines `36-46`), variables that are not modified by the users (extracted by the `vars()` function) are enumerated or labeled using some form of variable-

```
procedure modify(in T, S_0, U, out R)
/* T: previous feasible timetable, S_0: saved constraint network state,
   U: user modifications, R: rescheduled timetable */
      /* Initialization */
      O ← {X = t | (X = t) ∈ T, X ∈ vars(U)}
         /* Assignment constraints associating with user-modified variables */
      /* Post all user modifications */
      S ← propagate(S_0 ∪ U)       /* First and second type of constraints */
      if inconsistency found          /* User modifications are inconsistent */
            R ← fail
            return
      endif
      /* Prune infeasible values: can only delay arrival & departure time */
      for constraint (X = t) ∈ T\O
            S ← propagate(S ∪ (X ≥ t))      /* Third type of constraints */
            if inconsistency found
                  R ← fail
                  return
            endif
      endfor
      /* Rescheduling */
      E ← vars(T\O)                       /* Set of variables for rescheduling */
      A = labeling(E)
            /* Appropriate variable- and value-ordering should be used.
               The function labeling() returns either {} or a set of equality
               constraints (or bindings) for each variable X ∈ E */
      if A = {}
            R ← fail
      else
            R ← U ∪ A
      endif
```

Fig. 3. The Train Rescheduling Algorithm *(cont.)*

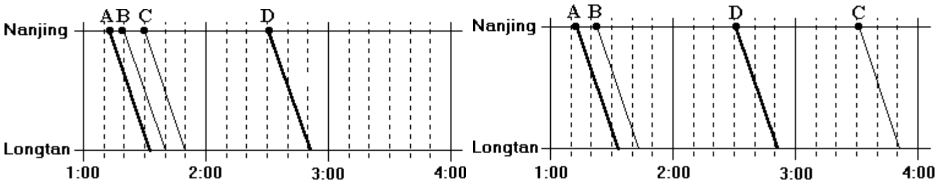
and value-ordering heuristics, which are embedded in the `labeling()` function, to speed up and direct the search towards a near-optimal solution.

There are two situations in which user modifications lead to a timetable T' that is non-repairable. First, the user modifications are self conflicting. Since user modified variables must be kept fixed during rescheduling, it is impossible to repair other variables to make the timetable valid. Second, user modifications are not self conflicting but there is no room for other variables to adjust to make the timetable valid. Constraint propagation algorithms are well-known to be incomplete [8]. Thus, phase two of the rescheduling algorithm can detect some, but not all, of this kind of conflicts. Theoretically speaking, the enumerating procedure in phase three can guarantee to detect inconsistency but it would usually take impractically long to do so. In cases when the rescheduling algorithm fails to return an answer within a few minutes, users are advised to abort the current computation, re-adjust the modifications, and restart the rescheduling process.

In the rest of this section, we present two variable labeling heuristics, which are designed to yield rescheduled timetables in the minimum-delay and the minimum-change optimal sense respectively.

Smallest-First Principle Variables are ordered in the ascending order of the lower bound of their domains. Values in the variable domains are also enumerated in ascending order. This principle is founded on the assumption that a short delay on a train visit will cause short delay on the subsequent one, which means that delays propagate in a monotonic fashion. Experimental results confirm that, using actual timetables from **PRaCoSy**, this heuristic usually helps to generate solutions that are minimum-delay optimal efficiently. We construct below an unrealistic artificial example that defeats the heuristic.

Figure 4 (a) shows a small segment of four journeys on a railway running



(a) Before Rescheduling (b) After Rescheduling

Fig. 4. A Non-Optimal Solution by Smallest-First Heuristic

map. The three journeys A, B, and C share the same track in the Nanjing station, while the two journeys C and D take the same line in traveling from Nanjing to Longtan. The journeys A and D are fixed (indicated by thick lines) by the users. Suppose the station occupancy and the station exit rules enforces that at least ten minutes among each of the three points A,B, and C, and sixty minutes between the points C and D respectively. The modified timetable is thus infeasible due to the insufficient long distance between the points A and B.

Figure 4 (b) shows the rescheduled timetable obtained using the smallest-first principle. The rescheduling starts from moving point B ahead in time to achieve the ten-minute requirement between points A and B. The movement in turn causes another conflict between points B and C. Point C is thus forced to move. However, there is no feasible location for point C to move between points B and D since point D is fixed by user. Therefore, we have to move point C one-hour ahead of point D. The maximum delay in this case is two hours. This solution is not optimal since a better solution can be obtained by simply moving point B twenty minutes ahead in time, as shown in figure 5.

