Constraints, 7, 167–198, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

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, China

Abstract. In this paper, we report the design and implementation of a constraint-based interactive train rescheduling tool, a project in collaboration with the International Institute for Software Technology, United Nations University (UNU/IIST), Macau. We formulate train rescheduling as constraint satisfaction and describe a constraint propagation approach for tackling the problem. Algorithms for timetable verification and train rescheduling are designed under a coherent framework. Formal correctness properties of the rescheduling algorithm are established. We define two optimality criteria for rescheduling that correspond to minimizing the number of station visits affected and passenger delay respectively. Two heuristics are then proposed to speed up and direct the search towards optimal solutions. The feasibility of our proposed algorithms and heuristics are confirmed with experimentation using real-life data.

Keywords: train rescheduling, constraint propagation, search heuristics

1. Introduction

The **PRaCoSy** (**P**eople's Republic of China **Ra**ilway **Co**mputing **Sy**stem) project [49] is undertaken by the International Institute for Software Technology, United Nations University (UNU/IIST), Macau. 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 to computerize the preparation and updating of the *running map*[1] for dispatching trains along the 600 kilometer railway section between Zhengzhou north and Wuhan south. This section 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 [48] contains information regarding the topology of the railway, train number and classification, arrival and departure times 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 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 1. 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

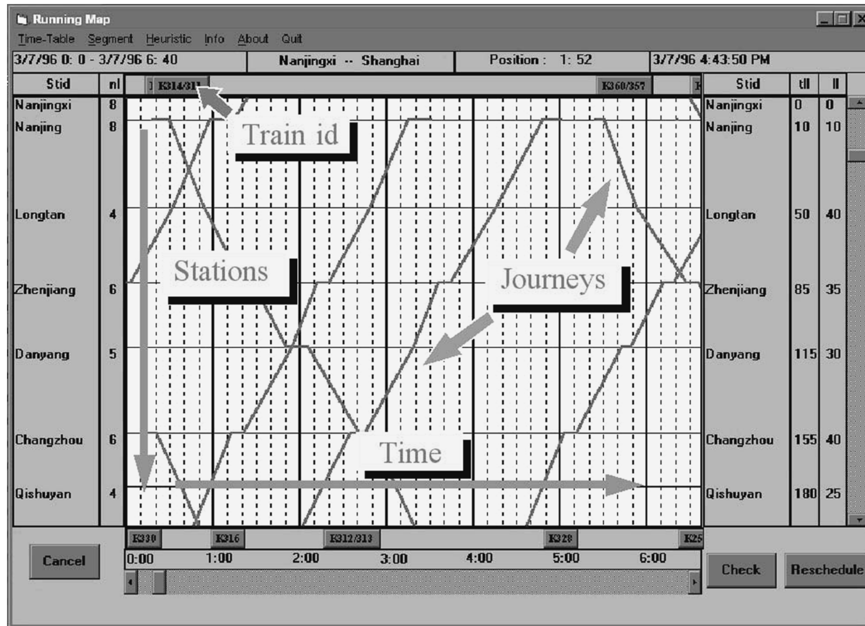

Figure 1. A running map tool.

against possible violation of these 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 [38]. Our task at hand is to enhance the **PRaCoSy** tool to perform automatic rescheduling, which can be considered as constraint re-satisfaction. 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. Since the running map tool was intended for actual deployment in the field and the duration of the project was only one year, the chosen technology must be state of the art and, at the same time, established and stable with good support. We have ruled out algorithms still in its experimental stage. Around 1995, most commercial constraint programming systems available in the market were based on a combination of tree search and constraint

propagation [31, 45]; hence our choice of technology. Our results do confirm the feasibility of tree search plus constraint propagation in this particular application.

In this paper, we show how timetable verification and train rescheduling can both be formulated as constraint satisfaction problems and give associated algorithms used in our tool. We demonstrate 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 this paper, an extended and revised version of [11], is organized as follows. Section 2 defines basic terminology, gives a tutorial on constraint satisfaction techniques, 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 establish the correctness of the algorithm formally. In Section 5, we discuss two heuristics that help to direct and speed up the search towards optimal solutions. In Section 6, we describe our prototype implementation and sample runs of the tool. We summarize our contribution and shed light on future work in Section 7.

2. Preliminaries

In the following, we provide definitions of necessary terminology according to Prehn [48]. The problem statement is also given formally to facilitate subsequent discussions. Before we review related work, we give a brief tutorial on constraint satisfaction techniques and tools.

2.1. Terminology

2.1.1 A Model of Railway Topology

A running map contains information regarding the railway topology. The model of a railway topology includes *station*, *track*, *line*, *network* and *segment*.

Station. A station is a place for trains to stay or pass through. Each station is associated with a unique station identifier (*stid*). There are one or more *tracks* in a station.

Track. A track is a place for a train to stay or run within a station. Each track has a unique track identifier (*trid*). Tracks can have different lengths and types. Track types are defined according to their appropriate use: LINE, SIDING, PLATFORM or FREIGHT. LINE tracks are used by trains passing through a station. PLATFORM tracks are places for passenger trains to reside on. Freight trains couple or decouple goods on FREIGHT tracks. Trains which need to wait can stay on SIDING tracks. Each track has a set (*flns*) of *lines* from which it can be reached, and another set (*tlns*) of lines to which a train can move from the track.

Line. Lines are railways which form the connection between two stations. Each line has a unique line identifier (*lnid*), its length and maximum speed. No trains should exceed a line's maximum speed when running on it. There are 3 types of lines: UP, DOWN or BOTH. Trains running on an UP line or DOWN line can only move along a certain direction, while trains on a BOTH line can move in either direction. We assume that two connected stations have only one line connecting them. In other words, there are no parallel lines connecting two stations.

Network. A network is defined as a "consistent" pair of stations and line descriptions. In other words, lines must connect known stations, in a manner which is consistent with the tracks of all stations described. For example, if a line *lnid* connects two stations S_1 and S_2, *lnid* must be contained in the *tlns* of S_1 and the *flns* of S_2.

Segment. A segment is a sequence of two or more stations, in which adjacent stations are connected in the network.

2.1.2 A Model of Traffic

The traffic condition of a train are described by *journeys*. A journey is the sequence of *station visits* by a train. A station visit consists of a station identifier, arrival and departure times, the track used by the train within the station, (optional) departure line, and arrival and departure train length.

2.1.3 A Model of Timetable

A *timetable* is an association from train identifiers to journeys to be made. Each train identifier contains a train number and a date which indicates the starting date of that train.

Scheduling rules are a set of temporal constraints to restrict the arrival time and departure time of each visit in order to prevent undesirable events such as a train crash.

A timetable is *valid* (or *feasible*) in a network if it conforms with the network topology and 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 the ith station visit ($1 \leq i \leq n$). *User modifications* are replacement of some assignments $AT_j = t_j^a$ (or $DT_j = t_j^d$) by $AT_j = t_j'^a$ (or $DT_j = t_j'^d$), for $j \in \{1 \ldots n\}$ and some $t_j'^a$ (or $t_j'^d$). Given an infeasible timetable, *rescheduling* is the process of modifying the timetable so as to make the timetable feasible. The rescheduling process, however, only attempts to adjust the values of the unmodified arrival and departure times. The rescheduling process replaces some assignments of the unmodified times $AT_j = t_j^a$ (or $DT_j = t_j^d$) by $AT_j = t_j'^a$ (or $DT_j = t_j'^d$) for some $t_j'^a \geq t_j^a$ (or $t_j'^d \geq t_j^d$) and $j \in \{1 \ldots n\}$. In other words, the process can only delay the arrival times or departure times of unmodified station visits.

Note that user modifications may violate the scheduling rules in two possible respects:

1. the modified arrival and departure times violate the scheduling rules directly; or
2. the modified arrival and departure times do not violate any scheduling rules, but they conflict with some unmodified arrival and departure times.

In the first case, the modifications have conflicts among themselves. Rescheduling can never succeed since, by definition, rescheduling should only attempt to adjust the unmodified arrival and departure times. User modifications must be respected since they are intended and produced by experienced train dispatchers. A rescheduling tool should just inform users of such conflicts. In the second case, there are no direct conflicts among the modifications. We can invoke rescheduling to attempt repairing the infeasible timetable.

2.2. Problem Statement

There are six types of scheduling rules [48] 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 modeled using the following *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 when 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 are at least ctr time units between two trains' occupancy of a track.

The Station Entry Constraint

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

This constraint enforces that there are at least cen time units between two trains' entrance to 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 are at least cex time units between two trains' departure 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 opposite 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 stays in a station for at least *cst* time units.

Rescheduling. Due to unanticipated events, users of the running map tool might want to modify the original valid 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 modified timetable to generate a feasible new 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 [20].

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-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 timetables respectively. The goal of this criterion is to minimize the following expression:

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

Figure 2 shows three trains $T1$, $T2$, and $T3$ staying in the same station on three different tracks. Suppose $T1$, $T2$, and $T3$ leave the station at 12:05, 12:10, and 12:15, respectively. If the constant *cex* of the station exit rule is 5 minutes, the timetable does not violate the station exit rule. However, if a user modifies the departure time of $T1$ to 12:10, then the station exit rule is violated and rescheduling is needed (Figure 3). Figure 4 shows a minimum-delay optimal solution. The timetable does not violate the station exit rule because no two trains would leave the station within any 5 minute interval. Both $T2$ and $T3$ have been delayed 5 minutes. Therefore, the maximum delay is 5 minutes in this case. To keep the station exit rule valid, it is obvious that we can

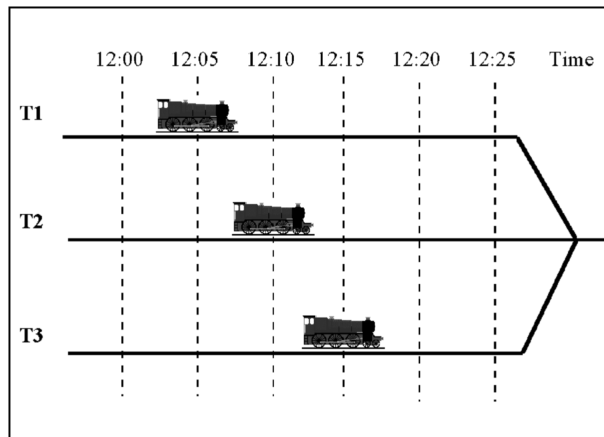

Figure 2. Comparison of two optimal criteria: Original timetable.

find no other solutions in which all trains are delayed for less than 5 minutes. Hence, Figure 4 has shown the minimum-delay optimal solution for this case.

