Solving Single-Track Railway Scheduling Problem Using Constraint Programming

by

Elias Silva de Oliveira

Submitted in accordance with the requirements for the degree of Doctor of Philosophy.

The University of Leeds
School of Computing

September 2001

The candidate confirms that the work submitted is his own and that appropriate credit has been given where reference has been made to the work of others.

Abstract

The Single-Track Railway Scheduling Problem can be modelled as a special case of the Job-Shop Scheduling Problem. This can be achieved by considering the train trips as jobs, which will be scheduled on tracks regarded as resources. A train trip may have many tasks that consist of traversing from one point to another on a track. Each of these distinct points can be a station or a signal placed along the track. Conflicts may occur when the desired timetable would result in two trains occupying the same section of the track at the same time.

The mapping of the problem we have proposed in this thesis is a more realistic approach when modelling the physical representation of a track railway. This is because we take into account the actual signals placed along the track delimiting each track segment. These signals control whether a train can or cannot go on that particular track segment, avoiding thus the possibility of trains running into each other. Previous authors adopted an approximation to what is found in practice by adding minimum headway constraints between trains into their model.

Two strategies for resolving the conflicts in a desired timetable are presented. The two strategies have their applicability in practice. For instance, resolving a conflict in the first strategy stems from the observed practice of train operators: in the first strategy a conflict is resolved by re-timing one of the trips at its departure time up to the point the conflict is resolved. Train operating companies do not typically want to plan for passenger trains being delayed after their departure.

On the other hand, the second strategy resolves a conflict by delaying only the conflicting piece of one of the two trips (and subsequent pieces of that trip). In this way, the section of the trip before the conflicting point does not need to be re-timed. This procedure yields solutions with lower total delay. Moreover, with this second strategy, we can also handle the usual practice with passenger trains of not delaying them after their departure, as well as situations in which a train may be delayed along the trip in order to resolve a conflict.

Furthermore, we present a group of practical constraints, incorporated into our software to solve the problem, that arise in real-life problems to which little attention has hitherto been paid.

Using the algorithms developed in this thesis, we are able to solve 21 real-life single-track railway scheduling instances of problems, 19 of them to optimality.

Acknowledgements

I would like to thank my supervisor, Professor Barbara M. Smith, for her great help and guidance in initiating me in research, having as a result this thesis.

I would like to thank my family for their support and encouragement throughout these years away from home.

I would like to thank my friends in the AI lab of the School of Computing and those from outside of the University for their support.

Additionally, I would like to thanks Wim Nuijten and Chris Beck for the internship opportunity they provided me at ILOG company in Paris in Autumn 2000. During my visit we had very insightful discussions throughout a project I participated in at that company. The professional advice I received during the project helped much to acquire more maturity with the tools I have been using in my thesis.

Finally, but not the least, my thanks go to my God for supporting me all along the way.

Declarations

Some parts of the work presented in this thesis have been published in the following articles:

- *E. Oliveira and Barbara M. Smith* 2001,
  Finding the Answer by Looking in the Wrong Place
  CP'01 Workshop on Modelling and Problem Formulation.

- *E. Oliveira and Barbara M. Smith* 2001,
  A Combined Constraint-Based Search Method for the Single-Track Railway Scheduling Problem
  10^{th} Portuguese Conference on Artificial Intelligence, EPIA'01.

- *E. Oliveira and Barbara M. Smith* 2001,
  A Hybrid Constraint-Based Method for the Single-Track Railway Scheduling Problem
  Research Report 2001.04, School of Computing, University of Leeds.

- *E. Oliveira and Barbara M. Smith* 2000,
  A Job-Shop Scheduling Model for the Single-Track Railway Scheduling Problem
  19^{th} Workshop of the UK Planning and Scheduling Special Interest Group – PlanSIG.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The deregulation which has recently occurred in the public railway sector in many parts of the world has increased the awareness in this sector of the level of service quality they need to offer to their customers [47, 54].

Under the pressure for improvements, computer tools have been developed to help planners in doing their work more efficiently and quickly [9, 31, 36, 78, 93]. In this scenario, the timetable plays a fundamental role in the management and operation of a public transport system, which is a highly complex and labour intensive task [31]. However, little research has been specifically carried out on train scheduling given the difficulty of dealing with numerous practical constraints.

Therefore, many of the proposed tools are of the form of an interactive *what-if* application, in which the goal of finding the optimal solution is relaxed, because short response times are often required. Although such a tool can be valuable, a train scheduling tool also capable of finding optimal solutions for the problem, within a reasonable time, is equally desired. Such a tool should, thus, be able to deal with the wide variety of complex constraints encountered in the practical train scheduling context.

Carey and Lockwood [21] commented that, in the traditional process, an approach to solving the train scheduling problem on a complex (multi-track) network topology might be to first decompose the network into railway corridors, and thereafter solve each corridor's problem individually. Each of these corridors consists typically of a set of single-tracks connecting a sequence of stations. Each train to be scheduled needs to be allocated to a corridor through the network initially; for instance, a corridor may be allocated trains running in one direction only. Resolving the train scheduling problem in this way may involve, afterwards, rescheduling other trains and, in addition, negotiations and trade-offs can take place between different corridors.

The result of the above rearrangements may result in an infeasible solution. This may happen, for instance, when two or more trains are planned to be simultaneously running on a stretch of the single-line which holds only one train at a time. Hence, the need for rescheduling may be due to the infeasibilities introduced into a train timetable by these rearrangements, or any other unforeseen situation which introduces infeasibilities into the timetable during operation. The primary objective of this approach is the production of a feasible timetable.

This thesis is, however, concerned with the single-track railway (re)scheduling problem where trains are only allowed to pass one another at stations, sidings and double-track sections. In many parts of the world, much or all of the rail network is single-track; for instance, many lines in Scotland are single-track, and there are single-track freight networks in Australia (see Higgins [45]). Therefore, the research in this thesis addresses a practical scheduling problem and moreover, could be used as part of a more general (re)scheduling system for a multi-track system.

The aim of previous work was basically to resolve these infeasibilities in the train timetable [58]. However, there are other practical constraints which train-operating companies would like to see also incorporated into a tool. This would provide them with the possibility of offering better planning services. Therefore, this tool must be capable of both resolving the basic occupancy infeasibilities and satisfying the additional constraints.

1.1 Thesis Outline

This thesis presents a model to map the single-track railway scheduling problem into a special case of the job-shop scheduling problem. In doing so, we may benefit from numerous techniques successfully applied to the job-shop scheduling problem.

In addition to the model, four practical situations are also presented in the train scheduling context which demand more sophisticated constraints, ignored so far in the literature.

The expressiveness provided by Constraint Programming (CP) languages allows us to deal with complex constraints in an easier manner than in other techniques, such as in mathematical programming. Therefore, in this thesis the CP paradigm is used to solve the single-track railway scheduling problem, and to code the four practical constraints presented.

This thesis along with providing new research in the area of train scheduling using CP, will give ideas on how to systematically use the concepts within CP to solve the problem focused on here or other similar complex problems. In what follows highlights the contents of each of the thesis chapters.

Chapter 1: Gives the motivation for this research and also gives the outline of this thesis.

Chapter 2: Introduces the train scheduling problem and some of the characteristics of the train scheduling features usually encountered in practice. This chapter goes on to describe the characteristics of the single-track railway considered within this thesis.

Chapter 3: The single-track railway scheduling problem is modelled in this thesis as a special case of the job-shop scheduling problem. In view of that, this chapter gives a brief overview of the literature of the job-shop scheduling problem. Some of the relevant aspects on this problem with respect to the problem tackled in this thesis are highlighted.

Chapter 4: Introduces some basic concepts of how to define a problem as a constraint satisfaction problem, and also how to find solutions for this problem by using constraint programming.

The CP technique is adopted in this thesis for its expressiveness in defining complex constraints, and furthermore, in keeping the problem set of constraints independent from the methods to solve the problem.

Chapter 5: Describes two slightly different proposed models to map the single-track railway scheduling problem into a special case of the job-shop scheduling problem.

Two complete algorithms and three heuristics are proposed to tackle the first model. For the second model, two methods of solving the problem are proposed. The first one uses one of the algorithms devised for the first model to systematically search for solutions. The second method is a combined method based on a local search heuristic and the complete algorithm used in the first method. The local search part of the algorithm is devised to use the critical path structure of the solution and iteratively improve the solution of the problem by swapping pairs of critical activities.

The numerical results show that, as well as dealing with some presented practical constraints, the CP approach proposed here is capable of scheduling real-world instances of the problem in a reasonable amount of time.

Chapter 6: Concludes with a discussion on the approaches applied in this thesis and some thoughts on their potential and future work.

Chapter 2

Single-Track Scheduling Problem

2.1 Introduction

A single-track network can be seen as an embryonic part of any other sort of railway network topology [21]. Therefore, the single-track line characteristics described in this chapter are as general in terms of features encountered in practice as possible. These features have to do either with the physical-constraint characteristics of the rail line, or with constraints to be handled in practical scheduling situation.

This chapter is structured as follows. In Section 2.2, some characteristics of a typical single-track railway network are described. Although these characteristics serve the purpose of specifying the single-line handled within this thesis, they can also help the understanding of some other classes of problems discussed in Section 2.3, a brief literature review of those more significant works related to this thesis. In Section 2.4 we present the summary of the chapter.

2.2 Describing a Single-Track Railway Network

A single-track railway network as considered in this thesis, consists of a set of stations, sidings and double-track sections $\mathcal{S}$, referred to as *passing points*; and a set of track segments $\mathcal{R}$.

Any two passing points along the single-track line are linked by a set of track segments, and at most one train can be on any track segment at any time. On

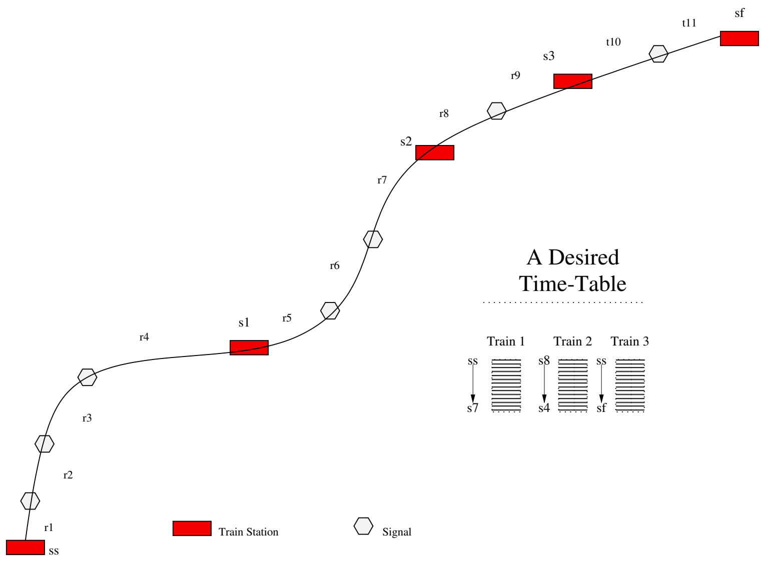


Figure 2.1: A Single-Track Railway network graphical example.

the other hand, a passing point has a specified capacity limit > 1 on the number of trains it can hold at any one time. Therefore, a conflict can occur when two or more train services are planned to use the same track segment at the same time. Likewise, at a passing point a conflict occurs when its capacity is exceeded.

The different types of passing points do, of course, differ in terms of functionality in important ways. Whereas stations are places where trains can also stop to be loaded, manoeuvre and change crew, at a siding a train can only stop or slow down in order to let another pass it. However, for the purpose of resolving conflicts between trains, these passing places will be considered in this thesis as a special case of stations, as already mentioned previously.

As stated above, between two consecutive stations there might be one or more track segments. It will also be considered that a track segment $r_i \in \mathcal{R}$ is necessarily delimited by two signals: one at the beginning and another at the end of the segment, which will control when a train either can or cannot proceed on that segment. This control is to avoid two trains running on the same segment at a time.

A desired timetable is given as data input by the network train-operating companies. For each desired train service the trip starting and ending points and the departure and arrival times are respectively specified. In Figure 2.1 is shown a graphical example of this sort of network in which 11 track segments: r_1, r_2, ..., r_{10}, r_{11}; and 5 stations: s_s, s_1, s_2, s_3 and s_f are presented.

A train trip is then described by a set of passing points $S_i = \{s_i, s_{i+1}, .., s_f\} \subseteq \mathcal{S}$

which a train must pass through. In this way a train is performing a task both when it is at a station, waiting for either loading or changing crew, and also when it is traversing track segments between two stations. Therefore, a trip can be viewed as a set of operations to be performed.

The given desired timetable may have conflicts, so that the trains need to be rescheduled in order to eliminate the conflicts. Conflicts may happen between trains running in opposite directions, but also between trains running in the same direction; for instance, a faster train which intends to overtake a slower one on a track segment.

The new timetable must be as close as possible to the first one. Hence, in order to eliminate a conflict one of the conflicting trips is chosen to be delayed. The process of resolving a conflict consists of holding up one of the conflicting trains at a passing point.

In addition to resolving conflicts between train trips, this thesis is concerned with handling more complex constraints which have been neglected so far in the literature. Therefore, the models (and solving programs) proposed in this thesis enable a planner to deal with a group of some special constraints such as:

1. *meet* (A, B, s_k, d) – provides that train A meets train B at station s_k and they stay there together for at least d time units;

2. *form* (A, B, d) – provides that the same vehicle used for train A will be used to form another trip planned for train B and d time units is the necessary time to set up the vehicle to perform the second trip;

3. *blocking* $(r_i,\ t_{initial}, t_{final})$ – provides that track segment r_i will be unavailable between time $t_{initial}$ and t_{final}.

4. *headway* (A, B, d_{AB}, d_{BA}) – provides that the separation between trains A and B must be at least d_{AB} time units when A is sequenced first, and d_{BA} time units otherwise.

It is necessary to stress the difference between the *headway* defined here and the headway already provided by the restriction which does not allow two trains on the same track segment at once. The former is to enforce a minimum time between two trains based on their characteristics, and the latter is to prevent trains from crashing. Furthermore, the headway defined above generalizes the concept of headway usually adopted by other authors. For instance, the headway defined here can be used to specify a larger minimum headway between a train carrying dangerous freight and a passenger train, if the freight train is scheduled first; a smaller minimum headway otherwise.

These constraints are specified, in this thesis, as part of the problem definition, and not as part of the program for solving the problem. Hence, the solving method is not tied to the constraints the problem has. This independence between the set of constraints and the method being used to solve the problem gives us great flexibility on combining different methods to solve the problem.

The single-track railway scheduling problem is a combinatorial problem and known to be NP-hard in the literature [19]. The number of possible resolved schedules is of order 2^{CO}, if CO is the number of conflicts in a given desired timetable [19, 45]. This is provided that each conflict has only two possible outcomes: either one of the two conflicting trains will be delayed.

Some work has been done in order to allow a computer tool to find, at least, a good feasible solution according to a criterion [19, 50, 51]. Other work has been directed towards finding an optimal solution [21, 46, 54, 80]. In this thesis the objective is also to find the solution with the minimum total delay according to the constraints posted.

2.3 Train Scheduling Literature Review

2.3.1 Introduction

In this section we review some of the literature works related to this thesis. Many of the previous works were basically based on a mathematical formulation for the problem, only recently some other approaches have flourished in the literature.

These works are grouped, in what follows, by their approaches: constraint programming and mathematical programming approaches. Our aim with this review is to describe some of the practical characteristics incorporated (or not) by the previous author(s), and compare with those provided by the framework proposed in this thesis.

2.3.2 Mathematical Programming Approaches

Although there is limited literature, one of the first works on the single-track railway scheduling problem goes back to the 70's, with the work by Szpigel [86].

Szpigel was the first to formulate the single-track railway scheduling problem as a disjunctive mathematical programming problem. The solution method employed was based on linear programming, incorporated within the standard branch-and-bound algorithm for machine scheduling problems. However, Szpigel did not apply any lower-bound techniques in his method. The method did not perform well on solving problems with 5 track segments and 10 trains.

In addition to the formulation, Szpigel was also the first to point out the similarity between the single-track railway scheduling problem and the job-shop scheduling problem, when the train velocity is fixed along the track segment.

Following the Szpigel disjunctive formulation, Jovanovic presented a mathematical formulation of a general minimum total weighted tardiness cost train dispatching problem. Jovanovic's model encompasses many practical constraints, such as the minimum dwell time to ensure that trains spend at least the scheduled amount of time at the points where they perform some work. Nevertheless, Jovanovic restricted his algorithms to solve instead the subproblem in which a fixed ordering of trains is pre-specified, and thus no intermediate overtaking is allowed. This restriction simplified greatly the algorithms for solving the subproblem.

The minimum headway constraints to separate two trains running in the same direction are handled in Jovanovic's work as part of the program specification. In this thesis, unlike in Jovanovic's work, the constraints to avoid trains crashing are specified as part of the single-line railway characteristics, as described in Section 2.2.

The way trains are handled along the trip has a great impact on their fuel consumption. In the light of this, Kraay *et al.* [55] proposed a more detailed model of a single-track network. In addition to keeping the delays at a minimum, the objective function aims at *pacing* trains so as to save fuel along a train journey. The fuel-consumption cost function proposed by them is related to the amount of work to move the vehicle on a track segment and takes into account: the grade of the track, the resistance of the track, the speed and mass of the vehicle.

In their model the speed of a vehicle is treated as a variable, and not fixed as is the case in this thesis. The authors claim that solving the '*where* trains will meet' (only resolving the conflicts) problem separately from the '*when* they meet' (controlling the vehicle speed) does not necessarily lead to the optimal solution for fuel saving. Therefore, their model is an approach to reconcile both objectives: the *where*- and *when*-trains meet.

Incorporating the fuel consumption into the objective function is a more general approach. This approach may fit well in medium to long term planning. However, the minimization of the tardiness of trains is a more realistic approach in the rescheduling context, because to keep a train as close as possible to the desired timetable is more important in operating passenger trains, for instance, than to conserve fuel.

An improvement on Jovanovic's work, combined with Kraay's model to save fuel, is presented by Higgins [46]. His model can handle the most common rail physical constraints: a specified dwell time for a train, priorities for trains, and minimum headway between following trains. Unlike in Jovanovic's work, Higgins' model allows overtaking of trains.

In addition to the above constraints, the lower and upper velocities for each train on each track segment are given. These constraints limit the traversal time of a train on a track segment and, in theory, for the whole trip.

As in Jovanovic's work, all these constraints are implemented as part of the solving program.

2.3.3 Constraint Programming Approaches

Some works on the application of constraint programming to Train Scheduling have appeared in the literature more recently. In [26] a rescheduling strategy is described to use within the PRaCoSy (People's Republic of China Railway Computing System), a system for the automatization of the Chinese Railways.

PRaCoSy is an interactive system which provides the user with the possibility of suggesting changes to a given timetable. These changes are considered by the rescheduling system as additional constraints for the problem.

The main aim of this system is to provide the user with a feasible solution. When this is not achievable by just propagating the constraints of the problem, including those constraints introduced by the user, a depth-first algorithm is devised to search for a solution. The system allows the user to stop the algorithm at any time, if the program takes too long to reach any feasible solution, and the user has to reconsider their changes, in this case.

A general topology of the network is considered for the system, and six sorts of constraints are handled by the system.

1. Speed Constraints – These constraints enforce the average maximum speed limit for a vehicle on a particular track segment.

2. Station Occupancy Constraints – Despite the name *Station* in the title, these constraints give the time separation between two trains which use the same track segment, *i.e.* a minimum headway constraint.

3. Station Entry Constraints – These constraints are to enforce a time separation between two trains entering a station via a line.

4. Station Exit Constraints – These are similar to the previous sort of constraints, but now for the exit situation.

5. Line Time Constraints – These constraints are for two kinds of purposes. The first guarantees that two trains travelling in the same direction will not overtake one another. The second enforces that if there are two journeys on a line in opposing directions, the line must be unoccupied for a specified period of time.

6. Stopover Constraints – These constraints enforce the minimum dwell time for a train at a station.

Another related work to this thesis is that by Kreuger *et al.* [58]. Their work is based on using Constraint Logic Programming to solve the Single-Track Railway Scheduling problem by mapping it as a Job-Shop Scheduling problem, as in this thesis. However, their work aimed only at finding a good feasible schedule by eliminating the inter-train conflicts, rather than going beyond that to introduce new constraints which might be of interest in a practical context. Moreover, the number of conflicts resolved for the set of trips exclusively on the single-line section of the network is not revealed. Therefore it is difficult to compare their results. Furthermore, to avoid dealing with a great number of tasks to be scheduled, Kreuger *et al.* [58] adopted a simplification which consists of modelling the problem as having only one track segment between any two stations. However, a headway constraint was imposed to allow any two trains running in the same direction to use the same track, as long as there was a minimum time distance between them.

In [58] the unknown is how much waiting time is to be given for each trip at each station in order to eliminate any possible conflict. The system can be set by the user to constrain the overall delay, the total time for each trip and, also, the makespan. All these constraints may be used as features to prune the search tree.

No attempt was made in [58] towards controlling the maximum number of trains allowed to stay simultaneously at a station. This could result occasionally in an unrealistic schedule if a number of trains are scheduled to a station which cannot hold them all at any one time.

More recently, we can find the work by Isaai and Singh [50, 51]. They describe a Hybrid Constraint-Based Heuristic and Tabu Search to solve a single-line network scheduling problem, which focused basically on finding quick solutions for the problems tackled rather than searching for the optimal one.

Three well known criteria were used to analyze the quality of the schedule during the experiments [71]. The first is the Average of Unit Waiting Time (AUWT). In order to avoid the drawback of having very good schedules for a group of trains but bad schedules for others, another function is also applied, the Maximum Ratio Waiting Time (MRWT). The third cost function applied is the Maximum Ratio of Waiting time to Journey time (MRWJ).

The third cost function takes into account the importance of each train by a weight measure. Regardless of which function is used, it is claimed the system outperformed the expert planners by about 5% in the worst problem result and about 27% in the best problem result. In any case this improvement means time-saving, which could allow the operators to introduce more services on the network.