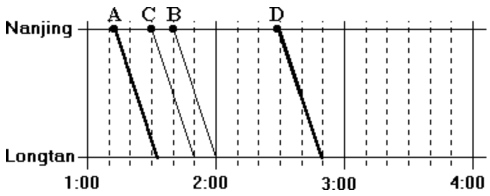


Fig. 5. A Smallest-Changes-Optimal Solution

Consistent-Assignment-First Principle This heuristic suggests to instantiate first those variables that can be instantiated with its time value in the original timetable. We call these variables *non-conflicting*. The other variables are *conflicting*. Therefore, the labeling of the non-conflicting variables will be backtracked into last. Again values in the variable domains are enumerated in ascending order. The idea is to maintain as many variables with its original value as possible. This heuristic direct searching towards a minimum-change optimal solution. Note that a non-conflicting variable may be instantiated with a value other than its original value eventually if the conflicting variables have tried all possible combination of time values and no solution is found.

For efficiency reason, we further classify the non-conflicting variables into two group. The first group contains variables that share journeys with one of conflicting variables. The second group contain the rest of the non-conflicting variables. Our heuristic suggest to label first the second group of non-conflicting variables. This ordering is essential since the labeling of variables sharing journeys with conflicting variables have a higher chance of being backtracked into.

We apply this heuristic to reschedule the infeasible timetable in figure 4 (a). Recall that the points A and D are fixed by users. We classify the variables associated with point B and point C as conflicting and non-conflicting respectively. Thus point C is labeled first to retain its original position in the map and point B is forced to move until it reaches the location ten-minute ahead point C. In this specific case, the minimum-change optimal solution coincides with the minimum-delay optimal solution. This example also shows that, in general, the two heuristics give different first solution.

5 Prototype Implementation

In order to demonstrate the feasibility of our algorithms, we have re-constructed and enhanced the **PRaCoSy** running map tool prototype [7] with rescheduling capability. The prototype consists of a constraint-based scheduler and a user-interface. The former is implemented in C++ with ILOG Solver library 2.0 [4] while the latter is built using Microsoft Visual Basic 3.0.

In the following, an overview of the running map tool is presented. We then give a sample session of our tool using a segment (from Nanjingxi to Shanghai) of the China railway which amounts to 3005 constraints and 596 variables. The rescheduled timetables generated by the two heuristics are explained. We conclude this section by showing two examples to which our tool fails to respond in a timely manner.

The running map display (figure 6) consists of six columns (regions). Column one and column four show the abbreviated identifiers of stations. Column two shows the number of arrival and departure lines of stations. Column three is the main window for presenting the graphical representation of a timetable, with time and locations as the X and Y axes respectively. Column five shows the cumulative distance from the first station. Column six shows the distance between the current station and the previous station. In the cases where a timetable is too large to fit into the main window, two scrollbars will be enabled.

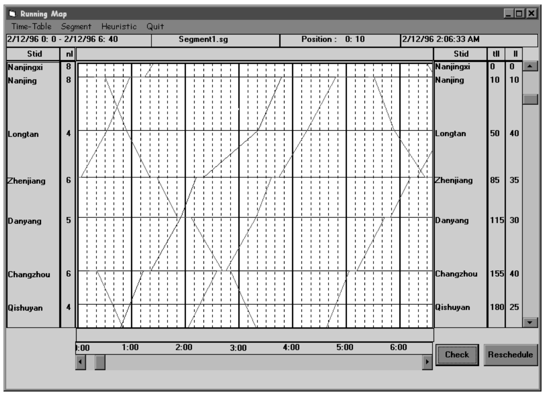


Fig. 6. The Running Map Tool

A user can click-and-drag any lines to modify the corresponding train visits on the map. The modified timetable will be validated using the verification algorithm when the "Check" button is pressed. If it is infeasible, a warning window, such as that shown in figure 7, will pop up to display all constraint violations. At this point, the user can either invoke the rescheduling algorithm by pressing the "Reschedule" button, correct the modifications manually, or restore the original feasible timetable.