With the minimum-delay optimal criterion, the maximum delay time is minimized but more journeys will be affected. Users would prefer this criterion if they are concerned more on minimizing the maximum delay time. Passengers may not mind the small delay although more passengers are affected.

A rescheduled timetable is *minimum-change optimal* with respect to the original timetable if the least number of station visits are modified. Figure 5 shows the minimum-change optimal solution for the modification in Figure 3. The new timetable does not

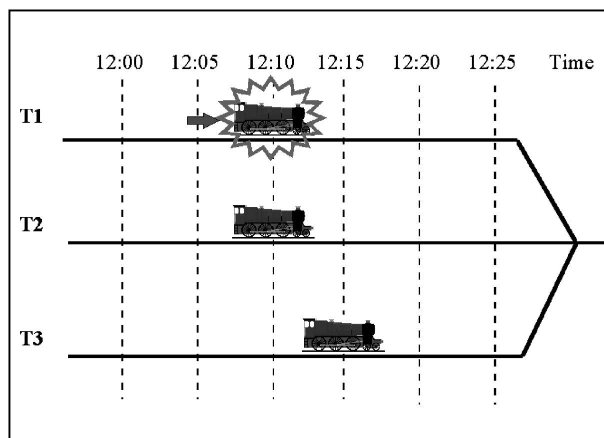

Figure 3. Comparison of two optimal criteria: After modification.

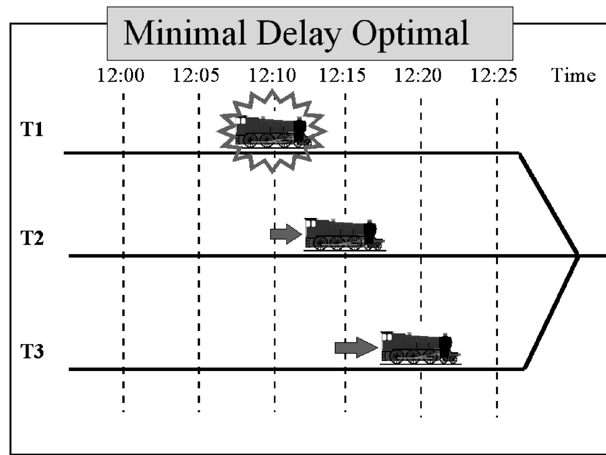

Figure 4. Comparison of two optimal criteria: Minimum-delay optimal.

violate the station exit rule because no two trains would leave the station within 5 minutes. Since only one train $T2$ is delayed, the number of journeys changed is minimized.

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 unreasonably long delays to some train visits. Thus this criterion should usually be applied with other criteria limiting the maximum delay.

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

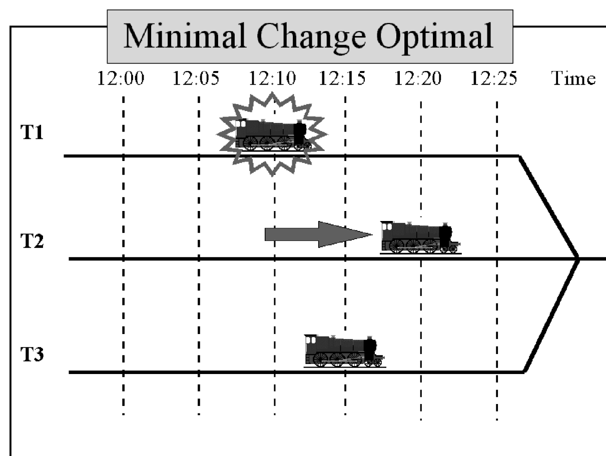

Figure 5. Comparison of two optimal criteria: Minimum-change optimal.

2.3. Constraint Programming

Real-world scheduling problems are combinatorial in nature. In its simplest form, scheduling problems are no different from classical combinatorial problems such as graph colorings, N-queen puzzles and crypt-arithmetic problem. All of the problems involve solution searching from a large search space under various kind of constraints [58]. These class of problems are collectively called constraint satisfaction problems (CSPs). In this section, we give a brief introduction to CSP, an overview of the constraint programming approach to solving CSPs, and the available tools.

2.3.1 Features of Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is defined as a 3-tuple (Z, D, C), where

- Z is a finite set of variables $x_1, x_2, \ldots, x_n$,
- D is a function mapping each variable $x \in Z$ to a finite set of possible values called the *domain* of x, and
- C is a finite set of constraints, each of which acts on a subset of variables in Z restricting the possible combination of values that these variables can take.

A solution to a CSP is a *consistent* variable assignment so as to make all constraints satisfied simultaneously. A naive solution for CSP is generate-and-test but it is grossly inefficient. A more efficient approach is backtracking tree search, which essentially performs a depth-first search [30] of the space of the potential CSP solutions. The performance of backtracking, although better than generate-and-test, is still poor due to its thrashing [24] behavior.

Constraint propagation, based on node/arc/path consistency techniques [2, 16, 39, 40, 42, 59], is a more sophisticated approach to tackle CSPs. In this approach, variables are represented by nodes in a graph and binary constraints are represented by edges. During constraint propagation along the edges, values are removed from the domains of the nodes until each arc (constraint) is individually satisfiable. However, global satisfiability is not guaranteed and searching is needed. Different constraint satisfaction algorithms are obtained by combining tree searching and different degrees of constraint propagation [15, 23, 24, 26, 41, 44, 50].

Given a CSP, we first use a constraint propagation algorithm, such as the well-known arc consistency algorithm AC-3 [39] as explained above, to attain a desired level of consistency. If no solution or inconsistency is found, then one of the variables with domain size larger than 1 is selected and a new CSP is created for each possible assignment of this variable. These new CSPs become the successor of the current CSP. Repeating this constraint propagation and domain enumeration process for each new CSP results in possibly more new CSPs. All newly generated CSPs can be organized conveniently in the form of a search tree with the original CSP as the root node. A standard backtracking algorithm can be used to visit these nodes to search for solutions.

We have two choices in the above tree search process: *variable-ordering* and *value-ordering*. These orderings can affect the efficiency of the search strategies significantly. Variable-ordering concerns which variable to instantiate next; whereas value-ordering sets the order in which values in the domain of the chosen variable are tried. Variable-ordering can greatly affect the branching factor and hence also the size of the resulting search space. Value-ordering affects the ordering of branches in the search tree. This is important when users are interested in obtaining the first solution of a CSP fast using some form of depth-first search strategy. Therefore, various variable- and value-ordering heuristics [4, 26], such as the first-failed principle, have been devised to speed up solving of particular CSP instances. These heuristics are, however, usually problem and domain specific.

2.3.2 Constraint Programming Tools

There are a wide variety of constraint programming languages and tools, such as Prolog III [13], CLP($\mathscr{R}$) [28], and CHIP [19, 58]. These languages and tools, each differing in its constraint domain and serving in different areas of applications, offer an expressive and flexible language for problem specification and heuristics programming, allowing rapid program development for complex problems and enabling programs to be easily modified and extended. CSP-solving and scheduling are most related to CHIP and ILOG SOLVER [52], the uniqueness of which come from their finite domain constraint-solving capabilities. The idea underlying CHIP is to enhance Prolog with constraint propagation techniques [39], which prunes the search space before performing tree search. ILOG SOLVER [52] is a constraint programming language that merges constraint propagation techniques [39] and object-oriented programming. ILOG SOLVER is realized as a C++ library that embodies such constraint logic programming concepts as logical variables, incremental constraint satisfaction, and backtracking. ILOG SCHEDULE [32], built on top of ILOG SOLVER, is a library of specialized constraints for constraint-based scheduling. CHIP and ILOG SOLVER have been applied successfully to solving such industrial applications as car sequencing, disjunctive scheduling, graph coloring, and firmware design. A fuller account of their applications can be found in [1, 17, 18, 33, 46, 47, 51, 58].

2.4. 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. [61] 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. Carey [6, 7] and Carey and Lockwood [8] tackle the problem using mathematical programming techniques. Tsuruta and Matsumoto [57] and Schaefer [54] use a knowledge-based approach and apply expert system technologies. Cheng [9] proposes a

hybrid method of the network-based simulation and the event-driven simulation for efficiently resolving resource conflicts in train traffic rescheduling in major automatic-block districts of double-track railway lines.