The model used by Isaai and Singh in [50, 51] does not allow two trains to simultaneously occupy a stretch of single-track between two consecutive stations, whereas in our model, this section of track might be divided into several track segments, delimited by signals, with several trains running in the same direction at once. Hence, in their model, the first train must complete its arrival at the next station before the next train departs from the previous station. This rule leads to a poorer solution, as slower trains can hold up other faster trains while traversing from one station to another.

2.3.4 Discussion

In our model, a more general and realistic representation of a single-line railway is considered. For instance, conflicts are resolved by Higgins in either one of two places: double tracks or *sidings*. In Higgins' model a siding is conceived as a station where a section of track is placed aside to be used for crossing and passing of trains. Therefore, this conception of a passing point is more restrictive than the idea of a passing point adopted in this thesis, as described in Section 2.2. As a result, Higgins cannot handle more than one train stopped at a station. In short, in Higgins' model any passing point can only hold 2 trains at any time [45], but one of them must be just passing through.

The headway constraints are modelled by the previous authors as a minimum time interval between trains. Therefore, this is only an approximation to what is found in practice. A more realistic approach is to consider the signals placed along the track delimiting each track segment. These signals control whether a train can or cannot go on that particular track segment avoiding, thus, the possibility of trains running into each other.

The headway in our model is specified as a characteristic of the single-line railway being considered, therefore, not as part of the solving program used to tackle the problem.

As in Higgins' model [46] and in this thesis, Isaai and Singh [50] also resolve conflicts at stations and double-track segments. Isaai reports that the largest problem solved is one with 22 trains, 51 stations, 10 double-track segments along the track, and 62 conflicts on single-track sections. If we consider each double-track segment as an abstraction of a station s_i with $capacity(s_i) = 2$, we will notice that in [50, 51] the ratio of conflicts to stations is nearly 1 for the largest problem, and 1:3 conflicts per station on average.

Higgins [46] discusses the effect of the increase in demand through additional trains and the reduction in the number of passing points available for conflict resolution. He claims that adding trains to the schedule will cause a linear increase in the number of conflicts per train. However, he also observed that the increase in average conflict delay is non-linear when reducing the number of the passing points from 14 to 4 in one of his problems. The average conflict delay per train increased more rapidly as the number of passing points approached 4 passing points.

The number of conflicts a problem has, as well as the number of passing points to resolve them, can have a great impact on the time to resolve such a problem. For instance, it is reported in [46] that a problem with 31 trains, 52 conflicts, 12 stations was solved in about 50 seconds, whereas introducing another 18 trains, giving in total 49 trains, 100 conflicts, and the same number of stations, increases the time to solve the problem to optimality to 10 minutes. The problems we solve in this thesis have a larger ratio of conflicts per station than those in Isaai and Singh [50].

With the problems used for the experiments in this thesis, the ratio is larger than 2 conflicts per station for the most difficult problem and not less than 1 on average; and so the problems we are solving here can be expected to be more difficult than those solved by Isaai. Furthermore, the objective in Isaai's work is basically to find quick solutions for the problems tackled rather than necessarily searching for the optimal one.

The models proposed in this thesis have also advantages over that presented by Jovanovic. This is because, although Jovanovic [54] presents a general problem model, a restricted subproblem is solved instead. In order to ease the algorithms for his mathematical formulation he restricts the general problem to that in which trains have a fixed ordering of overtaking, given from the outset. This limitation does not exist in our model.

2.4 Summary

This chapter describes a typical single-line network. The described network's characteristics are meant to be as general as possible so that some constraints existing in practice are not neglected in solving the model proposed in this thesis.

A set of special constraints are described, which are implemented and can be handled by the current version of the program that solves the problem sets used in this thesis. These constraints are specified as part of the problem definition, and not as part of the program for solving the problem. Because of this independence we have great flexibility in applying more general methods to solve the problem, as the methods are not tied to the type of constraints they can deal with.

In this chapter, in addition, we briefly review some of the previous works related to the subject of this thesis. We also discuss some of their limitations which are avoided in the models proposed in this thesis.

Chapter 3

Scheduling Problems

3.1 Introduction

Scheduling is a decision-making process which concerns the allocation of limited resources to tasks over time, and has a goal to optimize the objective(s). The idea of resource and tasks can take many forms. Typically, the resource may be machines in the manufacturing context, stretches of a track in the train scheduling context, but also lecturers in the university context. Tasks may be operations to be carried out in a production process, the train running from a point to another on a track, or lectures to be performed.

Scheduling and sequencing problems have been studied for many decades. One of the first pioneers was Henry Gantt (1861–1919). However, one of the first publications in operational research literature was in the early 50s in the *Naval Research Logistics Quarterly* [53]. In the late 1960s and 1970s the area had a sharp growth, and has retained its momentum ever since. In the 80s the area had another stimulus because of the use of personal computers permeating manufacturing facilities. From that time on, many scheduling systems started to be developed to provide the user with an automated scheduling tool [71].

Over these years of fertile research in the scheduling field, some problems in this area were formally classified and defined. In this thesis, we will be interested in the job-shop problem. This is because of its close similarities with the model we are proposing for the problem being tackled in this thesis: the single-track railway scheduling problem.

The aim of this chapter is to support the following chapters with some elements and terminologies from scheduling field and, in particular, from the job-shop scheduling problem literature. It is not possible to cover in this chapter the huge research effort devoted to the job-shop scheduling problem over the last 40 years, thereby a brief overview of a couple of important works related directly or indirectly to this thesis were chosen to compose the basis for this text. Hence, the remainder of the chapter is organized as follows. The next section formally defines the job-shop scheduling problem. A variation from the classical job-shop definition is also presented so as to be applied later on when describing the single-line railway model.

In Section 3.3, a disjunctive graph model is given for the problem. Section 3.4 describes the important role a set of activities play in a job-shop problem feasible solution. The *critical activities*, defined in Section 3.4, form an important feature: the *critical path*. This is often used to improve solutions on job-shop scheduling problem by swapping the order of critical activities. In Section 3.5, some exact and approximated search techniques in the literature for solving this problem are discussed. The summary of this chapter is presented in Section 3.6.

3.2 Job-Shop Scheduling Problem

The job-shop scheduling problem is one of the class of combinatorial problem most studied in scheduling area by the OR community over the last decades, as stated in the prior section, and is usually defined as follows. Given are a set of jobs J. For each job $J_i \in J$ a set of operations $J_i = \{o_{i1}, o_{i2}, \ldots, o_{ik}\}$ is also specified. These operations have all to be processed in a fixed order, and each of these tasks requires processing uninterruptedly (*non-preemptively*) on a unique machine $r_i \in \mathcal{R}$, the set of machines. Two operations o_{ij} and $o_{i'j'}$ executed by the same machine cannot be simultaneously processed. The fixed processing time p_{ij} for each operation o_{ij} is also given as input for the problem. In addition, each job has its release date d_i and its due date c_i, corresponding to the expected completion time of the activity $o_{i|J_i|}$. A schedule consists of finding a start time for each operation.

The most popular objective to minimize when searching for feasible schedules is the maximum of the jobs' completion time, or *makespan*. However, many other functions can be used to gauge the quality of a given solution [71]. For the purpose of this thesis, the *total tardiness* will be also considered. Therefore, if C_i denotes the actual completion time for job J_i, then the tardiness T_i of job J_i is defined as

$max(C_i - c_i, 0)$. The aim, when using this alternative measurement criterion, is to minimize the total tardiness given by the expression:

$$D = \sum_{1 \leq i \leq n} T_i; \tag{3.1}$$

i.e. D is the overall delay of jobs.

Only a few particular subclasses of this problem have been proved to be polynomially solvable. Therefore, the general problem is NP-complete in the strong sense [37]. The job-shop scheduling problem is one of the most computationally intractable combinatorial problems which have been tackled in the literature so far.

In the next section an analogy of this problem to a graph structure is presented. This analogy is often used to also represent the relations both between the sequential operations within a job, and the operations to be performed on each machine, which cannot be concurrent.

3.3 The Disjunctive Graph Model

The job-shop problem can be elegantly represented by a disjunctive graph [6]. Consider a directed graph $G = (N;\ A, B)$, where N is the set of vertices representing all the processing operations o_{ij} to be performed for each J_i job of J. A is a set of *conjunctive* directed arcs, representing the sequence of operations on a particular job, whereas B is the set of pairs of *disjunctive* arcs linking tasks being performed on the same machine.

The conjunctive arcs of A represent a series of precedence constraints given as part of the problem so that, for each pair of operations $(o_{ij}, o_{i(j+1)})$ of job J_i, both operations must be performed, o_{ij} must be performed before $o_{i(j+1)}$, and there is an arc $(o_{ij} \rightarrow o_{i(j+1)})$ in A.

Two operations from different jobs which are to be performed on the same machine are linked by two directed arcs going in opposite directions. The semantic of each of these *disjunctive* pairs of arcs is that one arc of each pair is to set one possible precedence order between two pair of tasks as $(o_{ij} \rightarrow o_{i'j'})$, and the other arc is to set the alternative order $(o_{i'j'} \rightarrow o_{ij})$. One of the two alternative orders is chosen in this way until an order between all the tasks on the machine is determined, when only one arc of the pair of arcs is left in the graph. The disjunctive arcs, in set B, form $\mid \mathcal{R} \mid \times\ B_k$-cliques of double arcs, one clique for each machine.

On one hand the conjunctive arc $(o_{ij}, o_{i(j+1)}) \in A$, linking two consecutive operations of job J_i, is associated with the processing time p_{ij} of task o_{ij}. On the other hand, the disjunctive pair of arcs linking two operations o_{ij} and $o_{i'j'}$, the arc $(o_{ij} \rightarrow o_{i'j'})$ is associated to the processing time p_{ij}, and the second arc $(o_{i'j'} \rightarrow o_{ij})$ is associated to $p_{i'j'}$.

In order to illustrate the disjunctive graph model, consider that three jobs are given, as shown in Table 3.1.

Jobs	Machines' Sequence	d_i	processing times	c_i
1	r_1, r_2, r_3	0	$p_{11} = 11, p_{12} = 9, p_{13} = 5$	25
2	r_2, r_1, r_4, r_3	0	$p_{21} = 9, p_{22} = 4, p_{23} = 6, p_{24} = 7$	26
3	r_1, r_2, r_4	0	$p_{31} = 5, p_{32} = 8, p_{33} = 4$	17

Table 3.1: A 3×4 job-shop problem example.

In this particular example every job can start its execution from time 0. The first task of Job J_2 must be performed on machine r_2, and requires $p_{21} = 9$ time units to be completed once started. The second operation of this job, o_{22}, is to be performed on machine r_1. This machine is also required by task o_{11} and o_{31} of jobs J_1 and J_3, respectively. This dispute for machine r_1 is represented in the graph in Figure 3.1 by double arcs (dashed lines) between these three activities.

Note that operations o_{23} and o_{33} are to be performed on the same machine, r_4. Therefore double arcs are placed between them. The processing time p_{23} is associated to one of the arcs meaning the execution duration of o_{23} when it is performed before o_{33}. The other arc means the alternative schedule order.

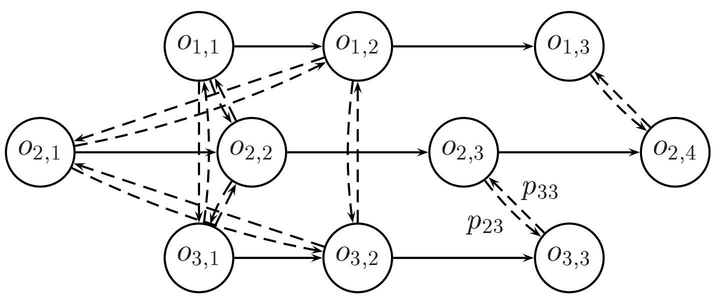


Figure 3.1: A disjunctive graph showing only the conjunctive and disjunctive arcs of the problem.

When the objective is to minimize the makespan, two dummy nodes are added to the graph in Figure 3.1. One node is to represent the source, from which the first task of each job will be connected by a conjunctive arc. The other node is to represent the sink, to which the last task of each job will be connected by a conjunctive arc.

However, in this thesis a different criterion is used to measure the quality of a solution. Our interest is in the total tardiness cost function, and a slightly different procedure is employed with respect to the addition of dummy nodes [81]. Only one dummy source S node is added to the graph G, as before. Nevertheless, with the total tardiness objective each job's completion time is to be taken into account. Thus, $n = \mid J \mid$ dummy nodes N_i, representing sink nodes, are added to the graph. Each of these nodes denotes the completion time of job J_i.

Whereas in Figure 3.1 a representation of the conjunctive and disjunctive arcs of the graph is given for the problem presented in Table 3.1, in Figure 3.2, the dummy nodes are introduced to the same problem when the total tardiness is to be minimized.

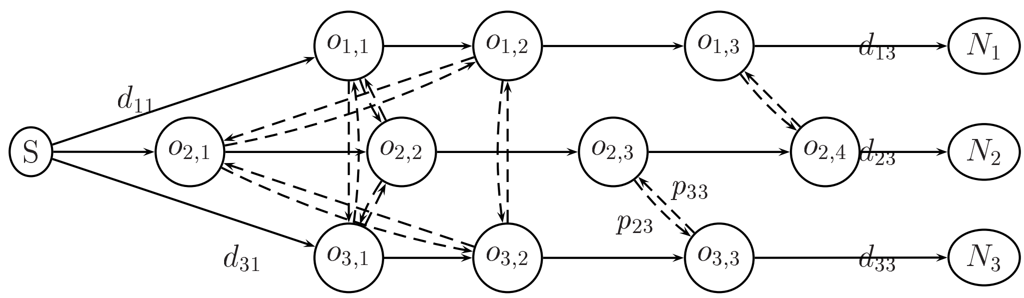


Figure 3.2: A disjunctive graph for a total tardiness minimization problem.

3.4 Critical Path

The *critical path* is formed by some particular operations. Given a feasible solution, the activities whose start time cannot be postponed without also changing the value of the objective function [71] are called *critical activities*, and the set of critical operations is referred to as the *critical path*. This criterion can also be applied when the total tardiness is the objective function.

A simple algorithm to find a critical path for a job-shop scheduling problem can be devised by scheduling one operation at a time starting from the earliest

$d = d_i$ release time. Whenever an operation has been executed, start all operations for which all predecessors have been completed. Thereafter, going backward from an activity with the latest completion time c_i, whose completion time gives the makespan for the problem, work toward time d, respecting the precedence relations existent in the problem. This way, those operations whose earliest starting times are equal to the latest starting times are the critical operations. They form a set of blocks of adjacent operations on the machines they are performed.

3.5 Solving the Job-Shop Scheduling problem

Most of the methods proposed for solving the job-shop scheduling problem are based on an enumerative algorithm, and the disjunctive graph formulation presented in Section 3.3 is often used. Nonetheless, many other strategies have also been proposed. Some of them are based on a Mixed Integer Programming (MIP) formulation of the problem, some on Constraint Programming, and others use some meta-heuristics framework to tackle the problem. In the following sections, some of the methods of interest in this thesis are briefly reviewed.

3.5.1 Complete Methods

The complete methods typically used in the literature use basically a *depth-first* search tree strategy. This sort of strategy is such that it goes on sequencing one task after another, constructing this way a partial schedule up to the point where no task is left to schedule. Every time a decision is made, with respect to a sequence of a task, is referred here to as a node in the search tree.

The algorithm steps back from any node to the latest schedule decision made any time the upper-bound constraint to the solution cost is broken, or other any constraint posted to the system. By *backtracking* to a previous decision, the algorithm tries to find the node in the search where a *wrong* decision was made, in order to choose an alternative scheduling decision, if there is one. The search stops once all the nodes have been implicitly or explicitly explored.

This sort of algorithm implicitly considers all the possible schedules to a given problem. However, its drawback is that it typically does an enormous number of computations both to reach a solution within a given upper-bound for the solution cost, and to prove the optimality of a solution. In a (n-jobs,m-machines)-job-shop problem there are $(n!)^m$ possible assignments, *i.e.* sequences of tasks on each machine. Even though many of these assignments will not be feasible owing to precedence and disjunctive constraints, complete enumeration of all feasible sequences to search for the optimal solution is simply impossible in practice.

Therefore a branch-and-bound algorithm is often used as an alternative, to reduce the number of search nodes to be generated. Unlike the simple depth-first algorithm, the branch-and-bound applies a *branching* scheme, and therefore expanding only those nodes which are likely to lead to the goal solutions. Another feature of a branch-and-bound algorithm is the bound(s) scheme. This sort of scheme dynamically puts limits to an expanded node so that the algorithm is forced to immediately backtrack from that node in the search if some of the limits are bound to be exceeded because of the course of decisions made so far. The combination of these two features eliminate fruitless nodes of the search tree.

In the next section a mathematical formulation is presented for the job-shop scheduling problem before we continue the discussion on more technical aspects of the branching and bounding features of branch-and-bound algorithms.

3.5.1.1 Mixed Integer Programming Formulation

With the aim of solving scheduling problems to optimality, these problems have been formulated so that mathematical programming techniques can be used. This is not different in the case of the job-shop scheduling problem.

One of the most common mathematical formulations for the job-shop scheduling problem is the MIP based upon that proposed by Manne [63], which is highlighted below. The symbol definitions are as described in Section 3.3.

Minimize $\phi(T)$, where $T = \{T_1, T_2, \ldots, T_{|J|}\}$ subject to:

release times	$d_{ij} \geq 0;\ i = \{1, 2, \ldots, \mid J \mid\};$	$j = \{1, 2, \ldots, \mid J_i \mid\}$
due times	$c_i > 0;\quad i = \{1, 2, \ldots, \mid J \mid\};$	
precedence constraints	$d_{i(j+1)} - d_{ij} \geq p_{ij}$	if $(o_{ij} \rightarrow o_{i(j+1)})$
disjunctive constraints	$\forall B_k \in B\ \forall o_{ij}, o_{i'j'} \in B_k$	
	$d_{ij} \geq d_{i'j'} + p_{i'j'} - K(1 - y_{d_{i'j'},d_{ij}});$	
	$d_{i'j'} \geq d_{ij} + p_{ij} - K y_{d_{i'j'},d_{ij}};$	

where K is some large constant, and $y_{d_{i'j'},d_{ij}} = 1$, if $o_{i'j'}$ is scheduled before o_{ij}, and 0 otherwise. The set B_k is a clique of pairs of disjunctive arcs of the B set. The objective function $\phi(T)$ can be defined as $\phi(T) = max\{T_i \mid T_i \in T\} = T_{max}$, for the minimization of the makespan, or defined as $\phi(T) = \sum_{T_i \in T} max(T_i - c_i, 0)$, when the aim is to minimize the total tardiness, for instance. Other objective functions can also be defined: the reader is referred to [71] for further details.

3.5.1.2 Upper Bound Strategies

The heuristics for yielding an initial solution are typically a tradeoff between the performance in finding it and the quality of the solution found. A good quality of an initial solution is desired because the closer this initial solution is to the optimal solution, the quicker one can get rid of a huge portion of the search space by posting a tighter constraint (bound) to the solution cost function of the problem.

Carlier and Pinson's algorithm [23] starts off the searching process by considering the best known makespan of the problem, from the literature, as its initial upper-bound. The problem is de-assembled into m one-machine problem instances, and thereafter, the *bottleneck* machine is completely scheduled first, following by the next unscheduled *bottleneck*, or *critical*, machine. The process goes on until there is no machine left to schedule. The bottleneck machine is typically that with the largest makespan given by a preemptive schedule.

After having scheduled all the machines, a second part of the process consists of rescheduling the machines over again. The process starts from that machine with the maximum makespan, and the sequences on the already scheduled machines are kept untouched. The rescheduling part of the process aims at finding another improved schedule configuration for the problem. Although in many cases the one-machine scheduling problem is also NP-complete, it can quickly be solved using the algorithm of Carlier [22].

Carlier's procedure [22] has been used to solve job-shop scheduling problems with some success. Nevertheless, this procedure does not take into account that the processing order of two operations on the same machine may already be fixed, and also does not take into account possible constraints between different operations on different machines, which is the case in this thesis.

During the process of fixing disjunctive arcs a set of specialized inference rules based on the *immediate selection of disjunctions* is also used in [23] and further improved in [24]. Applegate and Cook [4] improved many of the inference rules employed by Carlier and Pinson in [23] to determine if an unsequenced operation o_{ij} should be ordered before or after a fixed set of operations to be performed on the same machine. This method is known as *edge-finding*.

Applegate and Cook used the shifting bottleneck method in combination with the *edge-finding* scheduling algorithm [4]. Their strategy consists of scheduling all tasks on a set of machines according to the machines' criticality priority, one machine after another. However, for each of the remaining unscheduled machines their schedule is planned respecting the schedule on the previous scheduled machines.

3.5.1.3 Lower Bound Strategies

Calculating a lower-bound for a job-shop problem, together with an upper-bound, is an important part of the process of solving the problem itself. The better the lower-bound calculation is for a problem, the better is then the estimation of where the optimal solution might be. The lower-bound estimation is used in the course of search as a way of avoiding exploration of fruitless search paths.

Applegate and Cook [4] used a mathematical formulation of the job-shop scheduling problem, as described in Section 3.5.1.1, for obtaining lower-bounds on job-shop problems. Their approach consists of solving the relaxed linear problem, and progressively going on introducing inequalities (*cutting-planes*) which are valid for the problem formulation, but are not satisfied by the current optimal solution given by the linear programming relaxation. Therefore, at each optimal value of the linear programming problem, increasingly tight lower-bounds on the value of the schedule are gathered.

The approaches reported by Applegate and Cook [4] did not perform well. Even when combined with other techniques, the results indicate that the lower-bounds generated by their cutting-planes are not so good as those yielded by other techniques. Moreover, their technique is very time-consuming.

The most popular approach is that applied by Carlier and Pinson [23, 24] where the problem is relaxed into m one-machine problems, and then each problem solved separately. Because the one-machine problem is also NP-complete, a further relaxation which permits preemptions is adopted. In this case the one-machine problem can be solved in $O(n\ log\ n)$ time where n is the number of operations to be processed on the machine.