Figure 8 (a) shows a segment of a China railway timetable. Due to an acci-

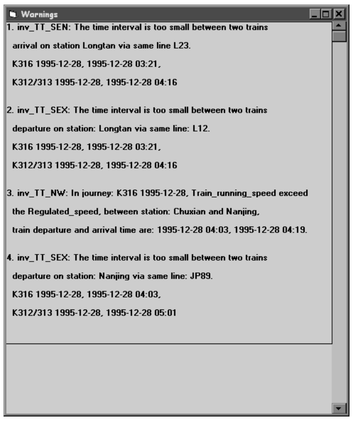
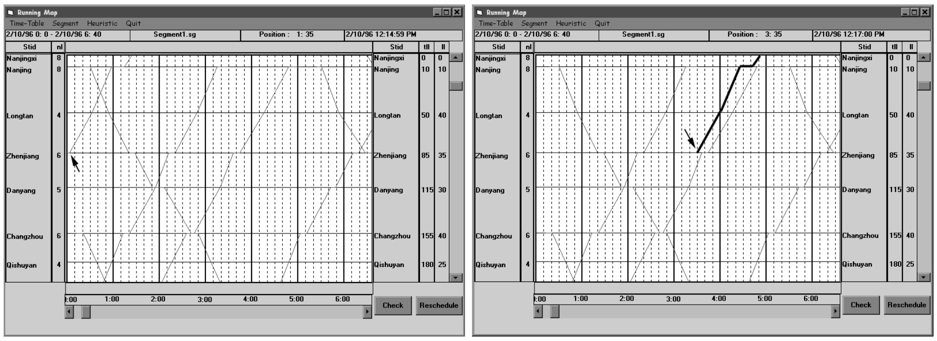


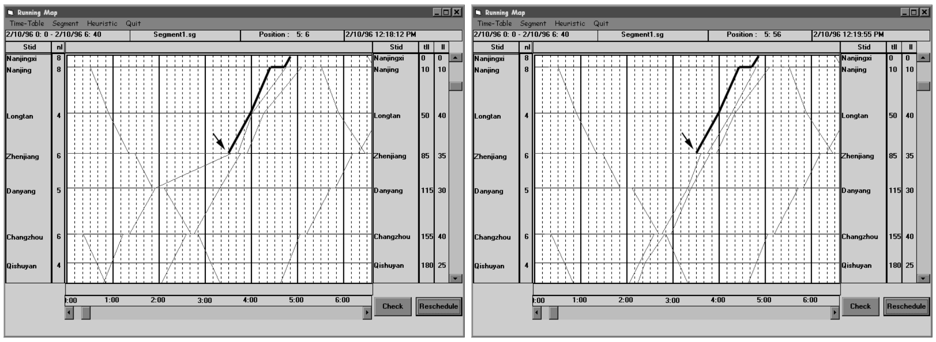
Fig. 7. A Warning Window



(a) Original Feasible Timetable (b) Modified Infeasible Timetable

Fig. 8. A Comparison of Two Heuristic

dent, we have to delay the train departuring from Zhenjiang at 1:05 to 3:30, yielding the map shown in figure 8 (b). The user-modified departure time (pointed by an arrow) and all the subsequent train visits on the journey (the highlighted segment) are then fixed immediately by the running map tool. This user movement incurs seven constraint violations between the modified journey and its left adjacent journey. We reschedule the infeasible timetable with our two heuristics. Both of them succeed in generating a feasible timetable within a few seconds. Figure 9 (a) and figure 9 (b) show the rescheduled timetable generated using the smallest-first principle and the consistent-assignment-first principle respectively.



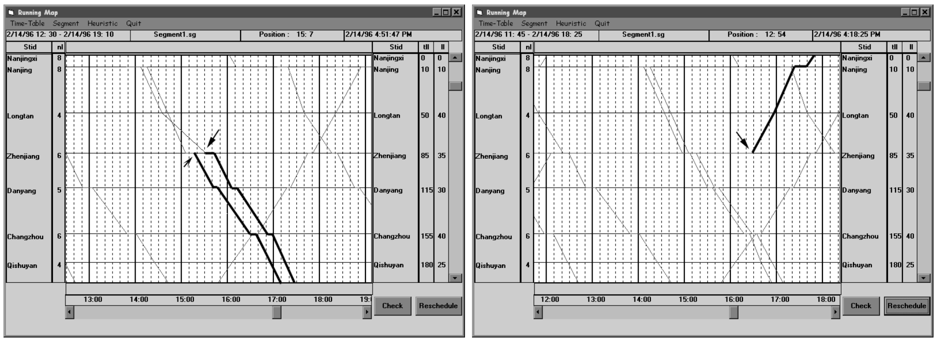
(a) Smallest-First Principle (b) Consistent-Assignment-First Principle

Fig. 9. A Comparison of Two Heuristic *(cont.)*

Applying the smallest-first principle, we process all visits on the map from the left (earliest) to the right (latest). For each visit, if its associated arrival (or departure) time does not violate any scheduling constraints, we preserve the current value. Otherwise, we move the time ahead as little as possible to eliminate the inconsistencies. Thus some visits on a journey may be modified while others remain unchanged. This explains why, visually, a journey is not only shifted right horizontally, but can also be "bent" by the rescheduling process. The movement propagates in the above fashion from left to right. Whenever no further movements are possible, backtracking takes place.