Our work is also related to repair and rescheduling in other domains. Tsukada and Shin [56] study the use of agent negotiation to recover from a disruption in a distributed plan. The idea is to resolve, as much as possible, locally the problem of finding a response to a disruption, in such a way that it will be least disruptives to other agents. Wu et al. [60] formulate one-machine rescheduling as a multiple criteria optimization problem. Sauer and Bruns [53] develop a generic framework using a knowledge-based approach for building practical scheduling systems. The framework has been applied to serious real-life problems in the manufacturing domain and an application in the medical domain. Goldman and Boddy [25] introduce the constraint envelope scheduling technique, a least-commitment and approximate reasoning approach to constraint-based rescheduling. The least-commitment nature enables schedules to be updated more easily. However, the scheduler can no longer construct a single timeline representing, for example, the change in resource availability over time. Instead, only the bounds on the system's behavior are computed. CREWS_NS [43] is a system that addresses the long-term scheduling of train crews at the Dutch Railways. The system adopts the white-box approach, in the sense that the planner can perceive what is going on, can interact with the system by proposing alternatives or querying decisions, and can adapt the behavior of the system to changing circumstances. CREWS_NS also uses extensive AI techniques, such as abstraction to reduce problem details, a modified version of beam search [3] with heuristics to explore the state space, and constraint satisfaction to reduce the state space and to guide search. Our work is unique in that we formulate rescheduling also as a constraint satisfaction problem and use simple variable-ordering heuristics to achieve optimality.

Somewhat related to our work is train scheduling. Komaya and Fukuda [29] 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. [22] use the tree search and constraint propagation technique with the concept 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 [10] have attempted to combine repair heuristics 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 [49] 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) [37] is a knowledge-based interactive train scheduling system incorporating both an automatic and a manual scheduler. Users and the computer system are thus able to bring complementary skills to the scheduling tasks.

```
1    procedure verify(in C, T, out I)
2    /* C: scheduling constraints, T: timetable, I: violated constraints */
3         /* Initialization */
4         I ← {}
5         /* Constraint Verification */
6         for each possibly violated constraint c ∈ C
7              if propagate(T ∪ {c}) = inconsistency then
8                   I ← I ∪ {c}
9              endif
10        endfor
11   end
```

Figure 6. The timetable verification algorithm.

3. Timetable Verification

Timetable verification is the process of examining whether a given modified timetable is valid with respect to the set of scheduling rules. Timetable verification is invoked in two occasions. First, when a (modified) timetable is read into the system, it is verified to see if any scheduling rules have been violated. Second, a user might invoke verification any time after he has modified a timetable.

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 6, assumes the existence of a sound propagation-based constraint solver [39] `propagate()`.

The input timetable T can be viewed as a set of constraints of the form $X = a$, where X is a variable, denoting either an arrival or a departure time of a station visit, and a is a constant. The verification algorithm considers each of the scheduling constraints in turn and identifies the violated constraints by making use of the `propagate()` routine. Since all variables in T are ground, the verification process propagate($T \cup \{c\}$) virtually becomes a simple substitute-and-test process. Note that we have the assumption that the timetable before user modification is feasible; hence, only the constraints which are related to the modified journeys can possibly be violated. These constraints usually form a very small subset of C. As an optimization, only these constraints will be checked in line 6.

4. Rescheduling as Constraint Satisfaction

Scheduling is a well-known instance of constraint satisfaction problems. In the following, we show that rescheduling can also be formulated as a constraint satisfaction problem. Given a timetable T, users modify T by adjusting its arrival and departure times, obtaining a new timetable T'. If T' is invalid, the *rescheduling* process attempts to repair T' to make it feasible. By repairing, we mean adjusting only the values of the unmodified variables so that (1) the timetable becomes valid again and (2) the new timetable T' is reasonably "close" to the original timetable T. By being close to T, we mean that

the new timetable should cause 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 the dispatcher's requirements.

Given the task of rescheduling, we can construct a constraint satisfaction problem as follows. The variables are the arrival and departure times of the new timetable T'. Each of these variables has the integer domain $\{0, \ldots, 1439\}$.[2] There are four types of constraints in the rescheduling problem, namely:

1. Scheduling constraints: the scheduling constraints set forth in Section 2.
2. Stopover-maintenance constraints: for the arrival time AT and departure time DT of each station visit, we have the constraint $DT - AT \geq OWT$, where OWT denotes the waiting time of the station visit in the old timetable before rescheduling. These constraints enforce that every train will stay in the station no less than its original waiting time.
3. Modification constraints: for each arrival or departure time X modified by user to new value t, we have the equality constraint $X = t$. This constraint enforces the user modifications to stay fixed during rescheduling.
4. Forward-labeling constraints: for each unmodified 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 is equivalent to finding a solution to the constraint satisfaction problem as specified. 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 7 and 8. The algorithm can be divided into three phases. In phase one (line 4), we post and propagate all scheduling constraints to prune infeasible values in the variables. The pruned constraint network is then saved in S_0. Actual rescheduling takes place in the procedure `modify()` (lines 17–47). In the second phase (lines 20–36) of rescheduling, information

```
01  procedure reschedule(in C, T1)
02  /* C: scheduling constraints, T1: feasible timetable */
03      /* Initialization */
04      S0 ← propagate(C)
05      /* Rescheduling, assuming no inconsistency detected in C */
06      i ← 0
07      while true
08          i ← i + 1
09          Read in user modifications Ui
10          modify(Ti, S0, Ui, Ri)
11          if R = fail then /* No feasible timetable */
12              Ti+1 ← Ti /* Prompt error messages */
13          else
14              Ti+1 ← Ri /* Display rescheduled timetable */
15          endif
16      endwhile
```

Figure 7. The train rescheduling algorithm.

```
17  procedure modify(in T, S0, U, out R)
18  /* T: previous feasible timetable, S0: saved constraint network state,
19      U: user modifications, R: rescheduled timetable */
20          /* Unmodified times */
21          O ← {X = t | (X = t) ∈ T ∧ X ∉ vars(U)}
22          /* Post constraints */
23          /* Post user modifications and scheduling constraints */
24          S ← propagate(S0 ∪ U)
25          if S = inconsistency then /* User modifications conflicting */
26                  R ← fail
27                  return
28          endif
29          /* Post forward-labeling constraints */
30          for each constraint (X = t) ∈ O
31                  S ← propagate(S ∪ {X ≥ t})
32                  if S = inconsistency then
33                          R ← fail
34                          return
35                  endif
36          endfor
37          /* Rescheduling */
38          E ← vars(T) \ vars(U)   /* Set of variables for rescheduling */
39          A = labeling(E)
40                  /* Appropriate variable- and value-ordering should be used.
41                      The function labeling() always returns either {} or a
42                      set of equality constraints for each variable X ∈ E */
43          if A = {} then
44                  R ← fail
45          else
46                  R ← U ∪ A
47          endif
```

Figure 8. The train rescheduling algorithm (continued).

is extracted from user modifications and the original timetable to post and propagate the modification constraints (lines 23–28) and the forward-labeling constraints (lines 29–36). If inconsistency is found, rescheduling is halted and failure is reported. In the third phase (lines 37–47), variables that are not modified by the users (extracted by the `vars()` function) are enumerated using some form of variable- and value-ordering heuristics (embedded in the `labeling()` function) to speed up and direct the search towards an optimal solution.

There are two situations in which user modifications can lead to a non-repairable timetable T'. First, since user-modified variables must be kept fixed during rescheduling, it is impossible to repair other variables to make the timetable valid if the user modifications are self-contradicting. Second, if user modifications are not self-conflicting, there might still be no room for other variables to adjust to make the timetable valid. Constraint propagation algorithms are well-known to be incomplete [39]. 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. The meaning of a few minutes usually depends on the patience of the particular train dispatchers and the urgency of the situations although, as stated before, a good upper bound for each run is ten minutes [20]. This iterative adjustment process appears to be a trial-and-error approach, but an experienced train dispatcher can usually finish the process within the allowed time limit.

The correctness of the rescheduling algorithm can be concluded by proving both the algorithms in Figures 7 and 8 correct. The formal proofs appear in [12]. We give the informal reasoning as follows. The goal of the algorithm in Figure 7 is to accept user modifications, which are in turn passed to the algorithm in Figure 8 for revising the timetable if possible. The algorithm in Figure 7 consists of a `propagate()` function and a non-terminating loop (lines 7–16). The `propagate()` function, which performs constraint propagation, is for improving efficiency and does not affect the correctness of the algorithm. The **while**-loop serves the purpose of accepting user modifications and passing them to the core procedure `modify()`.

The procedure `modify()` takes as input a feasible timetable T, a propagated constraint store S_0 of the scheduling constraints C, and a user modification U. It either returns a rescheduled timetable R satisfying all the constraints C and U as well as the forward-labeling constraints, or $R = \texttt{fail}$ if no such rescheduled timetable exists. Recall that T is a set of constraints over all variables and is in solved form (that is, in the form of $X = t$ where X is a variable and t a constant, and no variable appears in more than one constraint).

Now consider the rescheduling algorithm in Figure 4. The routine `reschedule()` takes as input a set C of rescheduling constraints and a solution T to C. It repeatedly accepts user modifications and reschedules the timetable accordingly. In a sense the user modifications are "accumulated" in successive rescheduling.

5. Heuristics

Since the rescheduling problem is NP-hard in general, generating optimal solutions is usually too time-consuming to afford. We are thus interested in sub-optimal solutions which have few service disruptions. In 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.

5.1. Smallest-First Principle

The smallest-first principle is designed towards generating a minimum-delay optimal timetable. Variables are first sorted in ascending order using the lower bound of their respective domains as key. Variables are then labeled according to the sorted order. Note that this is a static variable selection strategy since sorting is performed only once before labeling process begins. In labeling a variable, values are chosen using the standard

smallest-value-first ordering. This principle is founded on the assumption that a short delay on a train visit will cause a short delay on the subsequent one. In other words, delays are propagated in a monotonic fashion.