Brucker and Jurisch [17] present a rather different approach to calculate lower-bounds for the job-shop scheduling problem. Their method consists of considering the subproblems of dimensionality $2 \times m$ and $3 \times m$, for which an optimal solution can be found in polynomial time. The solution of either one of these subproblems gives the lower-bound for the main problem.

3.5.1.4 Branching Strategies

The seminal work is by Carlier and Pinson [23] for introducing the method of head and tail of an operation sharing the same resource. The head of an operation consists of the interval between the earliest and the latest start times the operation can assume without breaking any constraint posted to the system. The symmetrical idea is also applied to the tail, which is defined by the earliest and latest completion times a task may assume. Every time a decision is made on a resource these values are updated on that resource so that a great portion of the search space is pruned, as the possibility of scheduling tasks becomes steadily tighter. This idea is used later on by Caseau and Laburthe [25] to conceive the idea of *Task Intervals*, which are sets of tasks sharing the same resource represented by the *earliest* and *latest* members of the sets.

In addition to using the head and tail adjustment method, the *immediate selection* of disjunctions used in [23] gives a good method for choosing the disjunctive arc to be eliminated during the branching process in the branch-and-bound algorithm. An improvement on the *immediate selection* method is proposed in [24]. Using this more sophisticated version the authors [24] were able to greatly reduce the number of nodes in the search tree when compared to the previous results in [23]. Nonetheless, the sophistication introduced degraded the performance in terms of CPU time on solving the same set of previous problems in [23].

A similar branching technique is that used by Applegate and Cook [4]. Their technique consists of calculating for each possible disjunctive schedule on a machine its lower-bound using their *edge-finding* algorithm (see Section 3.5.1.2).

A rather different way of doing the branching is that used by Brucker [16, 18]. His method is based on swapping activities on the critical path. This technique has also been used with success by authors who apply meta-heuristic algorithms [7, 68] to solve the job-shop scheduling problem, as described in the next section.

3.5.2 Heuristic Methods

The complete methods are very time-consuming. A good solution (sometimes even any solution) can in certain cases suit a user who simply wants to improve on a manually constructed solution. In view of this assumption, many incomplete methods have become more and more popular in the optimization field. Their recent popularity is mainly due to their capability of producing good solutions (sometimes even optimal solutions) in a reasonable amount of time for a variety of combinatorial problems.

However, unlike complete methods, incomplete methods do not give any guarantee of reaching an optimal solution and cannot prove optimality, although many of them have been shown to give optimal or near-optimal solutions for particular instances of problems. In general, incomplete methods suffer from getting easily stuck in local minima: solutions which trap the algorithm in a cycle seeking for an improved solution without finding any.

An incomplete method can be divided into two stages. The first stage is the *local search*, where a given solution is *perturbed* and then transformed into another solution, hopefully feasible. To this end, the most popular manner of causing a perturbation consists of swapping activities on the critical path [7, 13, 89] of a schedule.

The second stage is the process of how to get free from local minima. In this respect, a *meta-heuristic* controls the local search when it gets stuck in a local minimum solution so as to set it free from that local optimal. A good strategy of getting way from local minima can take the search to new search areas where it is likely to continue finding improved solutions.

The sections which follow, give a brief overview of the characteristics of the *approximation* algorithms most successfully used in the literature to solve the job-shop scheduling problem.

3.5.2.1 Neighbourhood Structure

Considering a classical job-shop problem, the solution space S is of size $(n!)^m$. Of course, among these solutions there are many infeasible solutions when considering a particular problem with its specific constraints. Therefore let us denote by $F \subseteq S$ the actual feasible solution space.

A feasible schedule $x \in F$, is a problem solution where all the conjunctive arcs are fixed, and each disjunctive pair of arcs, in the disjunctive graph model, is already reduced to only one arc. Furthermore, all the constraints posted to the particular problem are satisfied.

We can then define two functions over the set S: one function to gauge the quality of a given solution, and another to map the neighbour(s) of a solution. The cost function ϕ is a mapping $\phi : S \to \mathbb{R}$. ϕ is the function to be optimized (see also Section 3.5.1.1).

The neighbourhood function Nb, in its turn, defines for each solution $x \in S$ a neighbouring set of solutions such that $Nb(x) \subseteq S$. Each solution in $Nb(x)$ is called

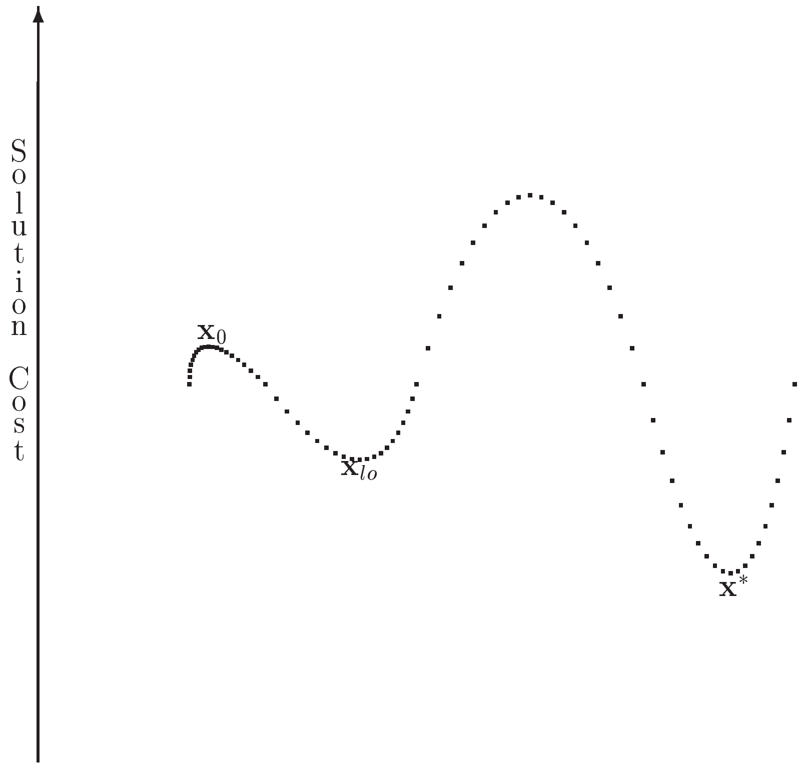


Figure 3.3: The tradeoff of sizing a neighbourhood for the search.

a neighbour of x, and the action of actually going from one solution to its neighbour is called a *move*.

In the light of above definitions, a local search procedure can be seen as a *walk* in S. In most of the cases, one is interested in a sort of walk which jumps from one feasible solution to another feasible solution. Hence, to this end, the Nb function must be devised so that given a schedule $x, Nb(x) \subseteq F$, for any defined *move*.

The choice of a good neighbourhood design is usually a difficult decision to make. This is because, on one hand, the bigger a neighbourhood is the more feasible solutions it can encompass and, therefore, the likelihood of finding a good neighbour in it is high. On the other hand, searching for good solutions in a larger neighbourhood may take more time. In Figure 3.3 the curved line represents feasible solutions for a hypothetical optimization problem. To show the effect the size of a neighbourhood may have on the number of feasible solutions in it, we suggest a small imaginary neighbourhood which includes the solution $\mathbf{x}_0$, but not $\mathbf{x}_{lo}$. Now consider a larger also imaginary neighbourhood which includes both solutions, however it is not large enough to include the solution $\mathbf{x}^*$. This way, we are increasing the size of the neighbourhood, which can also increase the time to reach solution $\mathbf{x}_{lo}$ from $\mathbf{x}_o$, as there are now more solutions that might be visited.

Consider a solution in Figure 3.4 for the example given in Table 3.1. This solution has a near-optimal makespan of 38 (the minimum makespan is 33). However, if we are considering the total tardiness, the result is 19. Initially, a neighbouring solution is one which can be obtained by perturbing the solution shown in Figure 3.4 in some way.

The job-shop literature over these years has enumerated at least six classical ways of constructing neighbourhood structures to obtain new solutions from an existing one. They are as follows.

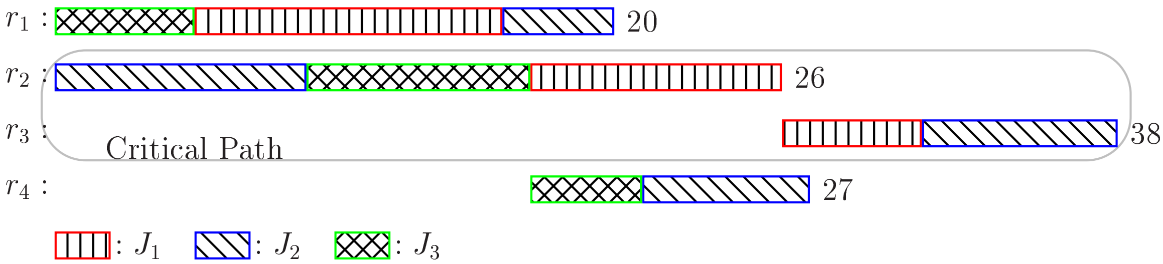


Figure 3.4: A solution of makespan of 38 for the problem depicted in Table 3.1.

Nb_1: The solutions in this set are constructed by swapping, in the disjunctive graph representation of the current solution, any arc $(o_{ij} \rightarrow o_{i'j'})$ on a critical path by its opposite arc $(o_{i'j'} \rightarrow o_{ij})$. The size of this neighbourhood depends on the number of critical paths in a given schedule and, on the number of operations on each critical path. It can thereby be quite large.

Figure 3.5 shows an example of a possible neighbour for the Nb_1 neighbourhood structure of the solution presented in Figure 3.4. This neighbour is obtained by swapping the last two operations o_{13} and o_{24} of the last block of adjacent operations on the critical path. Note that this neighbour does not improve the current solution.

Nb_2: This neighbourhood is more restrictive than the previous one. Some acceptable moves, and therefore neighbours, in the first neighbourhood will not be acceptable here. This will thus reduce the size of the neighbourhood in comparison to the first one.

Consider that $mp(o_{ij})$ is o_{ij}'s predecessor operation, and $ms(o_{ij})$ its successor, provided they exist, on the machine o_{ij} is to be performed on. This neighbourhood structure restricts the possible choice of arc $(o_{ij} \rightarrow o_{i'j'})$ to those

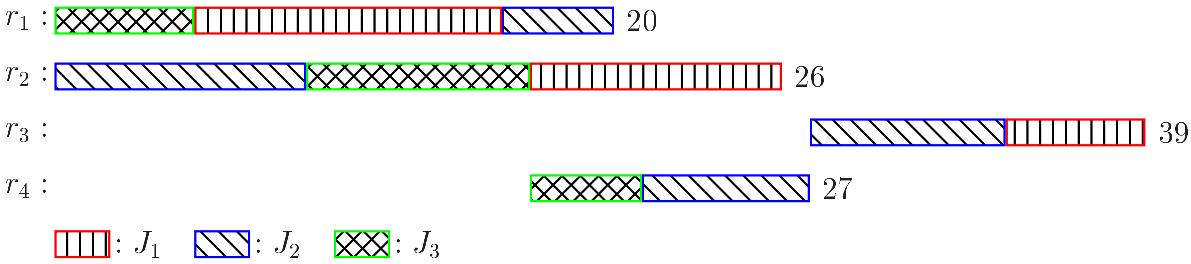


Figure 3.5: A neighbour solution of makespan of 39 for the problem in Table 3.1.

vertices such that at least one of the arcs $(mp(o_{ij}) \to o_{ij})$ or $(o_{i'j'} \to ms(o_{i'j'}))$ is not on a longest path, *i.e.* operations o_{ij} and $o_{i'j'}$ are at one of the edges of the block of adjacent activities.

The neighbour example given in Figure 3.5 fulfills the criteria described above, because operations o_{13} and o_{24} have not a $mp(o_{13})$, nor a $ms(o_{24})$, respectively. However, in the situation where there are more than three activities, say o_A, o_B, o_C, o_D, in the block which belongs to a critical path, only o_A can be swapped with o_B and o_C with o_D. In Nb_1, o_B could be swapped with o_C.

Nb_3: This neighbourhood structure considers all permutations of the three operations $\{mp(o_{ij}), o_{ij}, o_{i'j'}\}$ and $\{o_{ij}, o_{i'j'}, ms(o_{i'j'})\}$ in which $(o_{ij} \to o_{i'j'})$ is inverted and the resulting orientation is feasible.

Nb_4: This neighbourhood structure consists of moving an operation o_{ij} in a block of adjacent operations to the very beginning or to the very end of this block.

Nb_5: In this structure a smaller sized neighbourhood is considered by restricting Nb_1 (or Nb_4) to reversals on the border of a block. Furthermore, only one critical path arbitrarily chosen is considered.

In other words, a move is defined by interchanging two adjacent operations o_{ij} and $o_{i'j'}$, where o_{ij} or $o_{i'j'}$ is the first or last operation in a block that belongs to a critical path. The exception is made to the first and the last blocks. For the first block, only the last two operations, and in the last block of the critical path only the first two operations are interchanged.

Nb_6: Consider that $jp(o_{ij})$ is the o_{ij}'s job predecessor operation (*i.e.* $o_{i(j-1)}$), and $js(o_{ij})$ is its job successor, provided they exist. This neighbourhood structure considers the reversal of more than one disjunctive arc at a time, against only one arc per turn, as in the previous structures. Therefore, considering o_{ij} is performed before $o_{i'j'}$, a neighbour is obtained by at least an interchange of these two operations of a block on a critical path, o_{ij} and $o_{i'j'}$ not necessarily adjacent. Moreover, either $jp(o_{ij})$, or $js(o_{i'j'})$, must also belong to the critical path, and so that there is not directed path in the current critical path connecting the $js(o_{ij})$ to $o_{i'j'}$, or directed path linking o_{ij} to $jp(o_{i'j'})$. This neighbourhood was used by Balas and Vazacopoulos in [7] together with a proposed Guided Local Search. Due to the huge number of neighbours it is possible to generate from this neighbourhood, the authors implemented a limited depth tree within the algorithm to search for an improved neighbour.

One of the criteria presented in Balas and Vazacopoulos [7] for generating moves consists of taking two tasks on the critical path such that $(o_{ij}, o_{i'j'}, js(o_{i'j'}))$ also belong to the critical path, o_{ij} and $o_{i'j'}$ not necessarily adjacent. Consider the critical path of the solution in Figure 3.4. The critical operations (o_{32}, o_{12}) on machine r_2 and operation $js(o_{12}) = o_{13}$ on machine r_3 fulfill this criteria. According to authors in [7], operations (o_{32}, o_{12}) are entitled to a *forward interchange*, which consists of moving o_{32} right after o_{12} on that machine. The execution of the *forward interchange* move on these operations results into the optimal solution for the problem presented in Table 3.1. The solution in Figure 3.6 has the optimal makespan of 33, however, if we consider the total tardiness the result is 21, worse than we had in Figure 3.4.

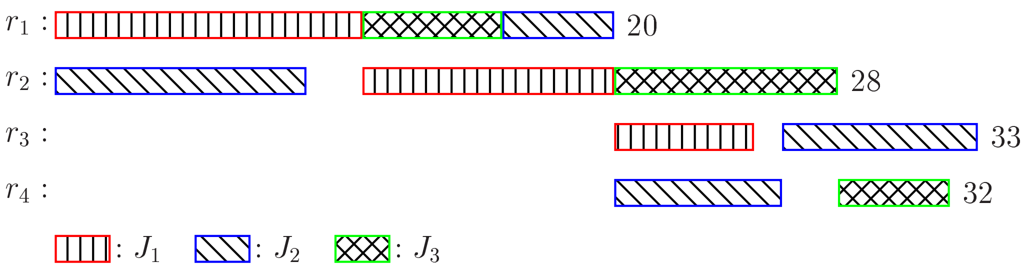


Figure 3.6: The optimal solution of makespan of 33 for the problem in Table 3.1.

3.5.2.2 Meta-Heuristics

The simplest algorithm to find a local minimum is called *iterative improvement*. From an initial solution, which can be either gathered by another algorithm, or randomly; the *iterative improvement* algorithm walks through feasible solutions by making *moves* which typically consist of swapping the order of two activities. At each move this algorithm tries to find a solution of lower cost than the current solution.

If such solution is found, the algorithm continues its search for better solutions, otherwise a local minimum cost solution has been found, and the algorithm stops.

The quality of a local minimum solution can be poor if compared to the actual optimal (see Figure 3.3). Hence many general heuristics have been proposed to control a local search heuristic so as to circumvent the problem of getting stuck in a locally optimal solution.

The most popular meta-heuristics in the literature for job-shop scheduling problem are Simulated Annealing, Genetic Algorithms and Tabu Search. Many others have also been used for this problem but have not got as good results as these three.

Simulated Annealing algorithm was for while the most effective meta-heuristic for the job-shop scheduling problem, as far as good solutions are concerned [59]. The simulated annealing algorithm works by building up solutions, from a given initial solution, as in the simple *iterative improvement* algorithm. However, the sequence of solutions does not roll monotonically down towards a local optimum.

First, a neighbour of the current solution $\mathbf{x}_0$ is chosen (usually, but not necessarily, at random). Then an improved move is always accepted. However, a non-improved move is either accepted or rejected, depending on whether $exp^{-\Delta/T} < P$. The amplitude $\Delta = f(\mathbf{x}) - f(\mathbf{x}_0)$, represents the gain of a move from solution $\mathbf{x}_0$ to $\mathbf{x}$, and $P \in [0, 1]$ is a uniform random number. T is a control parameter initially high, and is slowly reduced during the search process.

The process of acceptance is therefore nondeterministic. The probability of acceptance is planned in a certain way so that escaping from local optima is relatively easy during the first iterations. This way the procedure is able to explore the neighbouring space more freely. Nonetheless, as the iteration count increases, the acceptance criterion becomes steadily more restrictive, only improving transitions tend to be accepted, and the solution path is likely to terminate in a local optimum, hopefully the global optimum.

The algorithm stops either when a solution which is close enough to the given lower-bound is reached, or the defined maximum number of iterations is surpassed, or the time limit runs out.

Laarhoven *et al.* [59] used the Nb_1 neighbourhood structure described above in their simulated annealing procedure. Their results were comparable to a more tailored shifting bottleneck strategy proposed by Adams *et al.* [1] when the time to reach the best solution is disregarded. The authors claim the compensations for the disadvantage of large running times are the simplicity of the algorithm, its ease of implementation, the fact that it requires no deep insight into the combinatorial structure of the problem, and the high quality of the solutions it returns [59].

Tabu Search has recently been shown to be the fastest, and yet one of the most effective meta-heuristics not only for the job-shop scheduling problem, but also for many other combinatorial optimization problems [13, 38].

In tabu search a set of different *move* procedures can be defined to search different neighbourhoods in the course of the search. Furthermore, each time a move is made which improves the solution its inverted move is saved onto a list of *forbidden* moves. This *tabu list* is basically to avoid the search turning back to the solutions already visited during a defined number of previous steps.

The sort of memory described above is the *short-term memory* described by Glover [38], which keeps the very recent history of the search. However, Glover also described a *long-term* memory for distant history in his book.

It may happen during the search that an improved move is on the tabu list. Therefore, in order to perform such an improving move, an *aspiration function* is defined. This function evaluates the gain in taking a forbidden move. If it is accepted by the function, the tabu status of the move is disregarded and the move is performed.

The adaptive memory feature of tabu search allows the implementation of procedures that are capable of searching the solution space more economically and effectively.

The algorithm stops either when a solution which is close enough to the given lower-bound is reached, or the defined maximum number of iterations is surpassed, or the time limit runs out.

The tabu search implementation of Dell'Amico and Trubian [29] using the Nb_4 neighbourhood structure was the best heuristic method to solve the job-shop scheduling problem [13]. However, more recently Nowicki and Smutnick [68] presented a new implementation including some sophisticated features to their tabu search which put their implementation as the most efficient and effective one [13]. Nowicki and Smutnick [68] used a more restrictive neighbourhood structure, Nb_5, rather than Nb_4 as Dell'Amico and Trubian did.

In addition to using the basic short-term memory, Nowicki and Smutnick implemented a mechanism to get advantage of the previous successful runs. Hence, their algorithm employs a long-term memory list of the best solutions and associated it to the ordinary tabu list during the search. This way, whenever the classical tabu search has finished, the algorithm goes to the most recent entry, in other words, the algorithm *backtracks* to the most recent best solution in the long-term list, and restarts the classical tabu search. Whenever a new best solution is found the list of best solutions is updated.

The least successful meta-heuristic for solving the job-shop scheduling problem has been shown to be the Genetic Algorithm [13, 89].

A genetic algorithm is a search procedure based on the mechanics of natural selection and natural genetics. The basic structure of a genetic algorithm consists of: a set of *chromosomes*, a *mutation* function, and a *crossover* operator.

A chromosome in a genetic algorithm context is an encoding representation of a solution. The aim of a genetic algorithm is to produce near-optimal solutions by letting a population of random solutions – the *chromosomes*, be manipulated by a sequence of unary and binary transformations – mutation and crossover, respectively. The whole process is governed by a biased *selection* scheme which leads the set of solutions towards high-quality solutions.

The objective of the crossover operator is to implicitly combine good properties of two different chromosomes chosen by the selection operator, and therefore hopefully transfer these properties to their offsprings. Whereas the mutation is to push the search process towards a more wide search area by randomly slightly changing a chromosome encoding so as to increase the chances of finding good solutions.

The process of performing the crossover, the mutation and selecting the survivor elements of the population is then repeated until the stopping criteria are met [39].

The mapping of a job-shop problem schedule into the chromosome structure suitable for the crossover and mutation operators is the most cumbersome feature of many genetic algorithm implementations.

Many chromosome-configuration structures have been proposed to circumvent the mapping problem, from the classical binary representation [67] to the string of substrings [28] of a machine's schedule. In this later representation, each substring in a chromosome represents a schedule on one machine. This representation eases the design of crossover and mutation operators for the problem.