Instead of massaging several journeys to produce a feasible timetable, the consistent-assignment-first principle suggests to modify as few station visits as possible. This goal can be well approximated by first locating station visits that can remain unchanged with respect to the user modifications. These station visits will labeled first. The conflicting variables, having to change their original values, will be labeled last. Our experiments reveal that, in many cases, this heuristic produces timetable which is "almost identical" to the original timetable. As seen in figure 9 (b), most of the journeys retain their original locations. Even for the right-shifted journey, its shape is mostly preserved.

Experimental results confirm that rescheduling can usually be completed within seconds. This is not always the case. Figure 10 (a) and figure 10 (b) provide two such examples. The infeasible timetable in figure 10 (a) can be rescheduled using the consistent-assignment-first principle in a few seconds, but the smallest-first principle fails to return an answer within five minutes. Figure 10 (b) is simply a non-repairable timetable. Neither heuristics can return promptly to confirm the unsolvability of the problem.



(a) Infeasible Timetable 1 (b) Infeasible Timetable 2

Fig. 10. Poor Performance Example

6 Concluding Remarks

The contribution of this paper is three-fold. First, we define formally train rescheduling as a constraint satisfaction problem. Two algorithms for railway timetable verifications and rescheduling are then derived based on a propagation-based constraint solver. We define two optimality criteria, which are used to measure the "quality" of the rescheduled timetable. It is important to note that the optimality criteria are defined with respect to the original timetable. Second, based on the domain knowledge learned from domain analysis, we propose two heuristics to speed up and direct the search towards minimum-delay optimal and minimum-change optimal solutions respectively. The feasibility of our proposed algorithms and heuristics are confirmed with experimentation using real-life data. Third, we have re-constructed and enhanced the **PRaCoSy** running map tool prototype.

It would be an interesting future work to study different stochastic methods for train rescheduling. Work is also in progress to experiment our rescheduling method on larger-scale real-life railway timetables.

Acknowledgement

We acknowledge, with pleasure, the interaction we have had with Fellows and Staff of UNU/IIST, the United Nations University, International Institute for Software Technology, Macau. In particular, we are indebted to Prof. Dines Bjørner for inviting our participation in the **PRaCoSy** project. We also had numerous fruitful discussion and working sessions with Søren Prehn, Chris George, Yulin Dong, Liansuo Liu, and Dong Yang. Last but not least, we thank the anonymous referees for their constructive comments, which help to improve the final version of the paper.

References

1. T.W. Chiang and H.Y. Hau. Railway scheduling system using repair-based approach. In *Proceedings: Seventh International Conference on Tools with Artificial Intelligence*, pages 71–78, 1995.
2. Y. Dong. The Zhengzhou $\leftrightarrow$ Wuhan train dispatch system. UNU/IIST PRaCoSy Document DYL/1/3, International Institute for Software Technology, United Nations University, July 1994.
3. K. Fukumori, H. Sano, T. Hasegawa, and T. Sakai. Fundamental algorithm for train scheduling based on artificial intelligence. *Systems and Computers in Japan*, 18(3):52–63, 1987.
4. ILOG. *ILOG: Solver Reference Manual Version 2.0*, 1994.
5. K. Komaya and T. Fukuda. A knowledge-based approach for railway scheduling. In *Proceedings: Seventh IEEE Conference on Artificial Intelligence Applications*, pages 405–411, 1991.
6. H.C. Lin and C.C. Hsu. An interactive train scheduling workbench based on artificial intelligence. In *Proceedings: Sixth International Conference on Tools with Artificial Intelligence*, pages 42–48, 1994.
7. X. Liu. A simple running map display tool. UNU/IIST PRaCoSy Document lx/tool/02, International Institute for Software Technology, United Nations University, August 1994.
8. A.K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.
9. S. Prehn. A railway running map design. UNU/IIST PRaCoSy Document SP/12/3, International Institute for Software Technology, United Nations University, July 1994.
10. S. Prehn and D. Bjørner. PRaCoSy: An executive overview. UNU/IIST PRaCoSy Document par/02/09, International Institute for Software Technology, United Nations University, June 1994.
11. M. Zweben, E. Davis, B. Daun, and M.J. Deale. Scheduling and rescheduling with iterative repair. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6):1588–1596, November/December 1993.

This article was processed using the LaTeX macro package with LLNCS style