Figure 9 shows a valid timetable before rescheduling. A user modifies a journey (Figure 10) and some variables violate the scheduling rules. The timetable becomes invalid. Rescheduling with the smallest-first principle, variables are instantiated in ascending order of its value (Figure 11). In this case, eight variables of four journeys need to be changed. All of them need to be delayed 3 minutes (Figure 12). Therefore, the maximum delay is 3 minutes in this case. The smallest-first principle gives the minimum-delay optimal solution in this case.

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 13 shows a small segment of four journeys on a railway running map. The three journeys A, B, and C share the same track in Nanjing station, while the two journeys C and D take the same line in traveling from Nanjing to Longtan. Suppose a train dispatcher modifies journeys A and D (indicated by thick lines). Suppose further the station occupancy and the station exit rules enforce 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 dispatcher's modifications make the timetable infeasible since there is now insufficient time lag between journeys A and B.

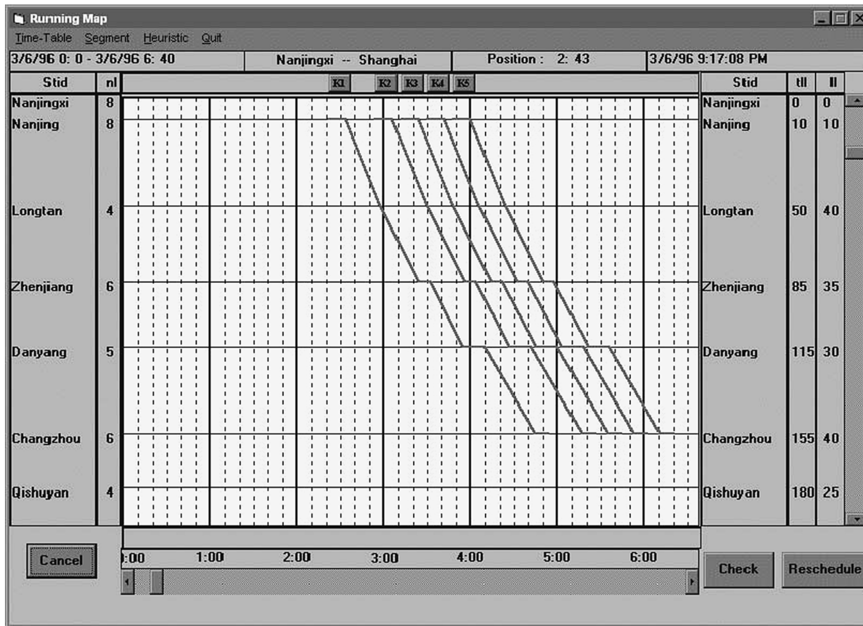

Figure 9. Smallest-first principle: Original timetable.

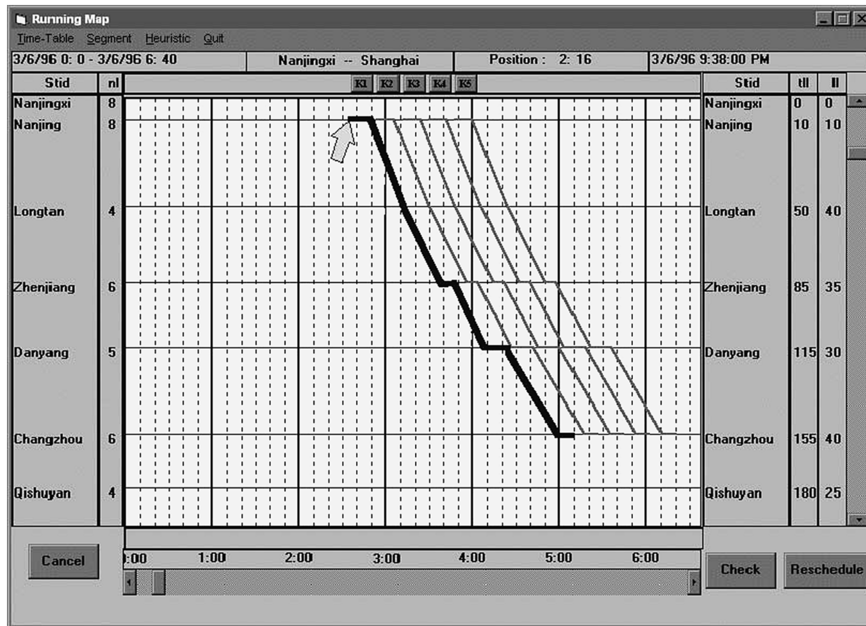

Figure 10. Smallest-first principle: After modification.

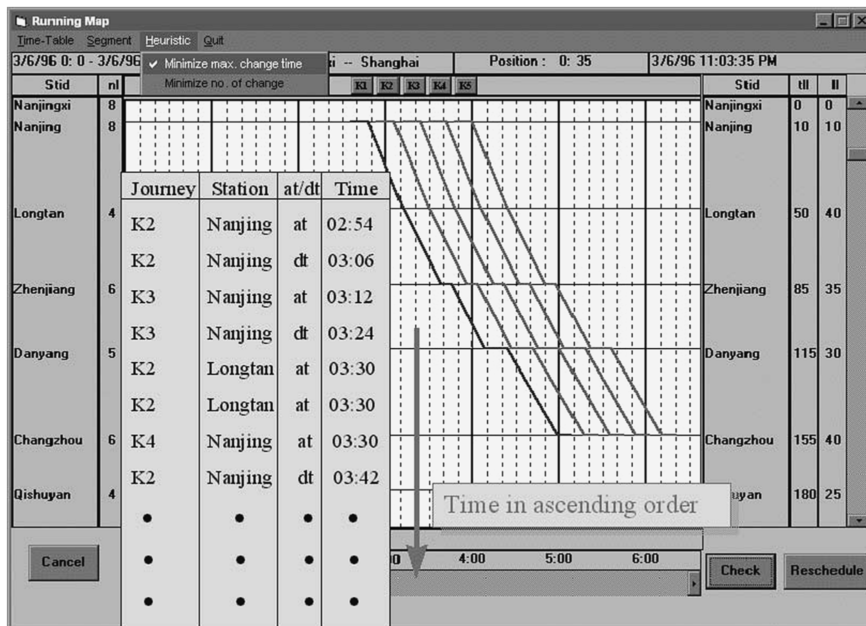

Figure 11. Smallest-first principle: Rescheduling.

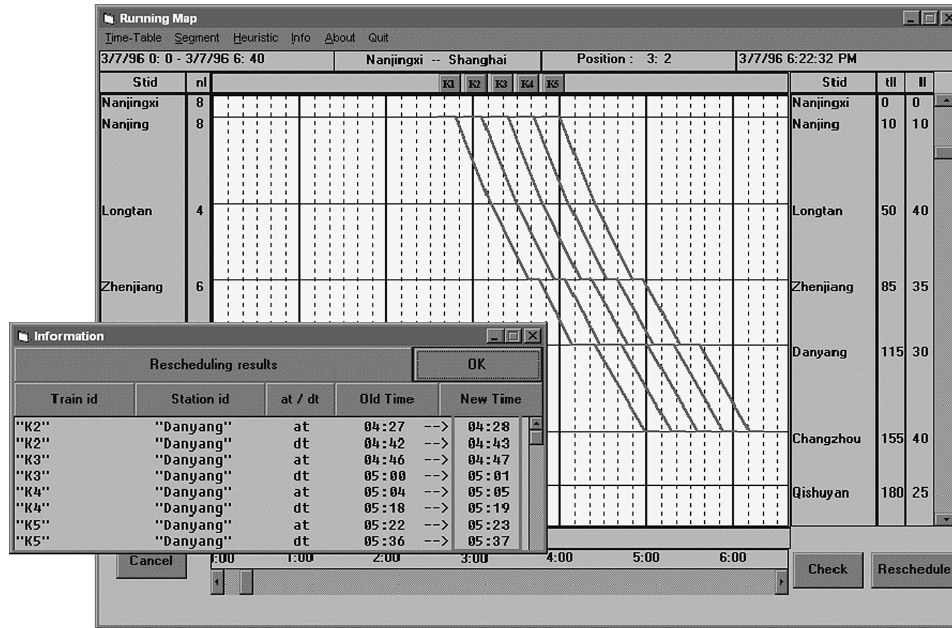

Figure 12. Smallest-first principle: Result generated.

Figure 14 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 the 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 15.

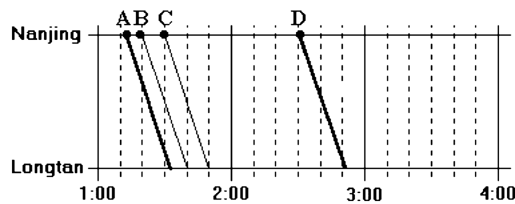

Figure 13. A non-optimal solution by smallest-first heuristic: Before rescheduling.

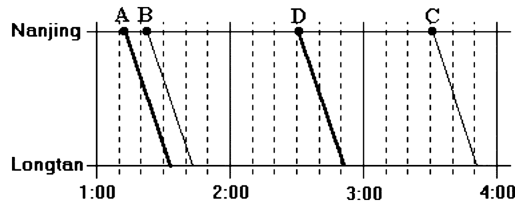

Figure 14. A non-optimal solution by smallest-first heuristic: After rescheduling.

5.2. Consistent-Assignment-First Principle

This heuristic aims to maintain as many variables with its original value as possible. First, variables are partitioned into two groups: non-conflicting variables and conflicting variables. *Non-conflicting variables (nonCONF)* are the ones whose values in the original timetable do not contradict the modifications imposed by the users directly. The other variables are *conflicting (CONF)*. The partitioning process is performed by examining each variable in turn and testing whether the variable conflicts with the user modifications. For efficiency reason, we further classify the non-conflicting variables into two groups. The first group $nonCONF_{share}$ contains variables that share journeys with one of the conflicting variables. The second group $nonCONF_{noShare}$ contains the rest of the non-conflicting variables.