3.5.3 Aggregating Special Constraints

The literature is very rich in work carried out on the classical formulation of the job-shop scheduling problem. Nevertheless very few works can be traced in the literature about formulations of the problem when some special restrictions are to be considered.

For instance, one of the points of interest of this thesis is in jobs whose tasks cannot wait to start being processed once their predecessors are completed. This restriction implies that a job must be processed from the start to completion, without any interruption either on or between machines. This sort of problem is referred to as the *no-wait* job-shop scheduling problem in the literature and very few of such problem classes are known to be polynomially solvable [56]. Furthermore, in general, the works presented in the literature are typically restricted to special cases with a small number of machines and jobs (usually $m \leq 3$), and processing time limited to a unit ($p_i = 1, i = \{1, 2, \ldots, \mid J \mid\}$) [41].

Another case of interest to us is when jobs have setup times (or setup costs) between activities. This situation occurs when the elapsed time to start processing a task on a machine depends on the previous task performed on the machine. Many authors assume setup as negligible, or as part of the processing time [2]. Moreover, Allahverdi *et al.* [2] point out that the single-machine problem is the class of problems which have received the most attention in the setup literature due to their relative simplicity, whereas for multi-stage shops, there are only a couple of papers on a very restrictive configuration of the problem.

Although the classical definition for a job-shop problem does not consider the existence of constraints between tasks of different jobs, we are also interested in this sort of configuration. As Beck *et al.* [10] pointed out, the 13 Allen relations [3] (*e.g.* meets, during, overlaps) have been recognized for many years, however they have been little used in scheduling research.

An application for this configuration of the problem can be illustrated by the case where pre-assignments (some relations between tasks, whether of the same or different jobs) on the machines are considered. For instance, two activities may be linked so that when one is performed, the execution of the other is restricted in time. This problem, although different, resembles that in which a set of tasks, fixed in time, are given. The objective is then to schedule the remaining tasks in such a way that they do not overlap with the pre-linked or fixed tasks.

3.6 Summary

This chapter gives a brief overview of some of the important points in the job-shop scheduling problem literature which serve as elements and terminologies for the forthcoming chapters.

The disjunctive graph model is presented in Section 3.3. This model is frequently used in the scheduling field as a way to represent instances of the job-shop scheduling problem, and in addition, as a tool to handle feasible solutions for the problem [6, 14].

We highlighted a couple of non-conventional constraints which are not usually tackled in the literature of job-shop scheduling problem. This might be because the definition of the problem itself does not consider them. Nevertheless, their importance in practical applications is greatly valuable.

Some of the most successful techniques in the literature for solving the job-shop scheduling problem both by complete algorithms and also by using some of the popular meta-heuristics, such as Genetic Algorithms, Simulated Annealing and Tabu Search, are presented. It is important to stress the fact that most of the work in the literature on job-shop scheduling problems are devoted to the optimization of the makespan [10, 14], with a few notable exceptions [57, 81, 82], despite the importance different objective functions may have in real life.

Chapter 4

Constraint Programming

4.1 Introduction

A way to represent and describe a problem clearly and easily has been a subject of intense research for many years. The result of this research effort is that many problems in the real-world can be represented by *constraints*, and the *satisfaction* of these constraints gives solutions for the respectively represented problems.

In the computer science area, the Constraint Satisfaction Problem (CSP) paradigm can be traced back to research in Artificial Intelligence and Computer Graphics in the sixties and seventies which focused on explicitly representing and manipulating constraints in computational systems [90].

However, the ease of representing a problem using constraints did not bring along the same ease level of resolving the problems. This is because CSPs are combinatorial in nature, and moreover most problems of interest are NP-complete, therefore an efficient algorithm is unlikely to exist [37].

Until recently, the implementation of algorithms for solving CSPs in software tools, Constraint Programming (CP) languages, were not powerful enough to call the attention of the practitioners. The paradigm was thus confined within academic circles. However, this has been changed since the last decade. The CP paradigm has been attaining more support not only from the purely research circles but also from the commercial applications sector. This growth has been boosted by the existence of tools such as: Oz, CHIP, ILOG Solver, and ILOG Scheduler, which have provided practitioners with means of devising successful large real-world applications. Some of these applications are cited in this thesis.

The success of these CP tools is due to a huge effort in research over the last decades. The fruitful results of these years of research are now incorporated within these tools as inference mechanisms for transforming a problem into another equivalent, but simpler problem to be solved.

This development has greatly improved the CP tools on solving a variety of classes of real-world problems. Furthermore, CP tools provide the user with a very expressive framework for dealing with constraints as they can be either linear, non-linear, or both. Some other tools such as those for Mixed Integer Linear Programming (MIP), for instance, restrict the user to use only linear constraints.

The following sections give an overview of the main aspects and techniques existent in the CP area. In addition, our intention is to present some characteristics of CP which make this paradigm rather different from the other techniques, for instance MIP in Operational Research.

In Section 4.2 we formally introduce the way of representing a problem as a CSP. The worked example aims to show the simplicity of the representation of a problem as a CSP. Next, in Section 4.3.1, some of the inference algorithms built within CP tools are discussed. These algorithms work over the variable domains, trying to infer infeasible values so as to eliminate them from the domain. This way, the search space of the problem can be reduced, as there will be fewer combinations of values to try out for each variable.

In addition to a process of transforming a given problem into another problem with its variable domains reduced (discussed in Section 4.3.1), CP tools provide a mechanism to search for a solution to a problem. The basic mechanisms often existing in a CP tool are discussed in Section 4.4.

Given a CSP representation of a problem, we might want to find a solution for the system of constraints, or to find all possible solutions, or to find the best (optimal) solution, when criteria are given to measure the quality of solutions.

In order to tackle optimization of CSPs some additional techniques are used. A way to find *the best* solution among all possible solutions of a CSP is also discussed in Section 4.4.

Many of the modern programming languages incorporate Object Orientation (OO), and this can be taken into the CP languages. Thereby, the sort of object structures that CP can manipulate range from simply numerical variables to an endless number of complex structures. In Section 4.5 we describe a particular *language* extension built on top of ILOG Solver to deal with general scheduling problems. Although using a scheduling language framework as an example, we also show that the possibility of extending a CP language to a particular class of problem is limitless.

In Section 4.6, some of the up to date CP tools are briefly described. Many commercial applications using CP as a tool to tackle them have emerged and been presented in the literature. Section 4.7 briefly discusses some of these applications of CP. Finally, in Section 4.8, the summing up of the chapter is given.

4.2 Defining a problem as a CSP

In the CP paradigm the emphasis lies in the manipulation of the variable domains and the relations among these variables – the constraints.

A CSP can formally be defined as a triple: (V, D, C), where V is the finite set of variables $\{v_1, v_2, \ldots, v_n\}$ to represent the problem. D is a function which maps every variable in V to a set of objects of arbitary type. Hence, $D(v_i) = D_i$ is the domain of variable v_i. In this thesis we will only consider variables with integer finite domains, although a CSP can also be defined with other type of values for the variable domains.

C is the set of constraints which restrict the values that the variables can simultaneously take. A constraint $c_i \in C$, between variables $v_i, v_j, v_k, \ldots$, is any subset of possible combinations of values of these variables, in other words $c_i(v_i, v_j, v_k, \ldots) \subseteq D_i \times D_j \times D_k \times \ldots$ [15]. However, the constraints of real problems are not represented this way in practice. This *extensional* way of representing a constraint may be, if not impractical, so computer memory-consuming.

Another way of representing constraints is *intensionally*. When constraints are expressed as a mathematical relationship the set of allowable n-tuples is implicitly defined.

In order to illustrate the above definitions the following is an example of a constraint problem.

$$\begin{cases} 4x + y \leq 20, \\ x, y = 1, 2, 3 \text{ or } 4 \end{cases} \tag{4.1}$$

where x, $y \in V$ are the variables of the represented problem, and their integer domain are as follow: $D_x = [1..4]$ and $D_y = [1..4]$, for instance.

For the sake of illustration, consider the extensional representation of the above constraint. The set {(1, 1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2), (2, 3), (2, 4), (3, 1), (3, 2), (3, 3), (3, 4), (4, 1), (4, 2), (4, 3), (4, 4)} contains the valid value combinations to satisfy the constraint in Problem (4.1).

In this example a very simple arithmetic constraint has been used as an illustration. However, in CP a constraint is often specified as an object entity of an OO programming language and, therefore, in theory any relation can be presented.

The fact that in a CSP the variable domains are finite means that, it is possible to calculate the number of possible assignments of values to variables and, thus, the maximum size of the search space. Hence, the potential size of the search space for a general CSP is given by

$$\mid D_1 \mid \times \mid D_2 \mid \times \ldots \times \mid D_n \mid = \prod_{i=1}^{n} \mid D_i \mid .$$

This number is probably the most commonly used criterion for measuring the size of a CSP. However, note that the actual number of solutions is usually much smaller than the given total above, as many values cannot be assigned to a variable due to the constraints posted in the problem.

In addition, CP paradigm uses *constraint propagation* through the variable domains to eliminate inconsistent values as the search evolves.

A more general way of thinking of constraint propagation over problem variables is of the transformation of a problem into another equivalent one in which the domains of variables are decreased, either by eliminating the inconsistent values, or by adding new inferred constraints.

In the next section the transformation process of a problem into another one with decreased domains is discussed, and some of the general algorithms for propagating the constraints are commented.

4.3 Consistency Methods

4.3.1 Arc Consistencies

A constraint works as a link between variables so that the involved variable domains are bound to obey whatever the constraint designates. For instance, if $D_x = D_y = \{1, 2, 3, 4\}$, a simple constraint $x + 1 = y$ on the variables x and y is represented in Figure 4.1. c_{xy} may be represented as the set {(1, 2), (2, 3), (3, 4)}, which are the valid possible combinations of values of x and y.

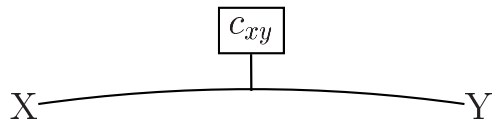


Figure 4.1: A constraint on variables.

Constraints can be classified according to their arity. *Unary* constraints are those which affect only one variable. An example would be to impose a constraint on the variable y to be, for instance, an *odd* integer. In this case, *node* consistency would eliminate every even value from the y's domain, so that $\forall b \in y, c_y \equiv odd$ is satisfied.

A *binary* constraint acts on two variables only. This is as represented in Figure 4.1. A more generic form of constraints is those which act on more than two variables simultaneously. Hence, n-ary constraints are those which act on n variables.

Binary CSPs can be represented by a graph. In doing so, variables are represented by the nodes of the graph, and there will exist an edge joining two variables if and only if there is a constraint between these two variables.

Therefore, the directed *arc* between x and y, $arc(x, y)$, is said to be *arc consistent* if for every value $a \in D_x$, there is a value $b \in D_y$ such that the assignments $x = a$ and $y = b$ satisfy the constraint c_{xy}.

Note that in the $x + 1 = y$ constraint, value 4 in x cannot satisfy the constraint, because no value in D_y exists such that $4 + 1 = y$. Hence, any value $a \in D_x$ for which c_{xy} cannot be satisfied can safely be removed from D_x. This is because such a value will not be part of any consistent solution.

Similar to the above process of making $arc(x, y)$ consistent, we could also eliminate inconsistent values from y's domain by making $arc(y, x)$ consistent.

In the literature there have been many algorithms proposed for making a binary CSP arc-consistent. The achievable worst-case time complexity for general constraints is $O(d^2c)$ [65]. d is the maximum domain size and c the number of binary constraints in the problem.

These algorithms are denominated as *A*rc *C*onsistency-n (AC) algorithms, starting from AC-1 up to AC-7 so far. These algorithms are thus for binary constraints.

One of the most influential among them is AC-4 [65], which introduces the idea that values of a variable domain may *support* a value within another variable's domain. A value $b \in D_y$ supports a value $a \in D_x$ if and only if $c_{xy}(a, b)$ is satisfied.

Consider the binary constraint $x + 1 = y$, presented above, between the variables x and y, as in the example given in Section 4.3.1. In this case, 2 is supporting value 1 in D_x, 3 is supporting value 2, and so on. But there is no value supporting the value 4 in D_x. This is the reason why value 4 can be removed from the domain of x. The same reasoning can be applied to the symmetrical case of the directed $arc(y, x)$.

For each value within a variable's domain, AC-4 compiles and maintains a list of supporting values from the domain of the other variable. Hence, if there are no supporting values or if all the supporting values are removed from the domains of their variables, the supported value should also be removed from the domain of its variable. The AC-4 algorithm runs in $O(d^2c)$ in the worst case [30, 65], and has an identical space complexity of $O(d^2c)$.

In Hentenryck *et al.* [30], AC-5 is proposed. This algorithm runs in $O(d^2c)$ in the worst case, but allows a specialized arc-consistency algorithm of time complexity of $O(dc)$ for *functional*, *anti-functional* and *monotonic* classes of constraints. Furthermore, this algorithm provides the user with an *interface* for applying consistency checks differently for different type of constraints.

Through this interface, the algorithm provides access to the difference $\Delta(x)$ between the current domain of a finite domain variable x, and the domain this variable had, just after the previous activation of the constraint [76].

The AC-5 algorithm is implemented within ILOG Solver [74, 76], the CP tool used throughout this thesis. A description of how the AC-5 algorithm allows the user to implement their own propagation method by using the AC-5 interface is described in Section 4.5.

The AC-6 algorithm [11] is an improvement on the AC-4 algorithm. AC-6 maintains the $O(d^2c)$ worst case complexity, however, its space complexity is reduced to $O(dc)$. This is accomplished by keeping only one supporting value at a time for each value to be supported, for each constraint. If this supporting value is removed, another value is sought for replacing the previous one. In case there is no other supporting value left, the supported value can then be also removed.

The AC-Inference (AC-7) algorithm [12] improves on the idea of *supporting* value applied in AC-4 and AC-6. The AC-7 algorithm uses a meta-knowledge that *support* is bidirectional. Therefore, the two-pass organization of AC-4 and AC-6 is saved. The idea is that a value $a \in D_x$ which is supported by a value $b \in D_y$ also supports value b.

Arc-consistency algorithms [11, 30] for general binary constraints are cheap and effective on pruning values for binary constraints. However, algorithms for general n-ary constraints are still very time-consuming. The General Arc Consistency (GAC-4) algorithm [66] is based on the same idea of supporting values as AC-4. Its worse-case time complexity is $O(d^r c)$, where r is the maximal arity of the constraints.

For some particular types of non-binary constraints faster and more specialized algorithms have also been proposed: for instance, for the *all-different* [75, 77] constraint. These algorithms take advantage of the semantic of the constraint so that more pruning can be carried out when compared with the binary representation of the constraint.

Consistency techniques have been well applied to general CSPs. However, even a simple consistency technique such as arc-consistency may turn out to be very time-consuming to carry out on problems where n-ary constraints exist [62]. An alternative to the n-ary constraints is, in general, *bounds propagation* which is a technique used for solving *numeric* CSPs. In the next section, we describe this technique in more detail.

4.3.2 Bounds Consistency

Unfortunately, maintaining complete arc-consistency can be very time-consuming for all but binary CSPs. Hence, an alternative technique can also be used: *Bounds consistency.*

Bounds consistency can be formally defined as follows. Consider a constraint c involving variables v_1, v_2, ..., v_n. $c(v_1, v_2, \ldots, v_n)$ is bound consistent if and only if for each variable v_i: $\forall a_i \in \{min(D_i), max(D_i)\}$, $\forall j \neq i, \exists a_j \in [min(D_j), max(D_j)]$, $c(a_1, a_2, \ldots, a_n)$ is satisfied. Extending the notion, a CSP is said to be bounds consistent if each constraint $c_i \in C$ is bounds consistent.

In words, what bounds consistency (also named *Interval propagation*, or arc-B-consistency [62]) does is to propagate only the bounds of each variable's domain through the other primitive constraints. If any of the bounds has no support value through the other variable domains, it must be removed. The time complexity to carry out this type of consistency check is $O(dc)$, thus as good as the AC-5 algorithm for some special classes of binary constraints.

Moreover, a very interesting result is that bounds consistency can, in some cases, cause as much pruning as the arc-consistency method [62]. The reason is that, informally, bounds consistency can be seen as an approximation to arc-consistency. The difference between them is that arc-consistency can prune some values within a domain which are locally inconsistent, whereas bounds consistency would not, if the bounds are already consistent.

In tools such as ILOG Solver, arc-consistency is applied to the binary constraints, but for general n-ary constraints bounds consistency is the consistency check used.

4.3.3 Further Consistency Methods

In the previous consistency method cases both node, arc, and bounds consistency make consistency checks on each constraint in turn. However, the idea of maintaining a constraint arc-consistent can be extended to involve more than one constraint by turns.

Consider a *path* (x,w,y), where $x,w,y \in V$ and there are binary constraints C_{xw} and C_{wy} between (x,w), and (w,y), respectively and a (possibly empty) binary constraint C_{xy}. The path is *path consistent* if and only if for every pair of values $a \in D_x$ and $b \in D_y$ which satisfies the constraint C_{xy} there is a value $c \in D_w$ such that $(a,c) \in C_{xw}$ and $(c,b) \in C_{wy}$.

In Path-Consistency (PC) algorithms, constraints are represented extensionally as sets of compatible values. Thus, those compatible values which do not have support from the adjacent variable w in the path should be removed from the constraint C_{xy}.

An algorithm to check all triples of variables where there is a pair of binary constraints to ensure path-consistency is very time-consuming. The best algorithm has worst case time complexity of $O(d^3c^3)$. This is the reason why such algorithms are not usually applied in practice.

Rather than considering only sets of binary constraints in path-consistency, k-consistency is an even more general concept of consistency check for considering any arity of constraints. If a CSP has n variables, then k-consistency is defined for $k \leq n$. A CSP is 1-consistent if and only if $\forall a \in D_{v_i}, \forall v_i \in V$, $v_i = a$ satisfies the unary constraint on the subject variable. This is thus identical to node-consistency.

Extending the concept, a CSP is k-consistent for $k > 1$ if and only if for every set of k-1 variables v_1, $v_2, \ldots, v_{k-1}$, $\forall a_1 \in D_{v_1}, \forall a_2 \in D_{v_2}, \ldots \forall a_{k-1} \in D_{v_{k-1}}$, such that $v_1 = a_1, v_2 = a_2, ...v_{k-1} = a_{k-1}$ satisfies the relevant constraints on these variables, and for any other variable v_k, there exists a value $a_k \in D_{v_k}$ such that the assignment can be extended to $v_k = a_k$ and all relevant constraints on the k

variables are satisfied [88]. Note that the particular case of 2-consistency is the same as the definition for arc-consistency.

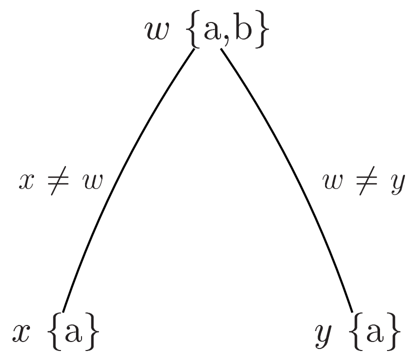


Figure 4.2: k-consistency does not imply $(k-1)$-consistency.

The weakness in the k-consistency definition is that a CSP can be k-consistent without necessarily being $(k-1)$-consistent. For instance, the problem in Figure 4.2 is 3-consistent as assigning b to w, a to x and a to y satisfy all the three constraints: c_w, c_{xw} and c_{wy}. However, the problem is not 2-consistent, because a should be removed from the domain of w to make c_{xw} and c_{wy} arc-consistent.

The above weakness in k-consistency definition led Freuder [35] to introduce the concept of *strong k-consistency*. According to this new concept, a CSP is *strong k-consistent* if it is 1-, 2-, ..., $(k-1)$- and k-consistent. In order to have the problem in Figure 4.2 strong 3-consistent, value a should be removed from the domain of variable w.

Achieving strong k-consistency for a CSP of k variables would yield therefore the complete solutions for the problem, eliminating this way the need for searching. Nonetheless, like in path-consistency, achieving k-consistency is so time-consuming that is not used in practice.

The next section describes the basic strategies present within a CP tool, such as ILOG Solver, to search for solutions.

4.4 Search Strategies

Many methods have recently been proposed in the literature for yielding a systematic complete search [43, 72, 92]. Nevertheless, all these algorithms are a variation of the naive *chronological backtrack* (BT) [40].

The basic BT strategy picks arbitrarily a variable to instantiate, and within this variable's domain, chooses a value to assign to it. This assignment is checked against the previous assignments for compatibility. A new different assignment is tried, should the current assignment be incompatible, and the current value is temporarily removed from the domain of the current variable. In case there is no other value to choose from within the current variable's domain, the algorithm *backtracks* to the previously assigned variable and a new value is chosen and tried for it. When a backtrack is needed, it is called a failure or a fail.

This sort of algorithm implicitly considers all the possible assignments to a given problem. The search terminates in either of two situations: one is if a solution is found, and another is when there is nowhere to backtrack to after a fail. This latter case means that there is no feasible solution for the problem. These termination characteristics make the BT algorithm a complete search. Therefore, disregarding the time, either a solution can always be found by the algorithm, or the algorithm will prove there is none.

A pure BT algorithm does not take advantage of any constraint propagation. Therefore, an improvement on this procedure is the *Forward Checking* (FC) algorithm [42].

A FC algorithm is also based on progressively instantiating each variable. However, every time an assignment is made, the values inconsistent with the previous assignments are removed from the domains of the remaining unassigned variables. The algorithm fails and backtracks if any variable's domain is wiped out. Note that, in doing the forward check, there is no need for checking the previous assignments' compatibility, as the incompatibility has already been forward removed.

However, this forward checking does not avoid having two unassigned variables which are compatible with all the previous assigned variables but not with each other.

The next section briefly discusses the effect that the choice of which variable to instantiate next can make on the search performance. As well as the variable ordering, we also discuss the impact a chosen value ordering from the variable's domain can have in the efficiency of the search strategy.