The consistent-assignment-first heuristic suggests labeling variables in the following order: (1) the $nonCONF_{noShare}$ variables, (2) the *CONF* variables, and (3) the $nonCONF_{share}$ variables. Within each of the three groups, variables are sorted into ascending order and labeled according to the lower bound of their respective domains, as in the smallest-first heuristic. Again, partitioning and sorting are performed only once before labeling begins.

The idea of the heuristic is to label first those variables that can be instantiated with its time value in the original timetable, and backtrack into these variables last. The $nonCONF_{noShare}$ variables are the most plausible to retain their values in the original timetable. Although the $nonCONF_{share}$ variables are also non-conflicting, they are likely to have their values changed since they share journeys with *CONF* variables, the values of which must be changed. Thus, if the $nonCONF_{share}$ variables are labeled early, a large amount of backtracking will be induced when the *CONF* variables are labeled. Since the new values of the *CONF* variables will directly affect the new values of the $nonCONF_{share}$ variables, we label the *CONF* variables before the $nonCONF_{share}$ variables.

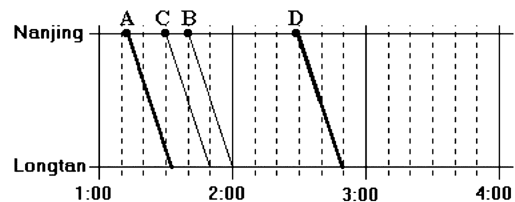

Figure 15. A smallest-delay-optimal solution.

Again, in labeling a variable, values in the variable domains are enumerated using the standard smallest-value-first ordering. This heuristic directs searching towards a minimum-change optimal solution. Note that a non-conflicting variable may still 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.

Figure 16 shows the modified timetable before rescheduling. Note that this timetable is the same as that in Figure 10. The rescheduling process first divides all variables into three groups. All variables on journeys $K3$, $K4$ and $K5$ are in $nonCONF_{noShare}$. Therefore, they should be instantiated first. This step tries to minimize the number of journeys moved, and thus moves towards a minimum-change optimal solution. There is only one conflicting (*CONF*) variable, the arrival time at Danyang of $K2$. All other variables on the journey $K2$ are in $nonCONF_{share}$ sharing the same journey with the conflicting variable. Therefore, we first instantiate the conflicting variable. Unfortunately, the earliest position that it can be placed without violating the scheduling rules is 5:40. Once the conflicting variable is instantiated to 5:40, all other variables on the same journey should also be delayed. From this example, we can see that backtracking is required if we instantiate the non-conflicting variables of $K2$ before the conflicting variable. And thus, our variable ordering can improve the efficiency in this case. From Figure 17, we can see that the solution generated by the consistent-assignment-first principle delays just one journey. It is obvious that this is the minimum-change optimal solution. Note, however, that there is no guarantee for the heuristic to generate an optimal solution in every test case.

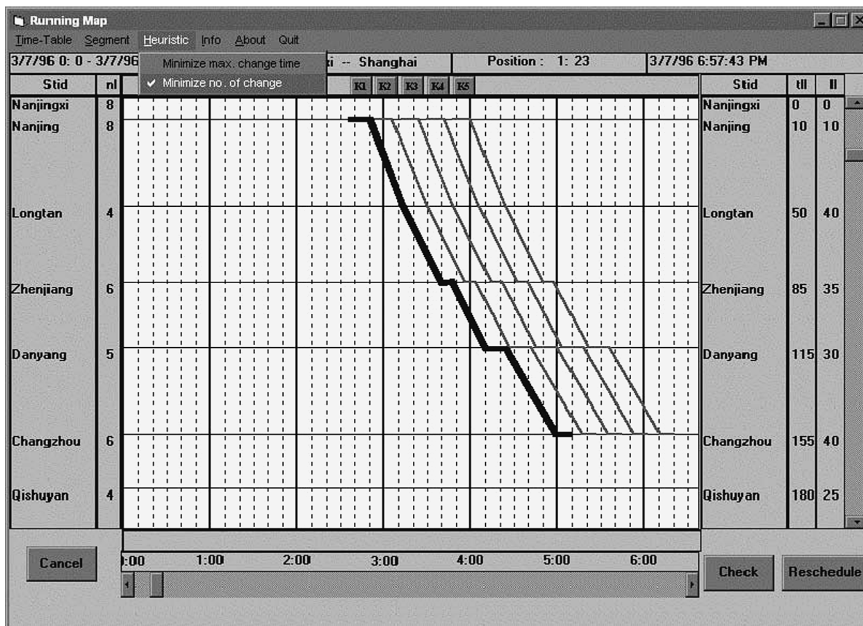

Figure 16. An optimal solution by consistent-assignment-first heuristic: Before rescheduling.

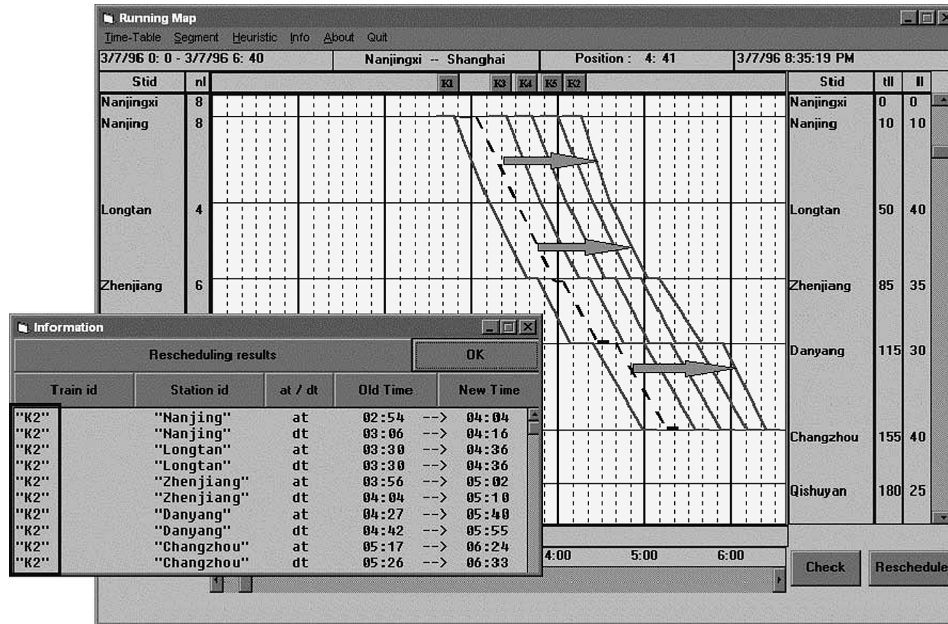

Figure 17. An optimal solution by consistent-assignment-first heuristic: After rescheduling.

Again, we apply this heuristic to reschedule the infeasible timetable in Figure 13. Recall that points A and D are fixed by the 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 minutes ahead of 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 solutions.

5.3. Comparison with Standard Heuristics

We compare our heuristics with two commonly used variable labeling heuristics: smallest-domain-first and most-constrained-first, both of which are direct application of the fail-first principle [26]. The *smallest-domain-first* heuristic selects the variable with the smallest domain at every labeling step; while the *most-constrained-first* heuristic picks the variable that is in the most number of constraints. For the small example in Figures 9 and 10, both the smallest-domain-first and the most-constrained-first heuristics produce the same answer as the consistent-assignment-first principle coincidentally. Using real timetables and more complex user-modifications, however, the smallest-domain-first and most-constrained-first heuristics almost always give worse answers than our proposed heuristics in both the minimum-delay and minimum-change optimal senses. In only a

small number of cases, these commonly used heuristics give the same answers as one of our heuristics, but the behavior is unpredictable. For example, the smallest-domain-first heuristic gives the same answer as our smallest-first heuristic in one case and gives the same answer as our consistent-assignment-first heuristic in another. These results can be expected since the smallest-domain-first and the most-constrained-first heuristics are designed for increasing search efficiency but not for achieving solution stability in any sense. In terms of search efficiency, we observe no substantial discrepancy between the two classes of heuristics although the fail-first heuristics are sometimes quicker in finding a solution. To conclude the comparison, we analyze the characteristics of the four heuristics in the following.

The smallest-first principle orders the variables using the chronology of station visits. The idea is to label the station visits in time-increasing order. Thus, variables of station visits with original time (lower bound of variable domains) immediately after the user modifications are labeled earlier. This heuristic has no direct relationship with variable domain size or constrainedness of variables. Given the fact that variables corresponding to station visits later in the day usually[3] have smaller domain size. The smallest-first principle can be approximated by the largest-domain-first heuristic, which always labels variables with the largest domain first. However, the smallest-first principle, employing only a static variable ordering, is more efficient than the largest-domain-first heuristic, which incurs overhead in maintaining variable ordering dynamically at every labeling step.

The consistent-assignment-first principle orders variables according to the degree of conflict of the variables' original values with the user modifications. The partitioning of variables depends on the distribution and the number of user modifications in the train timetable, as well as the scheduling constraints of the problem. Again, this heuristic has little relationship with variable domain size or constrainedness of variables.

The smallest-domain-first heuristic depends only on the size of the variable domains. As stated, variables corresponding to station visits later in a day usually have a smaller domains (although a variable corresponding to an arrival time usually has a smaller domain than the variable corresponding to the arrival time of the same station visit due to the stopover constraint). Thus, the smallest-domain-first heuristic labels variables in almost the opposite order of the smallest-first principle does.

The most-constrained-first heuristic depends on the number of constraints that each variable is involved in. In the rescheduling problem, each unmodified variable must be involved in a stopover-maintenance constraint, a forward-labeling constraint, a speed constraint, and a stopover constraint. Variables can be in different number of station occupancy constraints, station entry/exit constraints, and line time constraints. These constraints exist only if two trains share either a line or a track in a station. Effectively, the most-constrained-first heuristic order variables using the topology of train routing. It has no direct relationship with either the scheduling constraints or the user modifications.

In addition, both our heuristics enforce a static ordering of variables, which is determined at the start of the labeling process. The most-constrained-first heuristic adopts also a static ordering but it depends on the topology of the constraint graph of the CSP

being solved. In the smallest-domain-first heuristic, variable ordering changes dynamically according to the change in variable domain sizes effected by constraint propagation, which incurs a higher cost.

In conclusion, the standard fail-first heuristics described are designed as a generic heuristics for efficiency and takes no specific features of the rescheduling problem into account, while our heuristics are designed towards achieving the optimality criteria. The two classes of heuristics differ in purposes and thus also in behavior.

6. Prototype Implementation

In order to demonstrate the feasibility of our algorithms, we have re-constructed and enhanced the **PRaCoSy** running map tool prototype [38] 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 [27] 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 of the China railway. The rescheduled timetables generated by the two heuristics are explained. Two examples, to which our tool fails to respond in a timely manner, are shown.

The running map display (Figure 18) consists of six columns (regions). Column one and column four show the abbreviated identifiers of stations. Column two shows the

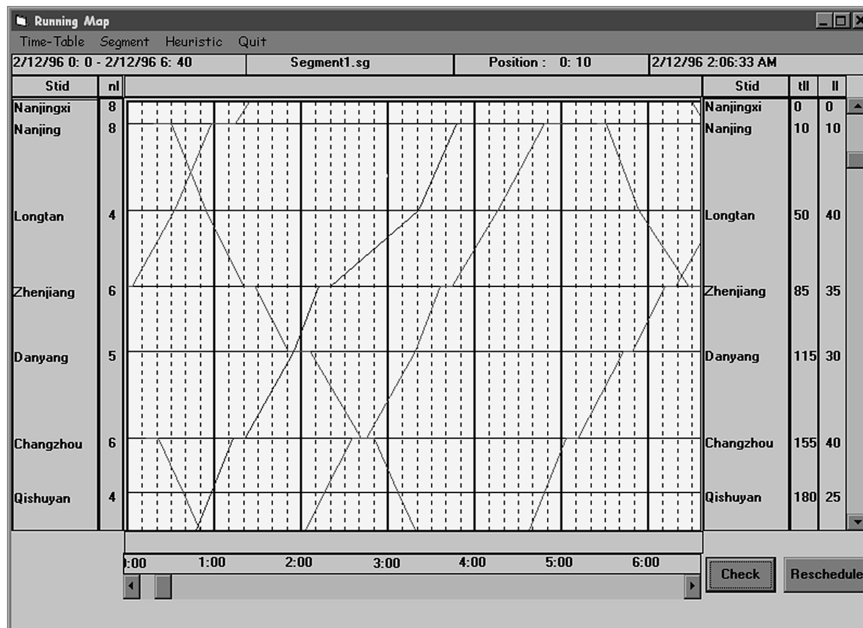

Figure 18. The running map tool.

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 are enabled.

A user can click-and-drag any lines to modify the corresponding train visits on the map. The modified timetable is validated using the verification algorithm when the "Check" button is pressed. If it is infeasible, a warning window, such as that shown in Figure 19, will pop up to display all constraint violations. At this point, the user can

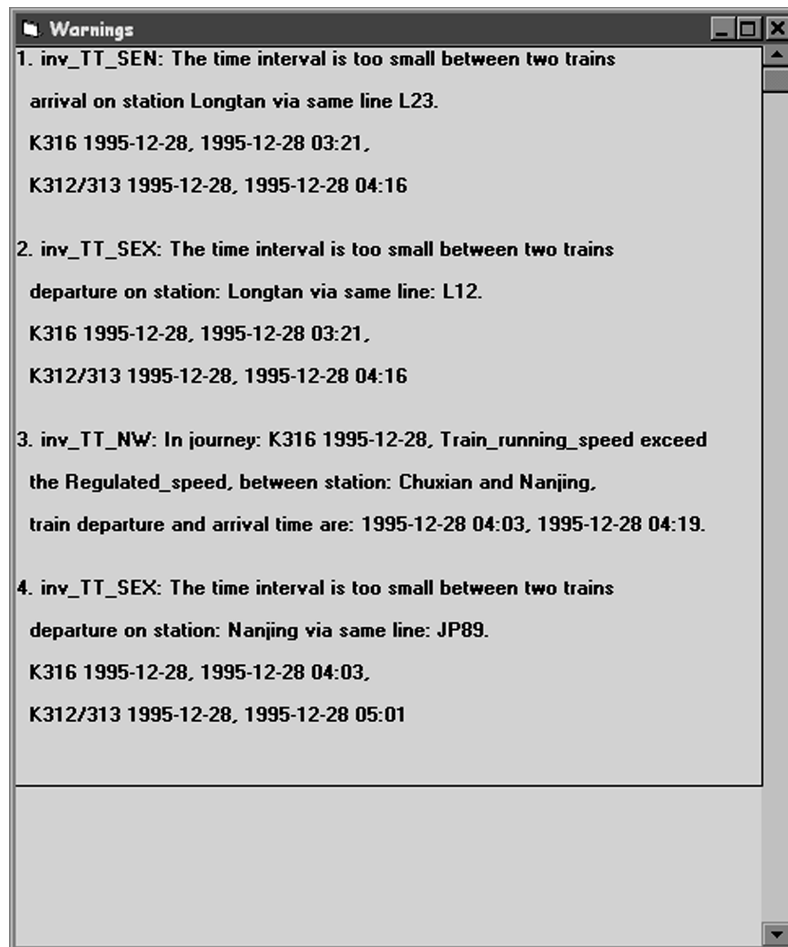

Figure 19. A warning window.

either invoke the rescheduling algorithm by pressing the “Reschedule” button, correct the modifications manually, or restore the original feasible timetable.

Figure 20 shows the segment from Nanjingxi to Shanghai of a China railway timetable, which amounts to 3307 constraints and 604 variables. Due to an accident, we have to delay the train departing from Zhenjiang at 0:03 to 3:30, yielding the map shown in Figure 21. 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 ten seconds. Figures 22 and 23 show the rescheduled timetable generated using the smallest-first principle and the consistent-assignment-first principle respectively.

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. In the

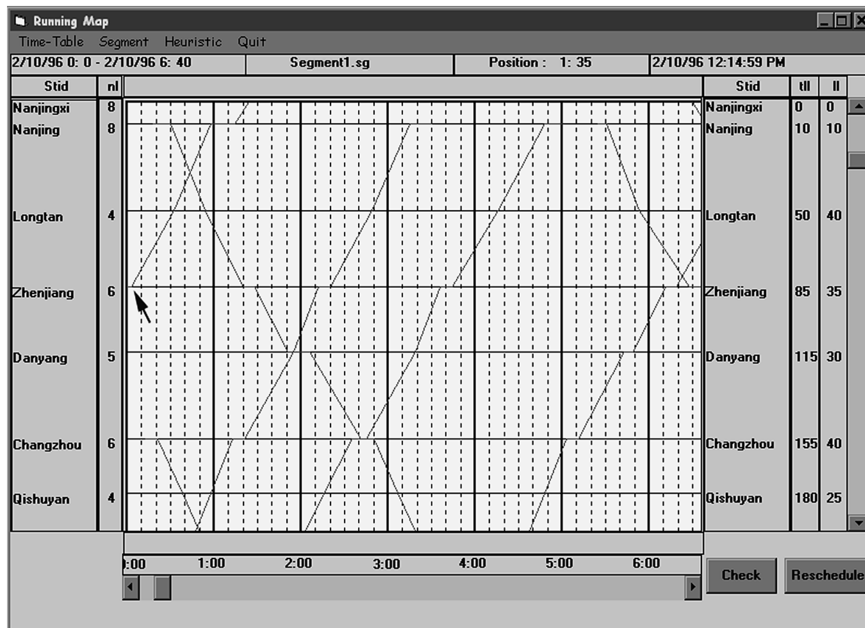

Figure 20. A comparison of two heuristics: Original feasible timetable.

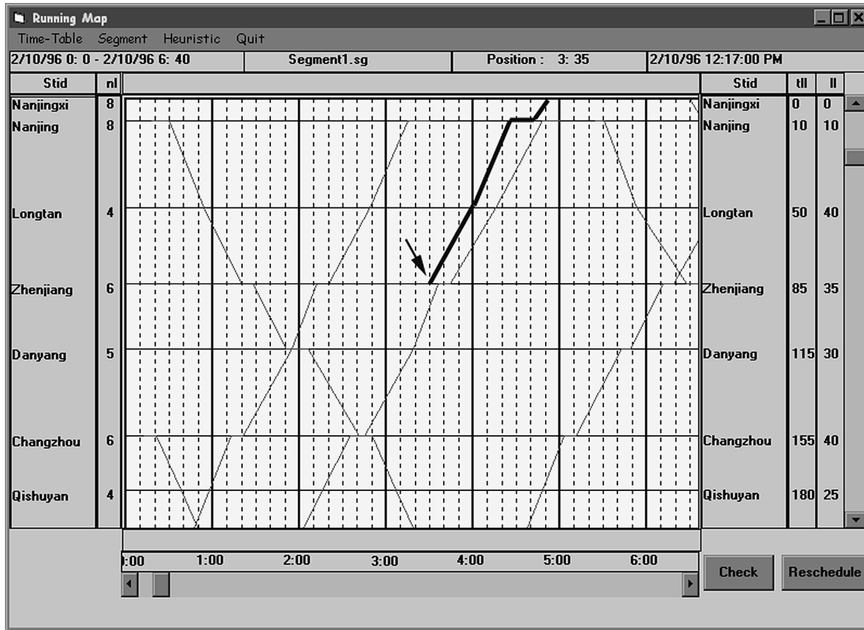

Figure 21. A comparison of two heuristics: Modified infeasible timetable.