4.4.1 Variable and Value Ordering

The ordering in which the variables are instantiated and the values are chosen play an important role in the number of backtracks required. The number of backtracks is one of the most important criteria of measuring the efficiency of an algorithm [88]. In an FC algorithm, the variable ordering can further affect the amount of search space cut off [42] by pruning inconsistent forward potential assignments.

The order of instantiation of the variables may be static. This means that their assignment order is determined from the outset of the search and it is not changed thereafter. On the other hand the order may be dynamic, in this case the variable to be assigned next is chosen in the course of the search based on heuristics which take into account the current state of the search.

In order to implement a dynamic variable ordering within a tree search algorithm, extra information on the domains of the variables needs to be available. Therefore, dynamic variable ordering is not feasible for a simple backtrack algorithm, as there is no information with respect to the variables' domains. However, with FC the current state of the search includes the domains of variables after being pruned by the current set of instantiations.

The *fail first* (FF) principle is often used as a rationale for dynamic variable ordering [42]. This principle consists of picking next the variable which minimizes the expected length of each branch: the hope is that this will minimize the search cost. However, Smith and Grant [84] showed that minimizing the length of branches does not necessarily minimize the size of the search tree and, thus, that the FF principle is unsound.

A common heuristic, often described as an implementation of this principle, is to choose next the variable with the smallest current domain. The smallest domain heuristic works well for many problems: it seems likely that the reason for its success is simply that it minimizes the branching factor at the current node. We can also, for instance as a tie-breaker, choose the most constraining variable (*e.g.* the one which constrains the largest number of future variables) which is likely to reduce the number of nodes lower down the search tree.

Once a variable is chosen to be assigned, the value ordering of assignment will influence how fast a single solution can be reached. However, if the aim is to find all solutions, value ordering is meaningless in chronological backtracking. The reason is that every value must be systematically tried to cover the entire search tree.

When the aim is to find a solution, the often recommended value ordering is the *succeed-first* strategy. The basis of this strategy is to choose a value, if it is possible to determine one, which most likely leads to a solution. The objective is to reduce the risk of having to backtrack to this variable and try an alternative value.

4.4.2 Modelling a Problem as a CSOP

In applications one may be interested in measuring the quality of the solutions obtained by the strategy applied to find them. In addition, sometimes the aim is to find the best solution.

In order to achieve any of the above goals, a Constraint Satisfaction Optimization Problem (CSOP) is defined as a CSP together with an optimization function f which maps every solution to a numerical value. This value will express the quality of a solution.

Therefore, a CSOP can formally be defined as a quadruple: (V, D, C, f), where V, D, and C are defined as before for a CSP (Section 4.2) and if S is the set of solution tuples of the associated CSP then

$$f : S \rightarrow \text{numerical cost of the solution.}$$

The function f is defined in terms of arithmetic expressions involving the variables in V. Therefore, if one knows that the sought solution cannot be greater than a value U_0 (if the optimization problem is of minimization), a constraint can be added to the set of the CSOP constraints being solved $(V, D, C + \{f(S) < U_0\}, f)$. Adding the bound constraint to the problem the variables are further constrained, this would prune a part of the search space reducing thus the effort in searching.

A solution for the CSOP $(V, D, C + \{f(S) < U_0\}, f)$ gives a new bound for the minimization problem, $f(S) = U_1 < U_0$. This way, new CSOPs can incrementally be solved until the optimal solution is reached.

4.4.3 Local Search

In practical large applications, one might be more interested in a quick, but good, solution rather than in the optimal solution which can take hours, perhaps days to be found. Therefore, a way to yield near-optimal solutions is desired from the user's standpoint.

Some authors have proposed the use of stochastic techniques to solve CSOPs in which near-optimal solutions are required. Tsang [87] carried out some experiments using genetic algorithms to solve a set of randomly generated CSOPs and compared the results to that yielded by a branch-and-bound approach solving the same set of problems. The results showed that the stochastic method used performed well finding good solutions within 90% of the optimal within only 10% of the time spent by the branch-and-bound to find the optimal.

Minton *et al.* [64] describes a simple heuristic method for solving large-scale constraint satisfaction and scheduling problems. Their idea consists of taking an initial inconsistent assignment for the variables and progressively going on repairing constraint violations until a consistent assignment is yielded. The combination of the local search employed together with the constraint propagation within the CP framework produced excellent results when compared with the traditional backtracking techniques.

Where the complete search fails to deliver good performance in time, local search approach has been shown to be a good alternative [5, 87]. The combination of these two approaches can thus bring the benefit of both sides. In one side a stochastic method delivering time performance when a good solution is needed within a limited time. On another side a CP framework which in addition to having the constraint propagation mechanism reducing the problem search space every time a decision is made, the possibility of searching for and proving the solution optimality is available when wanted.

4.5 An Object Oriented Paradigm for CP

In 1992, Puget [73] presented PECOS. PECOS was a Lisp-based Constraint Logic Programming Language package, whose the aim was to provide the user with means of encoding efficiently a new kind of constraint when needed.

The motivation to build PECOS came from the fact that to extend the other CP languages based on Prolog was difficult, because the built-in constraints in those languages were not written in Prolog, but in procedural languages such as C [73].

Since PECOS was developed using the OO paradigm, PECOS's developers concluded that a similar library in C++ could also be built. A new project was then launched by the French company called ILOG[1] to build what is called today ILOG Solver.

ILOG Solver is implemented as a C++ library [74] in such a way that everything is dealt with as an object: variables, constraints and even search algorithms.

[1] ILOG S.A. is a multinational company with its headquarters in Paris, France – `http://www.ilog.com`

The choice for the OO representation of the entities within ILOG Solver makes this tool very easy to extend. For this the user just needs to define new object classes. This is the case of ILOG Scheduler: a scheduling language built on top of ILOG Solver to tackle scheduling problems.

Furthermore, ILOG Solver allows one to build specialized frameworks for very complex problems by combining the access to the interface provided by the AC-5 (see Section 4.3.1), and the possibility for the user to define their own constraint propagation method [76].

4.5.1 Expanding the Constraint Programming Language

Based on ILOG Solver, ILOG Scheduler is a C++ library which provides the user with ways to represent *resources* and *activities*. Resources, in this case, work as abstract entities which, in practice, could represent the personnel or machines performing the activities requested [60].

In order to well express a variety of practical scheduling situations, there are two basic subclasses of resource provided by ILOG Scheduler: Capacity and State resources.

A capacity resource represents the positive number of copies, or instances, of the resource that are available to perform activities, whereas a state resource represents the transitions occurred by performing an activity. Therefore, a state resource can change its state depending on the activity it is performing, and moreover activities may require a particular state of a resource to be performed by that resource.

From the capacity subclass of resource two other subclasses, of interest in this thesis, are also provided: the Discrete and Unary resource classes. The first one is to express a capacity resource which can perform more than one activity at once. The second is limited to perform each activity in turn.

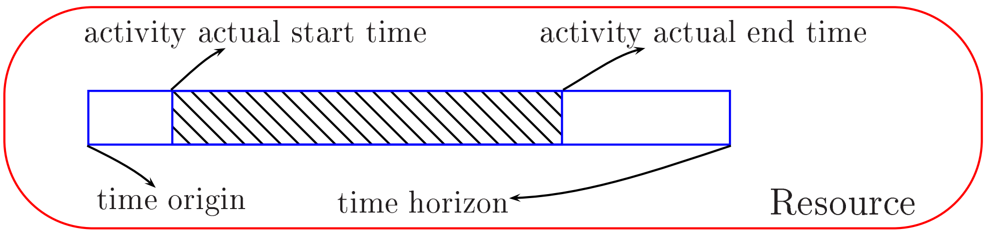


Figure 4.3: An activity when assigned to a resource.

Figure 4.3 shows an interpretation of the framework provided by ILOG Scheduler to represent activities and resources. In this case, an activity is being interpreted as an *interval activity* which executes without interruption from its start time to its end time [25]. Although in ILOG Scheduler the duration variable of an activity object can also be given by an interval variable to express a flexible duration, we will consider the duration only as a fixed variable in this thesis. In the example in Figure 4.3 the duration of the activity is represented by the hatched rectangle. However, note that an activity can start being performed on a resource from its allowed *time origin*, but finish not beyond its *time horizon*.

These constraint time-points (origin and horizon) to perform an activity may be given either by the definition of the problem, or by the reduction of the resource's available capacity by the demand of other activities equally assigned to the resource. The constraint propagation in this framework works thus updating this interval every time a schedule decision is made on fixing a start time for an activity. A scheduling algorithm fails if no time slot is left to execute an unperformed activity.

More formally defined, the start time of an activity A, accessible within ILOG Scheduler by *A.startVariable()*, is an interval variable $[\underline{s}, \overline{s}]$, where $\underline{s} =$ *time origin*, and $\overline{s} =$ *time horizon - duration*. Similarly, the end time variable, *A.endVariable()*, is an interval variable $[\underline{e}, \overline{e}]$, where $\underline{e} = \overline{s}+$ *duration*, and $\overline{e} =$ *time horizon*.

For a complex variable structure such as an activity, a set of temporal-constraint operators are provided within ILOG Scheduler. For instance, activities can be linked together by an operator like *activityA.startsAfterEnd (activityB, delay)*, meaning that *activityA* cannot start before the end of *activityB* given a delay of *delay*. *delay* is the minimum elapsed time between the end of *activityB* and the beginning of *activityA*, and if this value is negative means that *activityA* can start before the end of *activityB*, but the difference between the end of one and the start of another cannot exceed *-delay*. These operators constrain the time interval within which an activity can be executed.

Many powerful features have been incorporated into the tool. For instance, *edge-finding* (see Section 3.5.1.2, pp. 22) is an inference method to determine if an unperformed activity should be ordered before or after a fixed set of operations to be performed on the same resource [23]. This information can be used to advantage during the search [61] (see Section 5.2.5, pp. 76).

The special propagation process devised within ILOG Scheduler to reason over the interval activities and the operators provided to relate resources and activities show how ILOG Solver is highly extensible.

In addition to representing new classes of variables in ILOG Scheduler, the user still can access the variables and constraints of ILOG Solver. This facility opens up the possibility to represent specific constraints, and to implement and combine different strategies in order to provide better problem-solving strategies for the scheduling application under consideration.

4.5.2 Defining new Constraints

In order to define a new constraint type, the user needs just to define a new object class. This new object is inherited from the class *IlcConstraint*, an abstract class defined within ILOG Solver.

The IlcConstraint class provides the user with two main interfaces: a posting method, and a propagation method. The **post** method actually installs the defined constraint to the ILOG Solver framework. In doing so, the new constraint becomes part of the objects handled within the framework, along with the primitive variables.

The **propagate** method defines how the constraint must be propagated. Any C++ code can be used within the body of this function, and moreover the function can have entire access to the ILOG Solver variables. This interface gives to the user a great deal of expressiveness to create very complex propagation code.

When posting a constraint, the user chooses by which event the constraint's propagation method must be activated. This choice is made by installing the constraint by one, or more, of the following variable's domain methods: **whenValue**, **whenRange**, and **whenDomain**.

The **whenValue** is to trigger the constraint's propagation method when any value of the corresponding variable domain involved within the constraint is changed. The **whenRange** method triggers the propagation when the range is changed, and the **whenDomain** method when the variable's domain is changed.

To illustrate the use of these facilities, within ILOG Scheduler, provided by ILOG Solver, a new constraint to bound the start time of an activity B to the end of activity A is defined. The definition of *MyStartsAtEnd (activityA, activityB)* constraint is as follows.

```
class MyStartAtEndI : public IlcConstraintI
{
  IlcActivity _A, _B ;
public:
  MyStartAtEndI(IlcManager m, IlcActivity activityA, IlcActivity activityB):
    IlcConstraintI (m), _A(activityA), _B(activityB){}
```

```
  ~MyStartAtEndI () {} ;
  virtual void propagate () ;
  virtual void post      () ;
} ;
```

IlcActivity is an ILOG Scheduler object class to define an activity, and variables *_A* and *_B* of type *IlcActivity* are references to the external variables. *IlcManager m* is the reference variable that handles the entire ILOG Solver framework structure.

```
void  MyStartAtEndI::post ()
{
  _A.EndVariable().whenRange ( this ) ;
  _B.StartVariable().whenRange ( this ) ;
}
```

The propagation of the MyStartAtEnd constraint is triggered whenever the range of the end time variable of activity *_A* changes, and whenever the range of the start time variable of activity *_B* changes.

```
void MyStartAtEndI::propagate()
{
  IlcManager m = _A.getManager () ;
  m.add ( _A.EndVariable() == _B.StartVariable() ) ;
}
```

The propagation for this constraint is so that whenever the range of the end time of variable *_A* (the external variable *activityA*) changes, the system propagates this change to the start time variable of *activityB*. This way *activityB* is being forced to start right at the end time of *activityA*. Note that the propagation is *multi-directional*, therefore any changes to *activityB*'s start time also reflect back to the end time of activity *activityA*, and thus to the start time of *activityA*.

The potential for creating tailored and complex constraints is provided to the user within this framework. In addition to the potential for the user to create their own specialized constraints, the way the framework is devised also allows the user to borrow sophisticated solving techniques from Operational Research, if wanted.

4.6 Some Existent Tools for CP

Unlike the tools for CP such as ILOG Solver and ILOG Scheduler that are based on a procedural language (C++), many of the other tools are not. The list of CP tools we mention in this section have two other flavours. They are based on one of the two programming paradigms: functional or logic paradigm. This list however does not intend to be exhaustive, as more and more CP tools have emerged.

Logic Programming languages are general programming languages based on mathematical logic, and more specifically a subset of first-order logic. The main representative of the logic programming languages is Prolog, with its numerous dialects.

Prolog is in fact a restriction of the first-order logic language to Horn clauses. It acts as a theorem prover, and applies a general inference mechanism based on the resolution principle [79]. Prolog has embedded a chronological backtrack search algorithm (*generate-and-test* strategy) that enables to collect all the possible solutions for a given query clause.

Pure logic programming needs and only allows the statement of relations between terms, without any assumption on numerical properties. This fact makes of pure logic programming a cumbersome kind of language to deal with numerical problems.

Logic programming languages evolved then to the point where numerical problems, or the reasoning in domains more structured than the domain of syntactic terms, was possible to be dealt with. In other words, domains were associated to variables so that reasoning on the constraints could be actively used to restrict the domains before values have been tested on the variables. Thus the inefficient and passive *generate-and-test* approach in pure logic programming languages was replaced by the *constrain-and-generate* approach in the new logic programming languages to deal more efficiently with numerical constraints [33].

The roots of the Constraint Logic Programming (CLP) languages stem thus from the Logic Programming languages. One of the first attempts to introduce numerical constraints in Prolog (Prolog II) is attributed to Alain Colmerauer [27, 52].

In what follows we briefly give some more details about three constraint languages currently used commercially: CHIP, ECL^{i}PS^e and OZ.

CHIP – CHIP (Constraint Handling In Prolog) is a Prolog-based constraint tool combining thus the declarative aspect of logic programming with the efficiency of constraint manipulation techniques. It is an extention of the research prototype, also called CHIP, developed at the European Computer-Industry

Research Centre (ECRC) in Munich. CHIP has now been developed and distributed by COSYTEC[2]. Many commercial applications have already been developed using CHIP as the CP framework for optimization problems.

In a CHIP program, constraints are syntactically treated as predicates, as many other Prolog-like languages. Furthermore, CHIP provides a large number of pre-defined constraint predicates in a variety of domains: integers/finite domains, rationals and booleans.

CHIP provides the user with demon mechanisms [52]. These demons have the effect of re-evaluating a specified goal each time a given triggering event occurs, for instance, a change in a variable domain.

ECL^{i}PS^e – ECL^{i}PS^e (ECRC Common Logic Programming System) is a Prolog based language like CHIP. As CHIP, ECL^{i}PS^e is also one of the research fruits of the ECRC, and is now being further developed and maintained at IC-Parc (Centre for Planning and Resource Control at the Imperial College in London)[3].

The ECL^{i}PS^e's added-on library to support finite domain constraints, in addition to integers, also allows for atomic (*e.g.* atoms, strings, floats) and ground compound elements (*e.g.*, f(a,b)) [34]. As for CHIP, ECL^{i}PS^e has a great number of built-in constraint predicates, such as *all-different* and *at-most*.

Fernández and Hill [34] reported that ECL^{i}PS^e supports writing further extensions such as new user-defined constraints. However, these extensions demand from the user a good knowledge of the underlying system.

OZ – The OZ programming system has been developed by researchers from DFKI (the German Research Center for Artificial Intelligence), SICS (the Swedish Institute of Computer Science), the University of the Saarland, UCL (the Université Catholique de Louvain), and others collaborators. The software is freely available via Internet[4].

Unlike the other languages in the group of CLP, the constraint programming language OZ is instead a multi-paradigm language. OZ is based on the concurrent constraint model subsuming higher-order functional and object-oriented programming as facets of a general model [85].

[2]http://www.cosytec.com
[3]`http://www.icparc.ic.ac.uk/eclipse`
[4]`http://www.mozartproject.org`

Functional languages, such as Oz, are also declarative. Therefore, in theory, the programmer specifies *what* to compute, not *how* to compute it. However, search mechanism implementation in Oz is completely left in charge of the programmer, rather than being by default the left-to-right depth-first strategy as in a CLP language.

Moreover, Oz provides mechanisms to decide the satisfiability and implications for basic constraints of the form of $x = n, x = y$, or $x :: D$ where x and y are variables, n is an nonnegative integer and D is a finite domain. This basic constraints reside in what is called in Oz terminology the *constraint store*. Non-basic constraints, however, are not placed in the store. Therefore, more complex constraints such as $x + y = z$ are left in charge of the *propagators*. In Oz, a propagator is actually an agent which is in charge of updating the domain of variables it is related to [34, 85].

A thorough comparison between these languages and others is given by Fernández and Hill [34].

4.7 A Few Applications of CP

This section points out some of the applications of CP found in the literature where some of the languages cited in the previous section were used.

Constraint Programming has by now been widely used to solve real-life problems in many areas. In Dincbas *et al.* [32] we find one of the first successful practical applications of CP. They used Chip to solve a real-life two-dimensional cutting stock problem in a furniture factory. This problem consists of cutting a larger material into smaller pieces according to customers' requirements, so that the waste is minimized. Their results turned out to be comparable in efficiency to the tailored programs written in procedural languages.

The Crew Rostering Problem (CRP) is another type of problem which has received great attention in the air and rail transport industries. This problem and its variations are well known in OR, and considered to be very complex given the operational constraints deriving from union contract and company regulations. Briefly, the CRP aims at determining a sequencing, optimal if possible, of a given set of duties into rosters satisfying the constraints resulting from the agreement between the union and the company.

Caprara *et al.* [20] propose a combined method to solve this problem for the Italian Railway Company. They used ECL^{i}PS^e and an OR procedure, implemented using the procedural language C, to calculate the Lagrangian relaxation lower-bound for the problem. They compared the combined approach against using an OR approach solely. The results are comparable in terms of quality and performance. However, the combined approach was simpler to implement.

Another area where CP has been applied is Timetabling. One variation of this problem consists of finding a weekly timetable for the courses offered by a college, so that a set of constraints is satisfied. Amongst many other constraint programming solutions to this kind of problem, Henz and Würtz [44] present a timetabling application implemented in Oz to solve the timetabling of a College for Social Work in Germany.

More recently, an entire issue of the ”Constraints” journal (volume 6, issue – 2/3 June 2001) has been dedicated to applications of CP to the newly flourishing bioinformatics area.

The above examples show just a few of the numerous areas where CP has been applied. A thorough coverage of the use of CP in a wider variety of applications is given in [15, 52, 91].

4.8 Summary

In this chapter we presented a brief description of the foundation of constraint programming paradigm. Although the CSP paradigm can be traced back to research in the sixties and seventies, only recently the CP paradigm has been attaining more support also from the commercial applications sector.

The fruitful results of years of research are now incorporated within commercial tools as inference mechanisms for transforming a problem into another equivalent, but simpler problem to be solved. These developments have been provided practitioners with means of devising successful large real-world applications.

In the particular case of Ilog Solver, because this tool is devised under the OO paradigm, it is easy to extend the tool to a more specific class of problem. However, we believe that this could also be done in those tools based on Prolog.

Ilog Scheduler is an extention of Ilog Solver to the scheduling classes of problems. In this chapter we also described how the user can devise their own constraint within Ilog Scheduler. The possibility to codify their own propagation method, using any C++ piece of code, allows the user to borrow sophisticated constraint solving techniques from Operational Research, for instance, and incorporate it within their algorithm.

Chapter 5

Single-Track Scheduling using CP

5.1 Introduction

This chapter describes two slightly different models to map the single-line track railway scheduling problem into a special case of the job-shop scheduling problem.

In both models, only one train can occupy a track segment at one time. However, more than one train may be scheduled to use the same stretch of track at a time in the desired timetable. This conflict is resolved differently in each of the two models presented in this thesis.

5.1.1 Outline of First Model

In the first model, the conflict between two trains planned to use the same track segment simultaneously is resolved by re-timing one of the trips at its departure time up to the point the conflict is resolved. This model has its drawback, as it is not flexible. It does not allow re-timing only from the point when the conflict occurs. This first chosen strategy to resolve conflicts stems from the observed practice of train operators. Train operating companies do not typically want to plan for passenger trains being delayed after their departure.

An entire trip is formed by a sequence of small pieces of trips linking the departure origin s_0 up to the arrival destination s_f points. Each of these pieces represents the travel between two distinct points between the s_0–s_f stations.

Three algorithms are proposed to deal with this first model. Each of these algorithms is based on a *depth-first* strategy with bounds to prune fruitless parts of the search tree. The first algorithm is, therefore, a branch-and-bound algorithm devised to solve the single-track railway scheduling problem using this model. The algorithm is based on picking the next trip in chronological order and re-timing it, if it conflicts with any other trip [69].

The second algorithm is based on locally adding a precedence constraint between the conflicting pieces of trips. The remaining unscheduled and unconflicting pieces of trips are thus automatically re-adjusted by the propagation of the new start time of the current delayed piece of trip. The performance of this algorithm greatly improves on finding optimal solutions for the problem set used in this thesis, in comparison with the first algorithm.