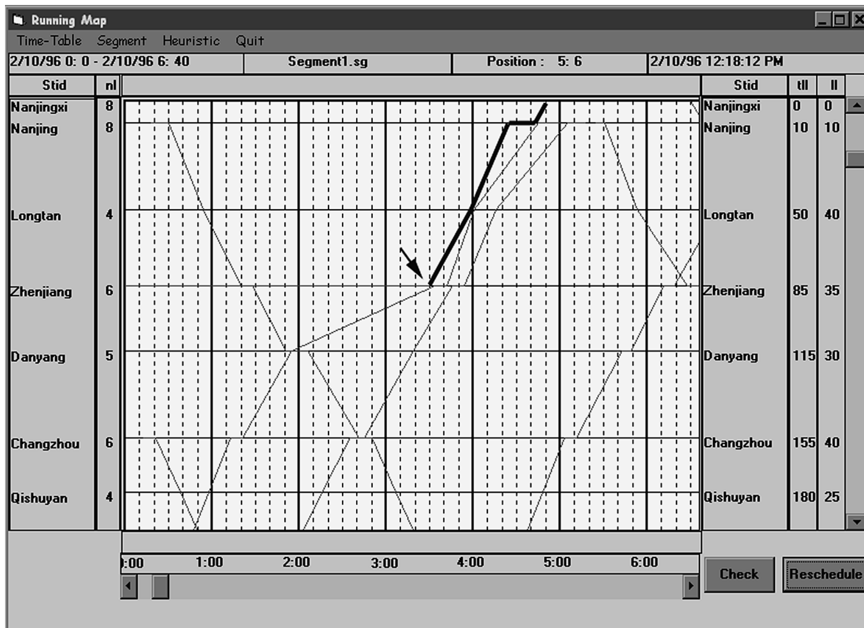

Figure 22. A comparison of two heuristics: Smallest-first principle.

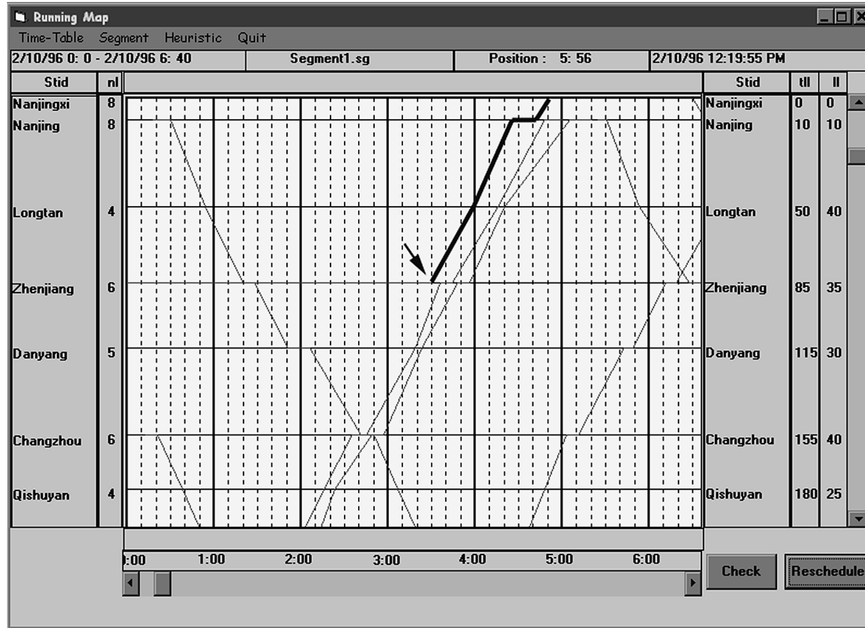

Figure 23. A comparison of two heuristics: Consistent-assignment-first principle.

rescheduled timetable, two journeys have been moved and the maximum delay among all the station visits is 90 minutes.

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 are labeled first. The conflicting variables, having to change their original values, are labeled last. Our experiments reveal that, in many cases, this heuristic produces a timetable which is "almost identical" to the original timetable. As seen in Figure 23, most of the journeys retain their original locations. Even for the right-shifted journey, its shape is mostly preserved. In the rescheduled timetable, only one journey has been moved. The maximum delay among all the station visits, however, is 116 minutes, which is greater than that generated by the smallest-first-principle.

Experimental results confirm that rescheduling can usually be completed within seconds. This is not always the case. Figures 24 and 25 provide two such examples. The infeasible timetable in Figure 24 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 25 is simply a non-repairable timetable. Neither heuristic can return promptly to confirm the unsolvability of the problem.

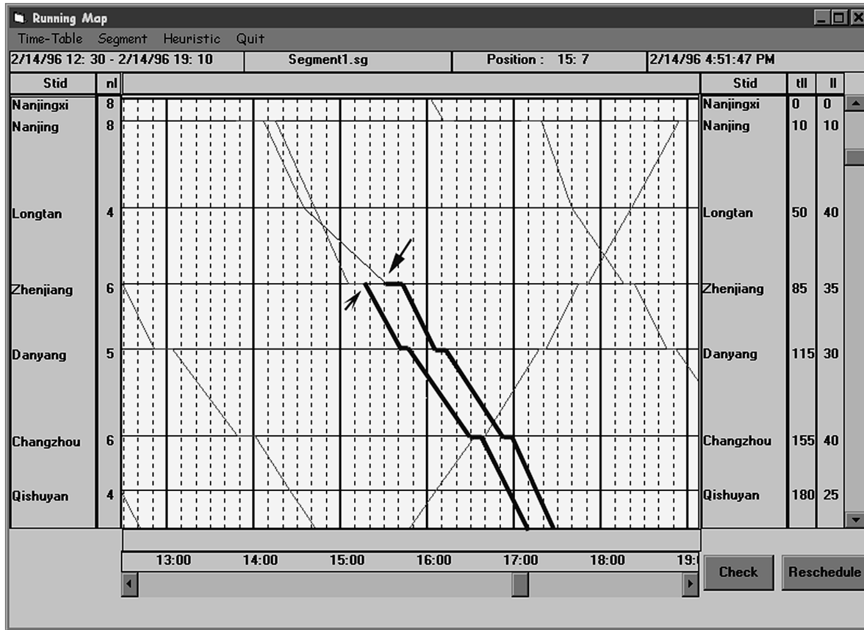

Figure 24. Poor performance examples: Infeasible timetable 1.

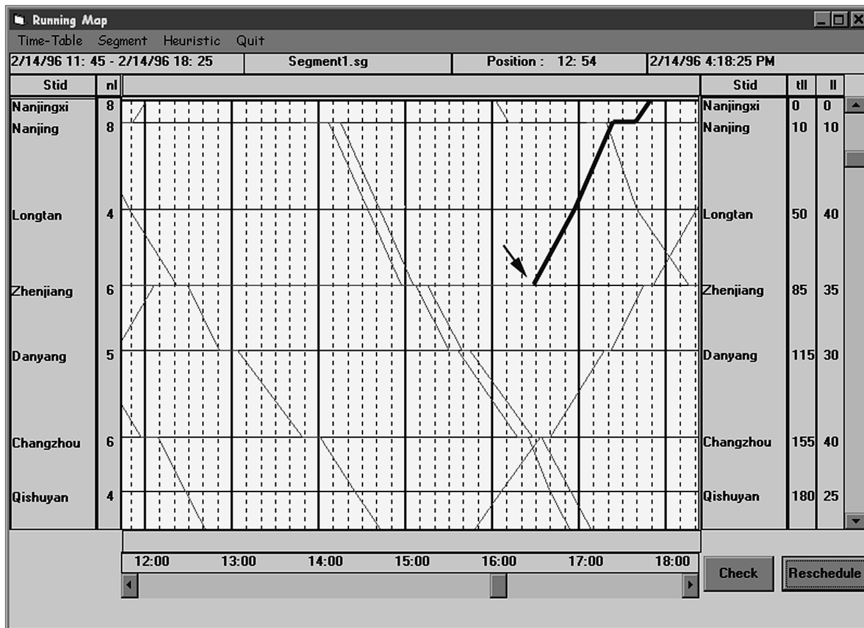

Figure 25. Poor performance examples: Infeasible timetable 2.

7. Concluding Remarks

The contribution of this paper is three-fold. First, we define formally train rescheduling as a constraint satisfaction problem. Algorithms for railway timetable verifications and rescheduling are then derived respectively 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 experiments using real-life data. Third, we have re-constructed and enhanced the **PRaCoSy** running map tool prototype.

Interesting future work includes studying different stochastic methods, such as GENET [14, 55] and E-GENET [34, 35, 36], for train rescheduling. We could also look at the possibility of accepting partial solutions [5, 21, 55] in the cases where neither heuristic is able to provide solutions. Work is also in progress to experiment our rescheduling method on larger-scale real-life railway timetables.

Acknowledgment

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 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 of the Second International Conference on Principles and Practice of Constraint Programming and the Constraints Journal for their constructive comments, which help to improve the final version of the paper.

Notes

1. A running map is a method of monitoring the movement of trains and rescheduling their arrivals and departures to satisfy operational constraints.
2. There are 1440 minutes in 24 hours.
3. This is not always true since the domain size of a variable also depends on the duration of a station stopover and the enforcement of the stopover constraint.