The third algorithm is composed of three heuristics. The first heuristic finds quickly a good first solution for the problem. The other two parts of the algorithm decouple the problem into two other subproblems based on an imposed bound on the most delayed trip.

The second heuristic searches for good solutions for the problem by imposing increasingly strong upper-bounds on the most delayed trip. Each solution found gives a new upper-bound for the next solution's cost of the problem. This heuristic terminates when it is not possible to further increase the upper-bounds on the most delayed trip.

The third heuristic works complementarily to the second heuristic trying to find solutions for the problem while increasingly delaying one trip. Similarly to the previous heuristic, each solution found gives a new upper-bound for the next solution's cost of the problem. This heuristic terminates when all the improved solutions have been found within the limited area of searching passed to the heuristic as parameters.

Although these three heuristics are devised in such a way that when they are used together the optimal solution will always be found, they could be used independently.

5.1.2 Outline of Second Model

Unlike the first model, the second model proposed resolves a conflict by delaying only the conflicting piece of one of the two trips (and subsequent pieces of that trip). In this way, the section of the trip before the conflicting point does not need to be re-timed. This procedure yields solutions with lower total delay.

This second method of resolving conflicts differs from the strategy adopted in the first model, because it can also handle the usual practice with passenger trains of not delaying them after their departure, as well as situations in which a train may be delayed along the trip in order to resolve a conflict. In Higgins' model, this flexibility is not provided: every train can be delayed at any passing point, should a conflict need to be resolved.

However, in the experiments carried out with the second model set of problems we consider, like in Higgins' model, that every train can be delayed at any passing point in order to resolve a conflict.

In addition to change the way a conflict is resolved, in the second model we use some of the techniques successfully used for the job-shop problem.

The job-shop scheduling problem has been studied for decades in the scientific community, as pointed out in Chapter 3. Many techniques have been applied to handle this difficult problem, well known to be NP-complete [37].

The strategy often used to solve a job-shop scheduling problem consists of two phases. The first phase is often based on a fast heuristic to find an initial solution. In a solution of a job-shop problem, one set of activities plays a very important role. These activities, which cannot be rescheduled without changing the objective cost [71], are called *critical activities*, and each path from the earliest to the latest critical activity forms a *critical path* (or set of critical paths) (see Section 3.4, pp. 19).

The second phase may use the critical path (or set of critical paths) of a job-shop problem solution as a powerful feature to search for a better solution to the problem. By swapping activities in the critical path one may transform the current solution into a better sequence of activities which can improve the solution. This way one may walk through better and better solutions more quickly than by navigating through every branch of a search tree.

This chapter also describes the way the critical path is constructed for the single-track railway scheduling problem in which the objective function is not the *makespan*, as usually adopted in classical job-shop scheduling problems, but the *total tardiness*.

A great improvement in performance was achieved by using a hill-climbing method combined with the second algorithm used in the first model. The hill-climbing algorithm uses the critical path structure of a solution to walk through improved solutions. This local search strategy calls the branch-and-bound algorithm when trapped in a local cost minimum.

The new combination of algorithms finds the optimal solution in a reasonable amount of time for 19 out of 21 real-world problems gathered from [45]. However, proving optimality for the two of the largest problems of the set is still hard [70].

The following sections describe each of the models applied, as well as the respective algorithms to solve the problems.

5.2 First Model

The single-track railway can be modelled as a special case of the job-shop scheduling problem. This can be achieved by considering the train trips as jobs, which will be scheduled on tracks, regarded as resources. A train trip may consist of many tasks that require traversing from one point to another on a track. Each of these distinct points can be a station or a signal placed along the track separating track segments. In order to eliminate a conflict a new departure time is chosen for one of the conflicting trips, so that the conflict is resolved.

The extension made in this thesis to the classical job-shop scheduling problem model is in the way a single-line railway is conceived as a sequence of resources. The single-track railway network considered here is such that only one train can occupy a track segment at a time, whereas more than one train can be at a passing point at a time as long as its capacity limit is observed. Therefore, in the extension made here, resources can deal with more than one task if their capacity allows.

Formally stating the single-track railway scheduling problem, a train trip J_i is described as $J_i = \{o_{i1}, o_{i2}, \ldots, o_{ik}\}$, where o_{ik} is the $k^{th} = \mid J_i \mid$ task of J_i. Each operation o_{ij} has its planned departure time d_{ij}, as well as its fixed processing time p_{ij} to complete. In addition, each of these operations requires processing on a unique stretch of track $r_i \in \mathcal{R}$, the set of track segments along the whole track network. In this first model, the consecutive operations of a job must be performed in such a way that

$$\overline{d}_{i(j+1)} = \overline{d}_{ij} + p_{ij}, \forall j \in \{1, 2, \ldots, k =\mid J_i \mid\}, \tag{5.1}$$

i.e. there is no delay between the end time of an operation and the start time of the next operation, and $\overline{d}_{ij} \geq d_{ij}$ is the actual departure time of o_{ij}. This approach was first adopted because, intuitively, it more closely represents the way a passenger trip is scheduled in practice. Typically, train operators do not schedule passenger trains to be delayed between their origin and destination points, although it sometimes happens that a slow (stopping) train is delayed to allow an express train to overtake it.

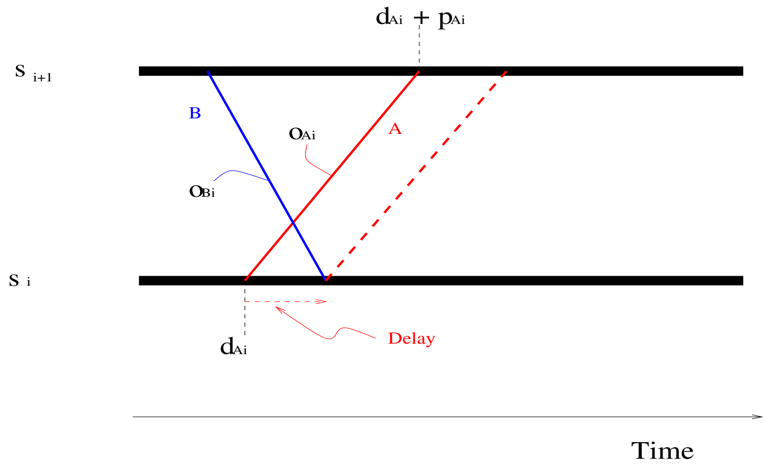


Figure 5.1: A typical conflict case.

The fact that a subsequent task of a job must start immediately after its antecedent also resembles the *no-wait* requirement phenomenon in some scheduling problems [56, 71] (see Section 3.5.3, pp. 32). This sort of problem is also known to be NP-complete. An example of such an operation is a steelmaking mill in which a slab of steel is not permitted to wait between two consecutive steel processing machines because it would cool off [83].

A delay to the departure time of the last task of a trip $T_i = \overline{d}_{ik} - d_{ik}$ is caused when any task of a trip is delayed from its planned departure d_{ij} in order to resolve a conflict on a track segment. Figure 5.1 shows a typical situation where a conflict is resolved by delaying task o_{Ai} in order to cause the minimum delay for the set of trips involved in the conflict.

In Figure 5.1, task o_{Ai} is performed when train A departs from signal s_i at d_{Ai} time units (or $\overline{d}_{Ai}$, if it is delayed) and takes p_{Ai} time units to reach the signal s_{i+1} at the end of the track segment.

To bring all those elements described above into a job-shop scheduling context, consider $\mathcal{J}$ is a set of jobs $\{J_1, J_2, \ldots, J_n\}$ (trips in the train context), where each job J_i has a set of tasks $o_{ij} \in J_i$ $(j = 1, 2, \ldots, k =| J_i |)$ to be performed. Every job is considered to have equal priority.

The d_{ij} variable value is the operation o_{ij}'s release time, which is a point in time from where a task can start its processing.

Let $\mathcal{R}$ be the set of resources (or machines). In both models $\mathcal{R}$ is assumed to be ordered and a task is to be performed on a specific resource. So, if o_{ij} is assigned to $r_k \in \mathcal{R}$, $o_{i(j+1)}$ is assigned to r_{k+1} when the task belongs to a job corresponding to an outbound train; and to r_{k-1} when it belongs to an inbound train.

Track segments are resources such that *capacity*$(r_i) = 1$, whereas a passing point has *capacity*$(r_i) > 1$. Moreover, the number of conflicts at a time t on a track section is given by *conflicts*(r_i, t) and the overall delay is given by the expression:

$$D = \sum_{1 \leq i \leq n} T_i \tag{5.2}$$

The aim is then to find a feasible departure time for all tasks, so that the conflicts are resolved, the additional constraints are satisfied, and the overall delay (or *Total Tardiness* in the job-shop context) given by the expression in (5.2) is minimized.

5.2.1 A Practical Example – 1^{st} Model

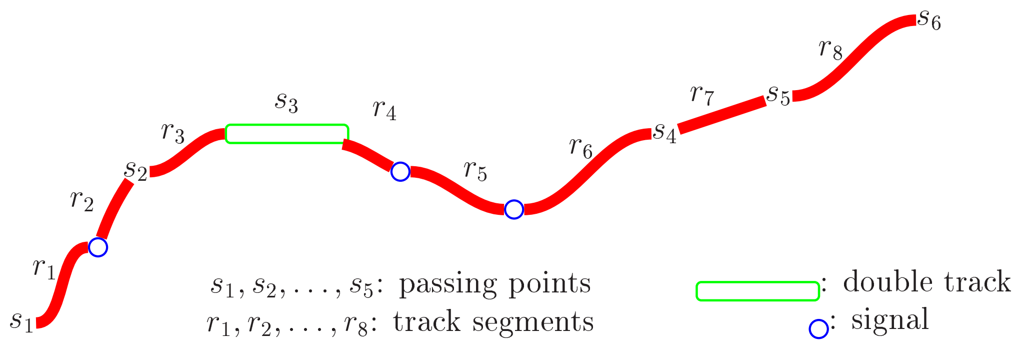


Figure 5.2: A single-track railway network: an example.

To exemplify the model described in the previous section, a small though illustrative example is given. Consider the desired timetable with four trains presented in Figure 5.3, where trains #11 and #13 are inbound and #14 and #16 are outbound trains.

The single-line track considered (Figure 5.2) has the following characteristics. It has 8 track sections, 5 stations and a short double track section between station s_2 and s_4, this double track siding is considered in this paper as another station s_3. Between s_1 and s_2 there are 2 track segments, another one just after s_2 at the beginning of the double track section, and three others just after it before s_4.

Each station is considered, for the sake of this illustration, as having infinite *capacity*. An exception, of course, is station s_3, whose *capacity* is 2: only two trains can use that station at one time.

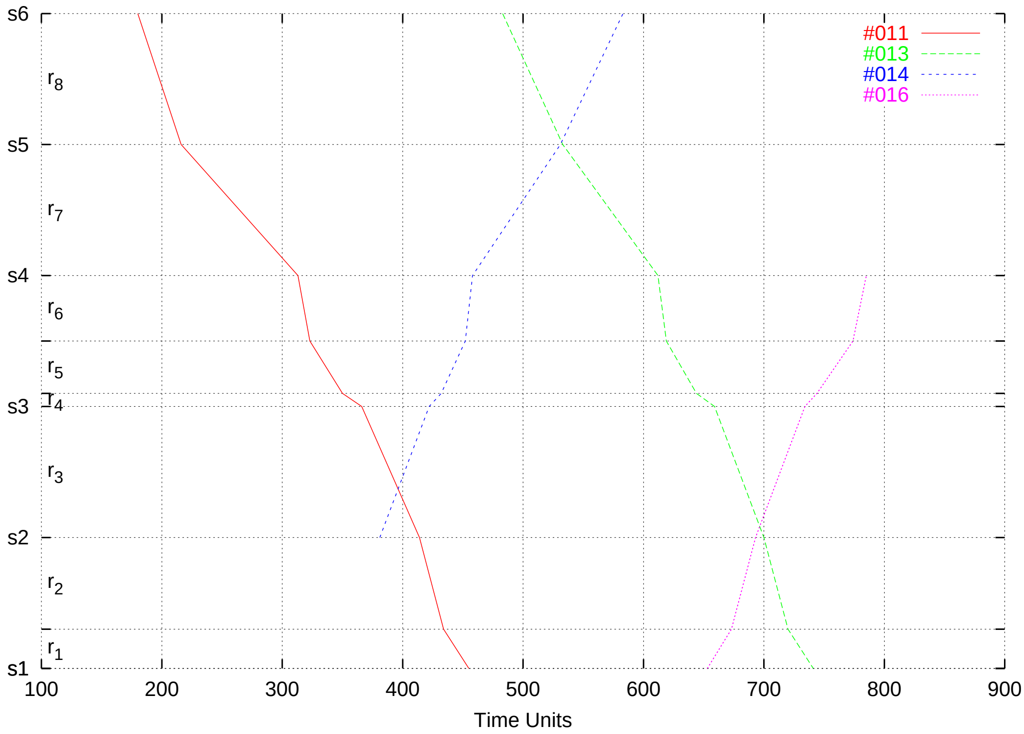


Figure 5.3: A given timetable with 3 conflict points at s_2–s_3, s_5–s_6 and s_2–s_3.

When the desired group of services was planned to be offered (see Figure 5.3), the fact that a track segment can only accommodate one train at a time was disregarded, and hence the timetable would result in three conflicts, as shown in Figure 5.3.

Trip ≡ Job	Machines' Sequence	d_i	p_{ij}
11	$r_8, r_7, r_6, r_5, r_4, r_3, r_2, r_1$	180	36,97,10,27,16,48,20,21
13	$r_8, r_7, r_6, r_5, r_4, r_3, r_2, r_1$	483	50,79, 7,25,15,41,20,21
14	$r_3, r_4, r_5, r_6, r_7, r_8$	381	41,10,20, 6,73,52
16	$r_1, r_2, r_3, r_4, r_5, r_6$	653	20,20,41,10,30,11

Table 5.1: A given infeasible timetable.

Another way of describing the given timetable presented in Figure 5.3 is as shown in Table 5.1.

In the example shown in Figure 5.3, we have trip $J_{11} = \{o_{11,1}, o_{11,2}, o_{11,3} \ldots o_{11,7}, o_{11,8}\}$(see Table 5.1). Each operation $o_{11,i}$ signifies a part of the trip J_{11} consisting of either travelling from one point to another along the track, or waiting at a passing point.

5.2.2 An Algorithm for the 1^{st} Model

The program developed to solve the problem described above has been implemented in ILOG Scheduler [48, 49], a C++ library that enables the user to represent scheduling constraints in terms of *resources* and *tasks*. Constraints can specify the sequencing order of any two arbitrary tasks, conditions on the utilization of a resource, setup time transitions, and so on. ILOG Scheduler is built on top of ILOG Solver, a constraint programming tool, which provides the mechanism of a systematic search that instantiates each variable, either according to the order they are presented or according to a specified heuristic. When a variable is assigned a value, the propagation deletes values from the other variable domains which are inconsistent with the assignment and the set of constraints posted. When either a variable domain becomes empty or no value can be assigned to the variable in order to satisfy all the constraints, a backtrack is needed in order to return to the previous assignment and choose another value for it to continue the search. A problem is infeasible when it is not possible to find values to instantiate all the variables which satisfy the constraints posted.

In order to resolve a conflict on a track segment, one of the conflicting tasks needs to be delayed on that track segment. This is done, in this first algorithm, by fixing an order of execution on the resource between the conflicting tasks. The first conflicting task is ordered to be performed at its earliest possible release time, and the second task is scheduled immediately after that. This decision is propagated to the entire trip. According to the model proposed in Section 5.2, each task must start as soon as its predecessor ends, therefore, the entire trip is also delayed when one of its tasks is delayed.

Given the structure of the problem, we decided to use a *chronological strategy* to resolve the conflicts. The idea is to order each stretch of trip on a track segment, where the earliest unresolved conflict occurs, starting from left to right in the Time-space Diagram up to the point where the earliest conflict is eliminated. Therefore, this process starts from the earliest conflict, on the left side of the Time-space Diagram, to the last operation up to the right edge of the diagram. This strategy is used because when resolving a conflict, by ordering the conflicting tasks a new conflict may arise. This new conflict will certainly be later than the current conflict being resolved. In addition, because the previous activities are only ordered, rather than fixed to be performed at a specific time, any new further delay to a trip already ordered earlier on does not create any additional conflict.

The algorithm devised, following this outline, terminates when there is no remaining conflict to resolve. In addition, it works as if *pushing* the activity which causes a bigger delay to the right side of the diagram. However, each decision to delay a task is made locally, ignoring the consequences it may cause for the rest of the trip or on other trips.

Algorithm 5.2.1 gives the pseudo-code outlining the strategy used to solve single-line track railway scheduling problems according to the model presented in Section 5.2. [1]

Algorithm 5.2.1

1 Schedule all track segments which a conflict is found on it
2 **begin**
3   **while** $\exists r_i \mid conflicts(r_i, t) \neq 0, t \in [0..horizon]$ **do**
4   pick a r_i in $\{r_i \in \mathcal{R} \mid conflicts(r_i, t)$ is the earliest$\}$
6   **do begin**
8   – selects a task by applying the `Shortest Processing Time`
9   over r_i's tasks; and
11   – try to schedule it after those which have already been scheduled.
13   **if** the selected task cannot be in this order
14   **then** `backtrack` choosing another task
15   **fi**
17   **end**
18   **od**
20   **od**
22 **where**
23 **proc** *Shortest Processing Time*$(o_A, o_B) \equiv$
24   o_A, o_B are two tasks on a track section
25   **if** $d_A + p_A - d_B < d_B + p_B - d_A$
26   **then** o_A is chosen to be scheduled first
27   **else** o_B is chosen instead
28   **fi.**
30 **end**

While there still are conflicts on any section (*i.e.* resource) r_i at any time within the scheduling horizon (line 3), Algorithm 5.2.1 chooses the track section which has the earliest conflict to schedule first (line 4).

[1] This pseudo-code was written within the LaTeX's *program* environment.

On a resource, tasks which are not conflicting with any other task are initially ordered according to their earliest possible departure time. For those which conflict, the *Shortest Processing Time* heuristic [71] (line 8) is used to schedule conflicting tasks and construct a partial solution along with the already unconflicted ones. When a chosen task cannot be scheduled after those which have already been scheduled on the resource, because it would increase the already known total delay (see expression (5.3)), the program backtracks (line 14) in order to choose a different order of tasks on this resource. When no other task can be placed without increasing the total delay, it backtracks to an upper level in the search tree, choosing a different resource to re-order if necessary.

A trip, as described in Chapter 2, may be scheduled at any point in time between its specified earliest departure and the *horizon* schedule limit, which is calculated as $horizon = d_* + \sum_{J_i \in \mathcal{J}} \sum_{o_{ij} \in J_i} p_{ij}$; $d_* = \{d_{ij} \mid \text{is the earliest departure time}\} \; \forall i, j$.

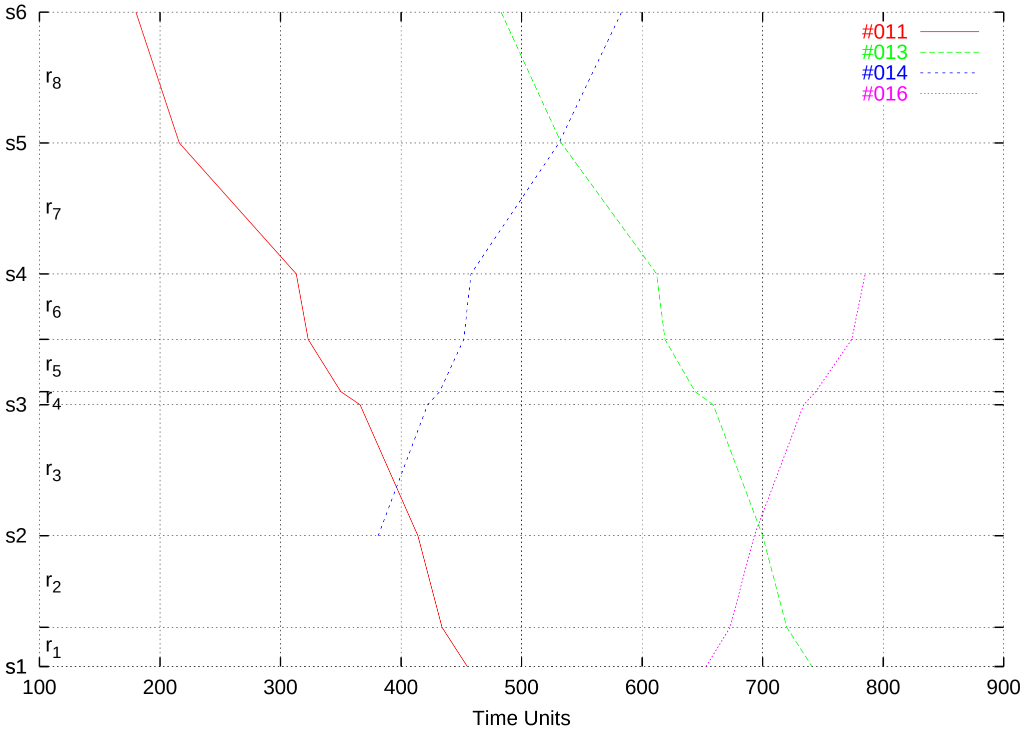
Initially, each $\overline{d}_{ij}$ variable can range from d_{ij} to *horizon*. This huge range of possible values for each variable is because it is not possible to guess what a potential solution configuration is going to look like beforehand, given that the system may have constraints of various kinds which need to be satisfied all together.