References

1. Baptiste, P., & Le Pape, C. (1995). Disjunctive constraints for manufacturing scheduling: Principles and extensions. In *Proceedings of the Third International Conference on Computer Integrated Manufacturing*, Singapore.

2. Bessière, C., & Cordier, M. (1993). Arc-consistency and arc-consistency again. In *Proceedings of AAAI-93*, pages 108–113.

3. Bisiani, R. (1987). Beam search. In S. C. Shapiro (ed.), *Encyclopedia of Artificial Intelligence*, pages 56–58. New York: Wiley.

4. Bitner, J., & Reingold, E. M. (1985). Backtrack programming techniques. *Communications of the ACM*, 18: 651–655.

5. Borning, A., Freeman-Benson, B., & Wilson, M. (1992). Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3): 223–270.

6. Carey, M. (1994). Extending a train pathing algorithm from one-way to two-way track. *Transportation Research*, 28B: 395–400.

7. Carey, M. (1994). A model and strategy for train pathing, with choice of lines, platforms and routes. *Transportation Research*, 28B: 333–353.

8. Carey, M., & Lockwood, D. (1995). A model, algorithm and strategy for train pathing. *Journal of Operational Research Society*, 46: 988–1005.

9. Cheng, Y. (1998). Hybrid simulation for resolving resource conflicts in train traffic rescheduling. *Computers in Industry*, 35(3): 233–246.

10. Chiang, T. W., & Hau, H. Y. (1995). Railway scheduling system using repair-based approach. In *Proceedings: Seventh International Conference on Tools with Artificial Intelligence*, pages 71–78.

11. Chiu, C. K., Chou, C. M., Lee, J. H. M., Leung, H. F., & Leung, Y. W. (1996). A constraint-based interactive train rescheduling tool. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 104–118.

12. Chiu, C. K., Chou, C. M., Lee, J. H. M., Leung, H. F., & Leung, Y. W. (1997). A constraint-based interactive train rescheduling tool. Technical report, Department of Computer Science and Engineering, The Chinese University of Hong Kong.

13. Colmerauer, A. (1990). An introduction to Prolog III. *Communications of the ACM*, July: 69–90.

14. Davenport, A., Tsang, E. P. K., Wang, C. J., & Zhu, K. (1994). GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proceedings of AAAI'94*.

15. Dechter, R., & Meiri, I. (1989). Experimental evaluation of preprocessing techniques in constraint-satisfaction problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 37–42, Menlo Park, CA. American Association for Artificial Intelligence.

16. Deville, Y., & Van Hentenryck, P. (1991). An efficient arc consistency algorithm for a class of CSP algorithm. In *Proceedings of IJCAI 1991*, pages 325–330.

17. Dincbas, M., Simonis, H., & Van Hentenryck, P. (1990). Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8: 75–93.

18. Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., & Graf, T. (1988). Applications of CHIP to industrial and engineering problems. In *Proceedings of the First International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems*, pages 887–892.

19. Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., & Berthier, F. (1988). The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan.

20. Dong, Y. (1994). The Zhengzhou $\leftrightarrow$ Wuhan train dispatch system. UNU/IIST PRaCoSy Document DYL/1/3, International Institute for Software Technology, United Nations University, July.

21. Freuder, E. C., & Wallace, R. (1992). Partial constraint satisfaction. *Artificial Intelligence*, 58: 21–70.

22. Fukumori, K., Sano, H., Hasegawa, T., & Sakai, T. (1987). Fundamental algorithm for train scheduling based on artificial intelligence. *Systems and Computers in Japan*, 18(3): 52–63.

23. Gaschnig, J. (1978). Experimental case studies of backtrack versus Waltz-type versus new algorithms for satisfying assignment problems. In *Proceedings of the Second Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, Canadian Information Processing Society.

24. Gaschnig, J. (1979). *Performance Measurement and Analysis of Certain Search Algorithms*. Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

25. Goldman, R. P., & Boddy, M. S. (1997). A constraint-based scheduler for batch manufacturing. *IEEE Expert—Intelligent Systems & Their Applications*, 12(1): 49–56.

26. Haralick, R. M., & Elliot, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problem. *Artificial Intelligence*, 14: 263–313.

27. ILOG. (1994). *ILOG: Solver Reference Manual Version 2.0*.

28. Jaffar, J., Michaylov, S., Stuckey, P. J., & Yap, R. H. C. (1992). The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3): 339–395.

29. Komaya, K., & Fukuda, T. (1991). A knowledge-based approach for railway scheduling. In *Proceedings: Seventh IEEE Conference on Artificial Intelligence Applications*, pages 405–411.

30. Kumar, V. (1987). Depth-first search. In S.C. Shapiro (ed.), *Encyclopedia of Artificial Intelligence*, vol. 2, pages 1004–1005, Wiley.

31. Kumar, V. (1992). Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1): 32–44.

32. Le Pape, C. (1994). Implementation of resource constraints in ILOG SCHEDULE: A library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3: 55–66.

33. Le Pape, C. (1995). Resource constraints in a library for constraint-based scheduling. In *Proceedings of the INRIA/IEEE Conference on Emerging Technologies and Factory Automation*, Paris, France.

34. Lee, J. H. M., Leung, H. F., & Won, H. W. (1995). Extending GENET for non-binary CSP's. In *Proceedings of the Seventh IEEE International Conference on Tools with Artificial Intelligence*, pages 338–343.

35. Lee, J. H. M., Leung, H. F., & Won, H. W. (1996). Towards a more efficient stochastic constraint solver. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 338–352.

36. Lee, J. H. M., Leung, H. F., & Won, H. W. (1998). A comprehensive and efficient constraint library using local search. In *Proceedings of the 11th Australian Joint Conference on Artificial Intelligence*, Brisbane, Australia.

37. Lin, H. C., & Hsu, C. C. (1994). An interactive train scheduling workbench based on artificial intelligence. In *Proceedings: Sixth International Conference on Tools with Artificial Intelligence*, pages 42–48.

38. Liu, X. (1994). A simple running map display tool. UNU/IIST PRaCoSy Document lx/tool/02, International Institute for Software Technology, United Nations University.

39. Mackworth, A. K. (1977). Consistency in networks of relations. *AI Journal*, 8(1): 99–118.

40. Mackworth, A. K., Mulder, J. A., & Havens, W. S. (1985). Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1: 118–126.

41. McGregor, J. (1979). Relational consistency algorithms and their applications in finding subgraph and graph isomorphism. *Information Science*, 19: 229–250.

42. Mohr, R., & Henderson, T. C. (1986). Arc and path consistency revisited. *Artificial Intelligence*, 28: 225–233.

43. Morgado, E. M., & Martins, J. P. (1998). CREWS_NS: Scheduling train crews in the Netherlands. *AI Magazine*, 19(1): 25–38.

44. Nadel, B. (1988). Tree search and arc consistency in constraint-satisfaction algorithms. In L. Kanal and V. Kumar (ed.), *Search in Artificial Intelligence*, pages 287–342, Springer-Verlag.

45. Nadel, B. A. (1989). Constraint satisfaction algorithms. *Computational Intelligence*, 5: 188–224.

46. Nuijten, W., & Aarts, E. (1995). A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *European Journal of Operational Research*.

47. Perrett, M. (1991). Using constraint logic programming techniques in container port planning. *ICL Technical Journal*, 7(3): 537–545.

48. Prehn, S. (1994). A railway running map design. UNU/IIST PRaCoSy Document SP/12/3, International Institute for Software Technology, United Nations University.

49. Prehn, S., & Bjørner, D. (1994). PRaCoSy: An executive overview. UNU/IIST PRaCoSy Document par/02/09, International Institute for Software Technology, United Nations University.

50. Prosser, P. (1991). Hybrid algorithms for the constraint-satisfaction problem. Research Report AISL-46-91, Computer Science Department, Univ. of Strathclyde.

51. Puget, J.-F. (1993). On the satisfiability of symmetrical constrained satisfaction problems. In *Proceedings of 7th ISMIS*, Trondheim, Norway.

52. Puget, J.-F. (1994). A C++ implementation of CLP. In *Proceedings of SPICIS 94*.

53. Sauer, J., & Bruns, R. (1997). Knowledge-based scheduling systems in industry and medicine. *IEEE Expert—Intelligent Systems & Their Applications*, 12(1): 24–31.

54. Schaefer, H. (1995). Computer-aided train dispatching with expert systems. In *Proceedings of the International Conference on Electric Railways in a United Europe*, pages 28–32.

55. Tsang, E. P. K. (1993). *Foundations of Constraint Satisfaction*. Academic Press.

56. Tsukada, T. K., & Shin, K. G. (1996). PRIAM: Polite rescheduler for intelligent automated manufacturing. *IEEE Transactions on Robotics and Automation*, 12(2): 235–245.

57. Tsuruta, S., & Matsumoto, K. (1988). A knowledge-based interactive train scheduling system. In *Proceedings of the IEEE International Workshop on Artificial Intelligence for Industrial Applications*, pages 490–495.

58. Van Hentenryck, P. (1989). *Constraint Satisfaction in Logic Programming*. The MIT Press.

59. Waltz, D. (1975). Understanding line drawings of scenes with shadows. In P. H. Winston (ed.), *The Psychology of Computer Vision*, pages 19–91, McGraw-Hill.

60. Wu, S. D., Storer, R. H., & Chang, P. C. (1993). One machine rescheduling heuristics with efficiency and stability as criteria. *Computers & Operations Research*, 20: 1–14.

61. Zweben, M., Davis, E., Daun, B., & Deale, M. J. (1993). Scheduling and rescheduling with iterative repair. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6): 1588–1596.