Once a solution is found, an important value is gathered, which is the total delay D, as defined by the expression (5.2, pp. 62), so far obtained. Thereby a new constraint can be included to the set of constraints of the problem, so that the total delay of the next solution must be less than the current D. In other words,

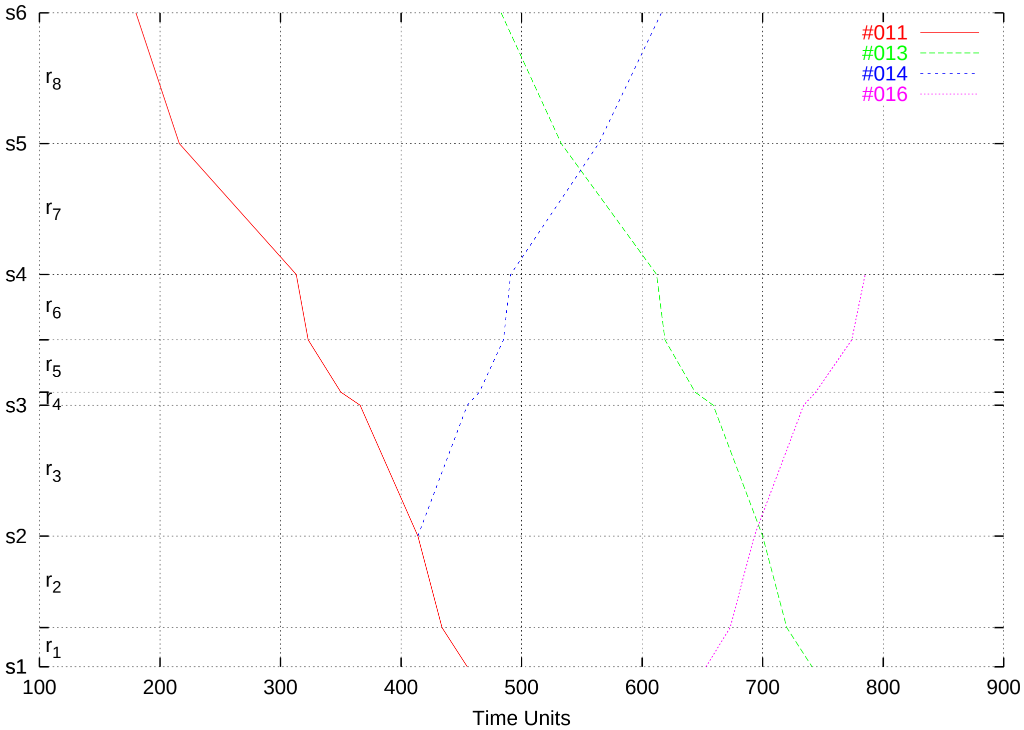
$$\sum_{J_i \in \mathcal{J}} \sum_{t_{ij} \in J_i} \overline{d}_{ij} - d_{ij} < D; \quad \forall J_i \in J. \tag{5.3}$$

This constraint reduces the possible movement in time of a task and so reduces the number of possible orderings a task can take, and consequently a trip; in other words, it discards branches of the search tree which would not lead to better solutions than what has already been found.

Taking the desired timetable in Figure 5.3 (pp. 63) as an example, the algorithm takes the first chronological conflict between train #11 and #14 to resolve. The conflict exists because train #11 is planned to depart from station s_3 at 366, arriving at s_2 at 414 time units, and train #14, on the other hand, is planned to depart from s_2 at 381 and arrive at the passing point s_3 at 422. In job-shop terminology, it is to say that operation $o_{11,6}$ is released at 366 to be performed on resource r_3, the track segment between station s_2 and s_3, and this task is performed in 48 time units (see Table 5.1, pp. 63). Likewise, operation $o_{14,1}$ is planned to start at 381, on the same



(a) Original Problem



(b) Algorithm's 1^{st} step

Figure 5.4: (a) The original problem previously presented in Figure 5.3 pp. 63. (b) The algorithm's first step is to resolve the 1^{st} chronological conflict, which is that between train #11 and #14 between station s_2 and the passing point s_3.

resource, and complete at 422.

According to the SPT dispatch rule used in the algorithm, delaying the whole trip #11 up to the point where this train arrives at 422 at the passing point s_3, would cause a delay of 56 time units. The worthwhile option is then to delay train #14 causing a smaller delay of only 33 time units. Figure 5.4(b) shows the first step of the algorithm. Note that, once the decision is made, it is propagated to the rest of the system. This initial ordering decision changes the point where trains #14 and #13 conflict.

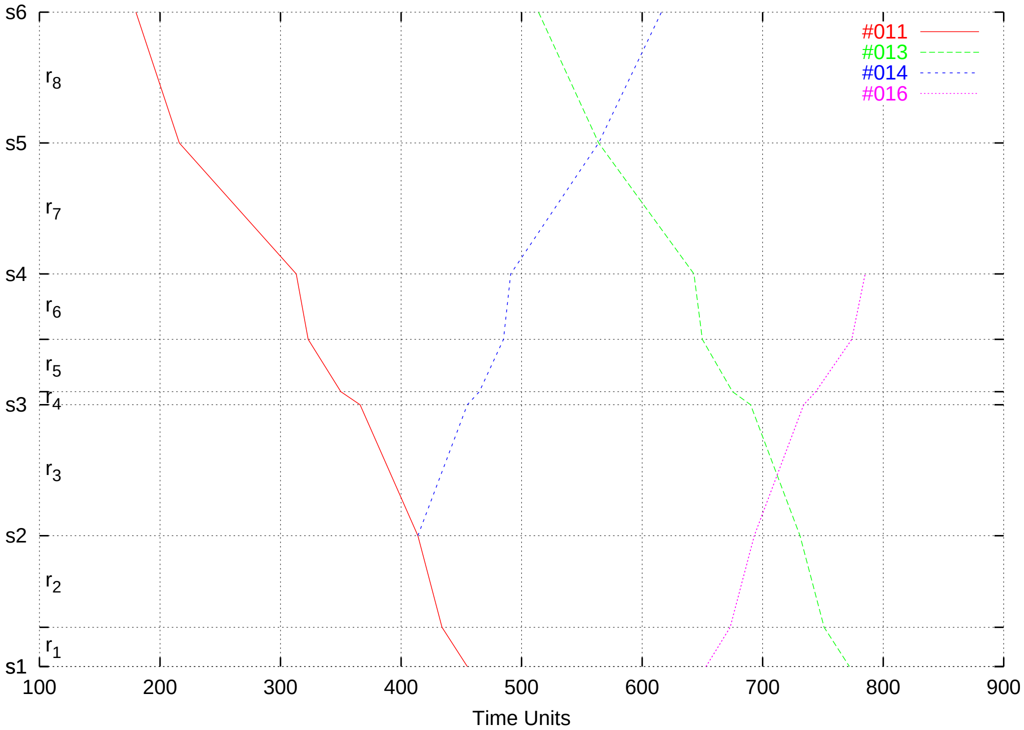
The consequence of the above ordering is that the problem still has left two conflicts to resolve. However, the next conflict chronologically has been changed from between tasks $o_{13,1}$ and $o_{14,8}$ to tasks $o_{13,2}$ and $o_{14,7}$. Therefore, the next track segment to be ordered is that between stations s_4 and s_5, resource r_7. Because the earliest task on this resource is $o_{11,1}$, which is not conflicting with any other operations, it is initially ordered to be the first task performed on this resource.

Train #13 is currently to depart from station s_5 at 533, as originally planned. On the other hand, as a result of the previous delay of 33 time units, train #14 is now planned to depart from station s_4 at 491. In other words, the release time of operation $o_{14,7}$ is now 491 on resource r_7, whereas operation $o_{13,2}$ is to start at 533 on this resource. According to the SPT rule, the choice is to delay train #13 by 31 time units to resolve this conflict. The alternative would be to further delay train #14, however, this would increase further the overall delay by 121, instead. Hence, the current overall delay is up to 64 when the second conflict is resolved by delaying train #13. The configuration of the trips is shown in Figure 5.5(a).

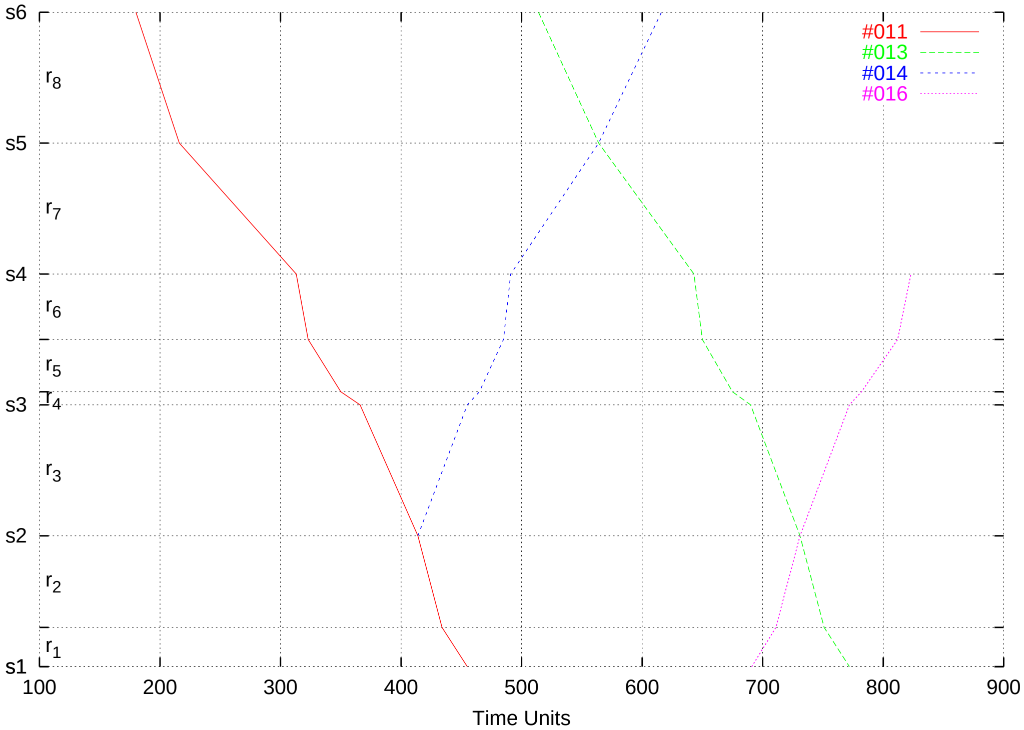
Finally, the only conflict left is between train #13 and #16 on track segment (s_2, s_3), resource r_3. This resource has already a partial order due to the ordering decision when resolving the first conflict between operations $o_{11,6}$ and $o_{14,1}$.

The last conflict is between the same operations as previously, but the point in time of the conflict has changed. Train #13 has been delayed by 31 time units, as a result, operation $o_{13,6}$ is to start now at 690, instead of at 659 time units as initially planned. Task $o_{16,3}$ is yet undelayed. Its planned release time on resource r_3 is at 693 and completion at 734 time units. Once again, using the SPT rule, if trip #13 is delayed, the overall delay would increase by 44 time units, whereas if the choice of delaying is on trip #16 the increase is only by 38. Therefore, trip #16 is chosen to be delayed resulting in a total delay of 102 time units. This first solution is depicted in Figure 5.5(b).

This initial solution with a total delay of 102 is not the optimal solution for



(a) Algorithm's 2^{nd} step



(b) Algorithm's 3^{nd} step

Figure 5.5: (a) Subsequently second algorithm's step when the conflict between train #14 and #13 is resolved. (b) The third step is to resolve the last conflict between train #13 and #16. The first solution is reached with a delay of 102.

this example. However, with the upper-bound of 102 found, an extra constraint $D < 102$ (see expression 5.2, pp. 62), is added to the problem's constraints forcing the algorithm to search for another solution whose total delay is smaller than the current one found.

The algorithm backtracks trying a different order from those already tried. The first re-order to try is reversing the order between train #13 and #16. This time trip #16 is tried to be the first on the track segment r_3 before trip #13. This would increase the overall delay up to 108, which is not allowed by the new constraint posted above. Next step is to try to change the previous decision point of the search tree, the conflict between train #14 and #13.

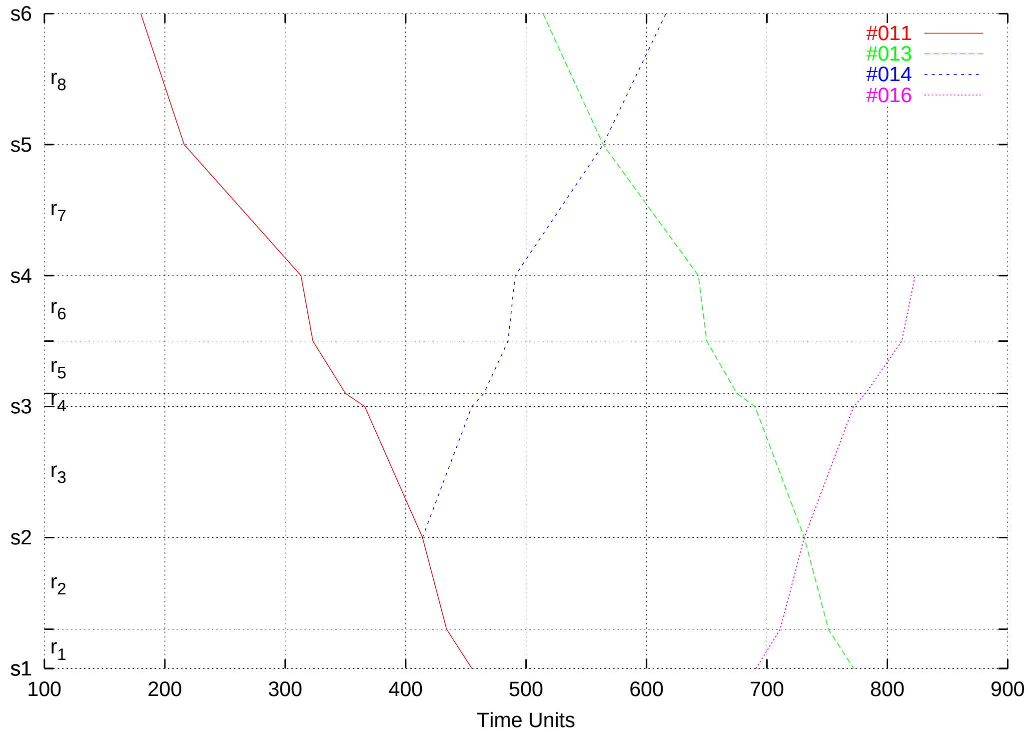
The choice, when resolving the conflict between #14 and #13 was to delay train #13 causing a delay of 31, against 121 otherwise. Hence, delaying #14 this turn, the overall delay goes up to 154, considering the delay when the first conflict is resolved. So, this is not an option either. The last resort is try to change the decision made at the top level of the search tree.

The first option was to let train #11 travel first through track segment r_3, because the local optimization ruled by the SPT procedure would give a delay of 33, instead of 56 otherwise. Now, the option is to try a different order, train #14 goes first on that stretch of track, giving an initial delay of 56 time units.

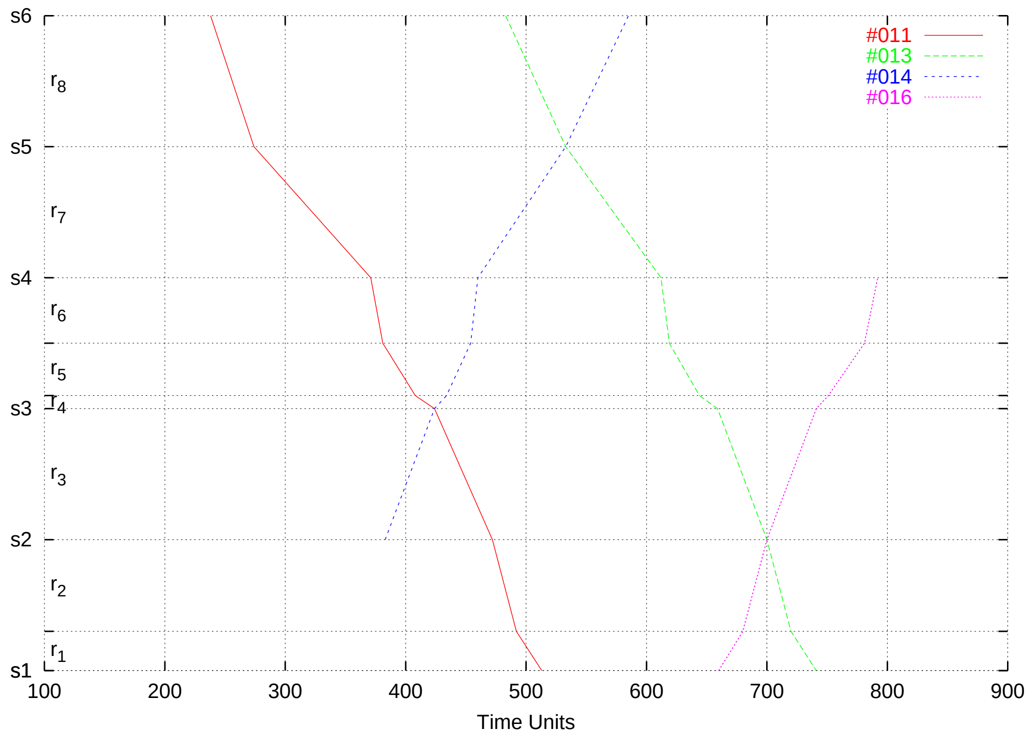
As train #14 is not delayed this time, $o_{14,6}$'s release time is 531, whereas the planned completion time for operation $o_{13,1}$ on resource r_8 is 533. This leads to the choice of delaying the start time of operation $o_{14,6}$ by only 2 time units in order to get rid of the conflict between these two tasks. Nonetheless, when this decision is made, because of the current model, the whole trip #14 is also shifted in time, which forces trip #11 to be further delayed, as it is fixed to use the track segment r_3 after trip #14. Therefore, the total delay sums up to 60.

So far, only train #11 and #14 were delayed. Hence, the third conflict between operations $o_{13,6}$ and $o_{16,3}$ occurs because, as already stated, task $o_{13,6}$ is planned to be released at 659 and completed at 700 time units, whereas tasks $o_{16,3}$ is to start at 693 on the same resource. The choice is then to delay task $o_{16,3}$ by 7 time units, so that the conflict between these two trips is resolved. This way, adding the last delay of trip #16, the overall delay is 67 time units, an improved upper-bound.

Likewise with the initial solution, this new upper-bound value is added to the problem's set of constraints in order to force the algorithm to search for an even better solution. However, this solution is optimal, therefore, the program fails to find any other solution. This solution is shown in Figure 5.6(b).



(a) Algorithm's 3^{nd} step



(b) Optimal solution

Figure 5.6: (a) The first solution is with a delay of 102. (b) The optimal solution with a delay of 67.

5.2.3 Solving 21 Real-Life Problems – 1^{st} Model Results

The datasets used to perform the following experiments were provided by Higgins and were used in [45]. However, because the data provided was in some cases incomplete, it was necessary to arbitrate some values of the desirable departure and arrival time for some trains. Although these times may be slightly different from the original, the problems are still the same as far as the number of conflicts to be resolved are concerned.

This section has the purpose of presenting the performance of solving some problems from [45] using constraint programming, following the model presented in Section 5.2 and the algorithm in Section 5.2.2. The objective in any problem presented here is to minimize the overall delay.

Problems				
Number	Total	Inbound	№ of Passing points	№ Conflicts
20-29	7	4	6	6-11
40-46	9	4	6	8-14
50,51	25	12	12	27,35
60,61	30	15	16	62,69

Table 5.2: Problems' classification according to their number of conflicts.

Within these problems presented in Table 5.2 there are none of those special constraints discussed in Chapter 2. Therefore, the aim here is both to evaluate the performance of the algorithm used and compare the performance with that obtained by Higgins in [45].

In Table 5.2 the *Number* column gives the problem number, *Total* is the total number of trains to schedule in the problem, *Inbound* is the number of inbound trains, *№ of Passing points* states the number of passing points which trains can use to pass each other; and *№ of Conflicts* is the range of conflicts which the problems of the group have.

The problems are classified according to the number of conflicts they have in their given original timetable. Higgins suggested that the number of conflicts in a problem is a good measure of its difficulty [45].

These experiments were performed on a networked Silicon Graphics O_2 workstation. The results are presented in Table 5.3.

The *mDelay* column, in Table 5.3, is the total of each minimum necessary delay when resolving each conflict individually in a given planned timetable. It is found

Problem	mDelay	First Solution Time	First Solution Delay	Best Solution Time	Best Solution Delay	CPU	%	Back-tracks
20	1080	36.79	3010	74.91	1524	88.44	84.70	420
21	196	14.36	580	16.52	375	22.30	74.08	95
22	184	62.75	206	62.75	206	69.56	90.21	27
23	186	53.76	425	55.41	339	62.81	88.22	113
24	205	12.84	842	22.66	448	35.25	64.28	231
25	271	0.97	591	6.27	305	9.66	64.91	132
26	120	14.42	629	18.40	390	22.25	82.70	113
27	168	0.91	320	1.49	212	5.66	26.33	32
28	150	2.47	691	4.65	348	9.35	49.73	96
29	200	13.19	589	16.11	357	21.41	75.25	120
40	198	26.01	381	39.25	329	52.73	74.44	195
41	245	44.63	1387	80.26	766	233.42	34.38	1944
42	307	52.62	871	55.97	527	78.12	71.65	291
43	298	251.73	766	251.73	766	374.32	67.25	1423
44	312	39.67	950	110.79	562	129.68	85.43	1032
45	215	282.73	447	284.66	358	301.32	94.47	117
46	311	74.88	1495	179.78	801	264.51	67.97	2110
50	625	9663.83	8195	–	1015	–	–	–
51	615	6690.44	5915	–	(1055)	–	–	–
60	296	–	–	–	–	–	–	–
61	325	–	–	–	–	–	–	–

Table 5.3: Results of solving 21 problems, the minimum estimated delay, the time to find the first solution, to find the best solution and the comparison between this time and the necessary time to prove a solution is optimal, all in seconds, and the total number of backtracks.

by considering each conflict in isolation and finding the minimum delay necessary to resolve it, ignoring the effect on other trains. Hence, this value represents in practice an estimation of the lower bound in the total delay for the problem.

The *First Solution* columns show the time to find the first feasible solution to a problem and its respective delay. Likewise, the *Best Solution* columns show the time to reach the best solution for that particular problem and the overall minimum delay (in brackets instead when not proved optimal).

In the *CPU* column is shown the total time in seconds to find the best solution and prove its optimality. In the % column is the time spent on finding the optimal solution (but not proving optimality) as a percentage of the total processing time.

Finally, the *Backtracks* column shows the number of backtracks during the process of finding solutions and proving optimality.

The algorithm proposed in Section 5.2.2, along with the overall delay D described in Section 5.2, which is used to constrain the problem even more each time a solution is found, is capable of solving to optimality most of the problems in Table 5.2.

Problems 50–61 are exceptions, because it was not possible to reach their optimal solutions within the maximum time of 15 hours stipulated for these experiments. In case of problems 60 and 61, the situation was even worse as the respective first solutions were not found within this time.

The respective better solutions for problems 50 and 51 were found only by re-running the program many times with arbitrary decreasing values for D, until reaching those solutions presented in Table 5.3. The best solution of problem 51 presented in Table 5.3, was shown not to be the optimal solution after solving this problem using an improved algorithm described in Section 5.2.4, when the optimal solution was found.

In some cases, such as in problem 22 and 43, the optimal solutions were found at once. In problem 43, backtracking to prove optimality was responsible for about 30% of the total processing time, whereas in the former only 10%. The high number of backtracks in problem 43 shows that once a solution is found, the constraint restricting the total delay is not tight enough in this problem to prevent the algorithm from making unfruitful decisions.

The algorithm manages, most of the time, to find a first solution with less than twice the delay of the optimal solution. The exceptions are problems 50 and 51, when the ratios go up to about 2.40 and 3.60, respectively. This suggests that the possibility of wrong decisions in the early stages of the search grows with the size of problem, causing this distance between the first and the best solution.

The factor of about 2 between the first solution found and the optimal solution is still high. The reason for this distance is that, although the strategy of eliminating delays by shifting in time the conflicting task which causes the minimum delay when not scheduled first, the process is blind as it does not take into account the global influence of each individual decision. In addition, because of the model used in this chapter, where each task (piece of trip) on a job (a trip) must be performed immediately after its predecessor ends, the entire trip is shifted when a single task is delayed. Hence, it is only by backtracking to the point where the wrong decision was made that a different order of trips is tried and, therefore, better solutions are eventually found. The backtracking process can take a very long time should the wrong decision be made at the outset.

In many cases, after finding the first solution the program found the best solution easily. In problems 25 and 44, however, the first solutions were found in about 15% and 35%, respectively, of the necessary time to find the best solution.

Likewise, the necessary running times to reach the best solution and a near-optimal solution for the problems 50 and 51, respectively, are more than four times the time to find the first solutions for these problems. For the other problems in Table 5.2, the time to find the first solution is more than 50% of the time to find the final solution.

The number of trains and passing points more than doubled for problems 50–61 in comparison with the previous problems; so did the number of conflicts to resolve in these problems. In other words, the size of these problems has greatly increased. The program could not find the optimal solution for these problems after running over 15 hours.

Because of the difficulty of finding the optimal solutions for problems 50 and 51, a *binary search* technique is used over the interval of possible values for D in order to speed up the search for solutions which approach incrementally the optimal solution.

Initially, an interval I of allowed total delay is set, where $I = [mDelay..horizon]$, and the value $horizon$ is calculated as described in Section 5.2.2. The value of $horizon$ is then large enough to accommodate all trips in sequence. The value of mDelay is used as a lower bound for the overall delay.

Firstly, an initial solution is found which gives a value to D, the overall delay so far. The interval I is then updated so that $I = [mDelay..D]$.

Subsequently, the program tries to find a better solution where the new overall delay $D' < (mDelay + D)$ **div** 2. In case another solution is found, the interval I is once more updated, $I = [mDelay..D']$; otherwise, the strategy is changed in order to find solutions sequentially from the last feasible solution found. This is so that time is not wasted searching for solutions in an interval that is unlikely to have a solution.

When using the binary search described above, and letting the program run for up to 15 hours, it is possible to reach much better solutions than before, within this limit of time. For instance, the program could reach solutions with delays of 1515 and 1285 for problems 50 and 51, respectively.

The binary search did improve the performance on finding better solutions. However, it is still difficult to reach the optimal solution and prove optimality for problems 50 and 51 within a maximum of 15 hours. The optimal solution for problem 50 was proved only after 29 hours using this method.

5.2.4 An Improved Algorithm for the 1^{st} Model

The ordering approach of the first algorithm yielded a great number of decision points in the search tree. As described in Section 5.2.2, this process orders each task on each resource up to the point the target conflict is resolved, therefore, even those activities which are not conflicting with any other need to have a fixed order on the resource. The use of the *edge-finder* inference reduced many fruitless possible ordering choices. However, this was not enough to lead the algorithm to reach their best solutions within 15 hours in the case of larger problems such as problems 50–61.

Unlike the first algorithm approach, this second alternative approach simplifies the way of resolving conflicting tasks. Still a chronological way of resolving conflicts, the idea in this approach consists of merely adding a precedence constraint between the conflicting activities, according to the dispatch rule adopted, instead of ordering every previous task on the resource where the earliest conflict occurs. In this case, the *edge-finder* information is not used during this process.

The structure of this second Algorithm 5.2.2 is quite similar to the first algorithm depicted in Section 5.2.2. The difference is in line 9.

Algorithm 5.2.2

1 Schedule all conflicting set of tasks
2 **begin**
3   **while** $\exists r_i \mid conflicts(r_i, t) \neq 0, t \in [0..horizon]$ **do**
4   pick a r_i in $\{r_i \in \mathcal{R} \mid conflicts(r_i, t)$ is the earliest$\}$
6   **for** the earliest pair of conflicting tasks on r_i **do**
7   select a task by applying the `Shortest Processing Time`
9   add a precedence constraint on them
11   **if** the selected task cannot be scheduled first
12   **then** `backtrack` choosing the other task
13   **fi**
14   **od**
16   **od**
18 **end**

5.2.5 Solving 21 Real-Life Problems using the 2^{nd} Algorithm

The same datasets used to perform the previous experiments presented in Section 5.2.3 are also used to evaluate the performance of this second algorithm against the previous approach. Therefore, the problems are still as described in Table 5.2, and the objective function is to minimize is the overall delay as formulated in Section 5.2.

This algorithm, as stated above, differs from the first approach at the point that a precedence constraint is posted between the conflicting tasks to resolve a conflict. When this constraint is added in, the propagation reduces the domain of the start time of the conflicting tasks. Note that resolving a conflict can also create a new conflict on the nearby tasks as a result of delaying one of the conflicting tasks.

This new approach runs much faster than the previous one, which allows us to find an initial solution for the two largest problems 60–61 within the limit time of 15 hours. However, the quality of initial solutions found have been worsened in most of the cases.

		First Solution		Best Solution				Back-
Problem	mDelay	Time	Delay	Time	Delay	CPU	%	tracks
20	1080	0.09	5068	2.21	1524	2.44	90.57	1024
21	196	0.17	723	0.28	375	0.33	84.84	84
22	184	0.44	1079	0.85	206	0.88	96.59	203
23	186	0.22	1093	0.32	339	0.39	82.05	71
24	205	0.14	1767	1.83	448	1.92	95.31	879
25	271	0.08	1436	1.31	305	1.34	97.76	581
26	120	0.08	629	0.15	390	0.20	75.00	61
27	168	0.05	681	0.12	212	0.14	85.71	40
28	150	0.03	691	0.06	348	0.12	50.00	41
29	200	0.06	1532	0.74	357	0.78	94.87	262
40	198	0.15	968	0.70	329	0.78	89.74	246
41	245	0.18	766	0.18	766	2.02	8.91	729
42	307	0.35	2250	8.72	527	9.03	96.56	3609
43	298	0.31	1007	0.46	766	2.00	23.00	886
44	312	0.20	950	1.59	562	1.83	86.88	768
45	215	0.11	1028	2.50	358	2.61	95.78	792
46	311	0.29	2324	7.42	801	8.69	85.38	3598
50	625	14.69	6275	29170.90	1015	29581.70	98.61	2514837
51	615	7.86	2755	2081.09	970	2433.04	85.53	276030
60	296	1532.07	6509	–	–	–	–	–
61	325	5188.27	8402	–	–	–	–	–

Table 5.4: Results of solving 21 problems using the 2^{nd} algorithm approach, the minimum estimated delay, the time to find an initial solution, to find the best solution and the comparison between this time and the necessary time to prove a solution is optimal, all in seconds, and the total number of backtracks.

The quality of the initial solutions in this new approach is worse than the previous approach because the decision on resolving conflicting tasks is based now only on the SPT dispatch rule adopted, whereas the edge-finding inference was also taken into account in the first approach.

The time to reach good solutions, however, for the problems has noticeably improved. Nevertheless, the time to find initial solutions for the two largest problems is still too long. We attribute the long time to the process of finding the earliest conflict to the function $conflicts(r_i, t)$ in the algorithm. This function is called every time before resolving a conflict, as presented in Algorithm 5.2.2.

In order to find the conflicts on a particular resource, and thereafter the earliest, this function compares each time interval an activity can be performed against every other activity. Therefore, the complexity of this function is $O(n^2)$, where n is the number of tasks requiring the resource. Hence the entire process of finding the earliest conflict over all the resources is of complexity $O(mn^2)$, where m is the number of resources.

In the next section we present three heuristics to further speed up the search for improved solutions to the problem. The idea behind each of these heuristics is to set a limited region in which to search for solutions. Moreover, these three heuristics are devised in such a way that, when used appropriately, the optimal solution will always be found. However, they could be used independently, if finding the optimal solution is not the aim.

5.2.6 Preprocessing Heuristics

The original problem tackled in this thesis is that of minimizing the total tardiness (total delay) of the single-track railway scheduling problem. Let $\mathcal{T} = \{T_1, T_2, \ldots, T_{|J|}\}$ be the set of tardiness variables for each job (trip) $J_i \in \mathcal{J}$, where $\mathcal{J}$ is the set of jobs. Hence the objective is to minimize $\sum_{T_i \in T} T_i$, as stated in Section 5.2.

Although the objective is to minimize total tardiness, let us for now consider the maximum tardiness amongst the individual trips. Consider that $T_{max} \in \mathcal{T}$ is such that $T_{max} = \max\{T_1, T_2, \ldots, T_{|J|}\}$, where $[\min(T_{max}), \max(T_{max})]$ is the domain of T_{max}, an interval variable. Consider further that $\mathcal{S}ol = \langle Sol_1, Sol_2, \ldots, Sol_n = Sol_* \rangle$ is a sequence of solutions with decreasing total tardiness which might be found for a given single-track railway scheduling problem. Sol_1 is the first solution, and $\text{Cost}(Sol_*)$ is the cost of the optimal solution (minimum total tardiness) for the problem. t_i of solution $Sol_i \in \mathcal{S}ol$ is the value assumed by the variable T_{max} in this solution.

It can be expected that solutions with small T_{max} will have small total delay. However, it is not necessarily true that the solution with the minimum T_{max} will be the solution with the minimum total delay. Therefore, we will see later in this section, that minimizing T_{max} does not necessarily give us the optimal solution for the problem being tackled in this thesis.

Although the process of minimizing T_{max} does not necessarily give us the optimal solution, it can give us good bounds for the total tardiness. Furthermore, more propagation can occur when bounding the variable T_{max}, as this is immediately propagated to all the T_i variables, whereas the propagation due to bounding the total tardiness is much weaker, and often a tighter bound on the total tardiness will not lead to tighter bounds on the individual T_i variables.

In the light of that, three heuristics are proposed which use the T_{max} variable in a preprocessing stage to tackle the problem. Each time a solution is found by any of these heuristics, it gives a new upper-bound on the total tardiness.

The first heuristic only aims at speeding up the process of finding an initial solution. This heuristic transforms the original problem into another problem whose objective is to find a solution for the problem given an initial upper-bound for T_{max}.

The second heuristic is an extension of the first heuristic. In this second heuristic, the aim is to search for solutions with decreasing upper-bounds on T_{max}. Thus, a binary search procedure on the interval of the domain of variable T_{max} is applied to impose increasingly tight bounds on T_{max}, and reach a minimum value T'_{max}. At the same time, we are imposing tighter bounds on the total tardiness whenever a new solution is found, so T'_{max} may not be the minimum.

The third preprocessing heuristic is a complementary procedure to the second heuristic. Once the minimum upper-bound value for T_{max}, T'_{max}, is found by the second heuristic, we know thus that there will not be any other solution with smaller T_{max} and smaller total tardiness. Therefore, the third heuristic searches within the search space where $T'_{max} \leq T_{max} < \textit{current total tardiness}$, by progressively increasing lower-bounds on T_{max}.

Using these heuristics we are optimizing two objectives simultaneously: while searching by one of the above heuristics we are simultaneously adding tighter bounds to the total tardiness.

The following sections further detail each of these heuristics, and how these heuristics can also be used to increasingly narrow the search space. Because the way these heuristics are used here covers the entire search space where the optimal solution might be, the result on finding optimal solutions is also presented.

5.2.6.1 Improving on Searching for a 1^{st}-Solution

The aim of this first heuristic is only to find a good initial solution. Therefore, we start the process by imposing an initial bound on T_{max} using the value of $mDelay$. This choice of $mDelay$ is because it is an approximate lower-bound for the total tardiness problem (see Section 5.2.3 pp. 72). Hence, it is likely that T_{max} will be smaller than $mDelay$.

The tighter the upper-bound on T_{max} is, the earlier the algorithm can backtrack due to a wrong decision made during the search. Therefore, in case a problem has no solution with $T_{max} < mDelay$, the initial upper-bound, the heuristic adds the constraint $mDelay \leq T_{max}$ to the problem instead. In other words, the upper-bound becomes a lower-bound for T_{max} when a solution is not found with the T_{max}'s initial bound. An increased upper-bound is imposed on T_{max} until the first solution is found. The bound increase used in this heuristic is arbitrarily chosen as 20% over the previous unsuccessful bound.

Note that when no solution is found when imposing the initial bound on T_{max}, this heuristic actually narrows the search space of the problem by adding lower-bounds for T_{max} until a solution is found.

5.2.6.2 Minimizing the Maximum Tardiness Objective

This heuristic takes the problem as a maximum tardiness minimization problem rather than a total tardiness minimization problem. The choice to minimize the maximum tardiness is because we can have more propagation when imposing an upper-bound on the T_{max} variable than on a sum of variables, as is the case in total tardiness objective.

Moreover, this second preprocessing heuristic is an extension of the first heuristic. This is because rather than just searching for only one solution, this second heuristic seeks better and better solutions imposing increasingly tight upper-bounds on the T_{max} variable. Each time a solution is found by the minimization process above, the corresponding total tardiness solution cost is gathered and posted as a new upper-bound on the total tardiness solution. This second objective may prevent the algorithm from reaching the optimal minimum value for T_{max}, as previously stated.

Because this heuristic is devised with the aim of just narrowing the search space, it can start in either of two ways. The first is from scratch, where no upper-bound is given to the total tardiness, nor to the T_{max} variable. The second way is when either or both bounds are initially given.

However, in the following experiments we are interested in finding the optimal total tardiness solution for the problem. Thus, in order to cover the entire search space where the optimal total tardiness solution can be found, in the experiments carried out with these heuristics, we use the total tardiness in the solution found by the first heuristic as the initial upper-bound for the total tardiness within this second heuristic. In addition, we constrain T_{max} to be smaller than the last T_{max} found in the first solution.

The minimization of T_{max} in this heuristic is performed by a binary search on its limits. Initially, an interval I of allowed T_{max} delay is set, where $I = [L..U]$, L is initially made 0, and the value U is the initial bound on T_{max} passed to the heuristic. A new upper-bound U' calculated by $U' = (L + U)$ **div** 2 is then imposed on T_{max}.

With this new U' upper-bound applied on T_{max}, a new solution is sought for the problem. If a solution is found with this new upper-bound on T_{max}, U is made equal to U', and $L = U'$ otherwise. Thereafter, the process is repeated until $L \geq U$, when then we will have found the minimum value T'_{max} for T_{max}. At this point we know that there is no other solution with smaller T_{max} and smaller total tardiness.

An interesting fact, that the experiments will show, is that the t_is are not necessarily monotonically increasing (or monotonically decreasing) as the cost of the solutions Sol_i, Cost(Sol_i), decreases. Therefore, when this heuristic terminates we may not have found the optimal solution for the total tardiness. However, it is faster at finding good solutions than directly minimizing total tardiness.

In the next section we describe the third heuristic, which is devised to search for solutions with $T'_{max} \leq T_{max}$, if any exist.

5.2.6.3 Maximizing the Maximum Tardiness Objective

The previous heuristic worked with the aim of finding good bounds for the total tardiness objective while minimizing T_{max}. This third heuristic, on the other hand, aims at finding all the improved solutions for the total tardiness while imposing increasing lower-bounds on the T_{max} variable. Like in the second heuristic, in this heuristic each time a solution is found during the search process, it gives a new upper-bound on the total tardiness.

This third heuristic receives as parameters an initial upper-bound for the total tardiness, and an initial lower-bound for the T_{max} variable.

In the experiments carried out with this heuristic here, we will give the final total tardiness, yielded by the second heuristic, as an initial upper-bound on the next total tardiness. The minimum upper-bound, T'_{max}, found by the second heuristic is posted as a constraint $T'_{max} \leq T_{max}$ from the outset in this heuristic. The choice of T'_{max} as an initial lower-bound for T_{max} is because we know from the second heuristic there is no other solution where $T_{max} < T'_{max}$ and smaller total tardiness. However, we may still find improved solutions, where $T'_{max} \leq T_{max}$, unreached by the second heuristic.

The searching process within this third heuristic is thus rather different from that used in the second heuristic. The process in this heuristic consists of adding increasing lower-bounds on T_{max} towards finding its maximum limit. While doing this, we intend to find still improved solutions for the total tardiness objective. Note again that the new upper-bounds imposed on the total tardiness each time a solution is found will prevent the search from actually finding the maximum for T_{max}.

Therefore, having the initial bounds: $T'_{max} \leq T_{max} <$ *current total tardiness*, a solution is sought for the problem regarding the constraints currently posted to the problem. Each time a solution $Sol_i \in \mathcal{S}ol$ is found, the t_i of that solution is posted as a new lower-bound for the next T_{max}. This new lower-bound t_i is also added to a list of applied lower-bounds. New solutions are sought until no further solution can be found.

Recall that $\mathcal{S}ol$ is a sequence of solutions with decreasing total tardiness (total delay). In addition, as previously stated, the t_i values are not necessarily monotonically increasing (or monotonically decreasing) as the cost of the solutions Sol_i decreases. Hence, in maximizing T_{max} we may sequentially find solutions Sol_i and Sol_k, but skip solution Sol_j, because $t_i < t_j < t_k$, even though $\text{Cost}(Sol_j) < \text{Cost}(Sol_k) < \text{Cost}(Sol_i)$. Nevertheless, when no further solution can be found by the process just described above, we know that there is no other solution with both larger T_{max} and smaller total tardiness.

In order not to lose any improved solution, whenever a solution cannot be found, the latest list's lower-bound on T_{max} is deleted from it and posted as an upper-bound constraint on T_{max} instead. Then all the improved solutions are sought within the search space delimited by the latest list's lower-bound, now posted as an upper-bound on T_{max}, and its preceding lower-bound. This process continues until the list of applied lower-bounds is emptied. In doing this, if an improved solution is skipped when applying the increasing lower-bounds on T_{max}, it will be found when using the list's elements as upper-bounds.

To illustrate the above explanation let us consider, for instance, that this third heuristic finds the solution Sol_5 (Figure 5.7), where Cost$(Sol_5) = 2993$, and $t_5 = 765$. In the sequence, this heuristic imposes t_5 as the new lower-bound for T_{max} and adds t_5 to the list of applied lower-bounds. A new solution is then sought with this new lower-bound.

The next solution found can be solution Sol_7, where Cost$(Sol_7) = 2922$, and $t_7 = 856$. Again, t_7 is imposed as the new lower-bound for T_{max} and added to the list of applied lower-bounds. Note that, in this case, solution Sol_{11} has been skipped, even though $t_{11} = 819$ and Cost$(Sol_{11}) = 2295$.

The heuristic finds next solution Sol_8, where Cost$(Sol_8) = 2875$ and $t_8 = 1082$. However, no further solution can be found by increasing the lower-bound on T_{max}.

Note that, in Figure 5.7, some improved solutions were skipped by this process, including the optimal solution: Cost$(Sol_{17}) = 1524$, $t_{17} = 876$. The second part of the heuristic, however, overcomes this problem by imposing sequentially each of the values on the list of applied lower-bounds as upper-bound and searching for further solutions.

The lastest t_i (t_8) included in the list of applied lower-bounds is deleted from this list and applied now as an upper-bound on T_{max}. In doing this, the heuristic searches now for all the improved solutions with T_{max} within the range of $t_7 < T_{max} \leq t_8$. Through this process, solution Sol_9 is found, where Cost$(Sol_9) = 2326$, and $t_9 = 1063$. t_9 is thus added to the list of applied lower-bounds. The heuristic searches now for all the improved solutions with T_{max} within the range of $t_9 < T_{max} \leq t_8$.

No further solution can be found within the latest T_{max} interval. Therefore, t_9 is deleted from the list and now applied as the new upper-bound on T_{max}, and the heuristic searches now for all the improved solutions with T_{max} within the range of $t_7 < T_{max} \leq t_9$. When searching within this interval the optimal solution (Sol_{17}) is then found (Figure 5.7).

5.2.6.4 Applying the Preprocessing Heuristics

Before showing the results on solving the set of 21 problems (see Section 5.2.3) by using the preprocessing heuristics, we shall first have a close look at the t_i values for a set of decreasing total tardiness solutions for problem 20.

Figure 5.7 shows the value of t_i for each of the solutions found for problem 20 when the search is done only by the Algorithm 5.2.2 described in Section 5.2.4 pp. 76. On the most left side of this Figure, in the first solution for the problem with total

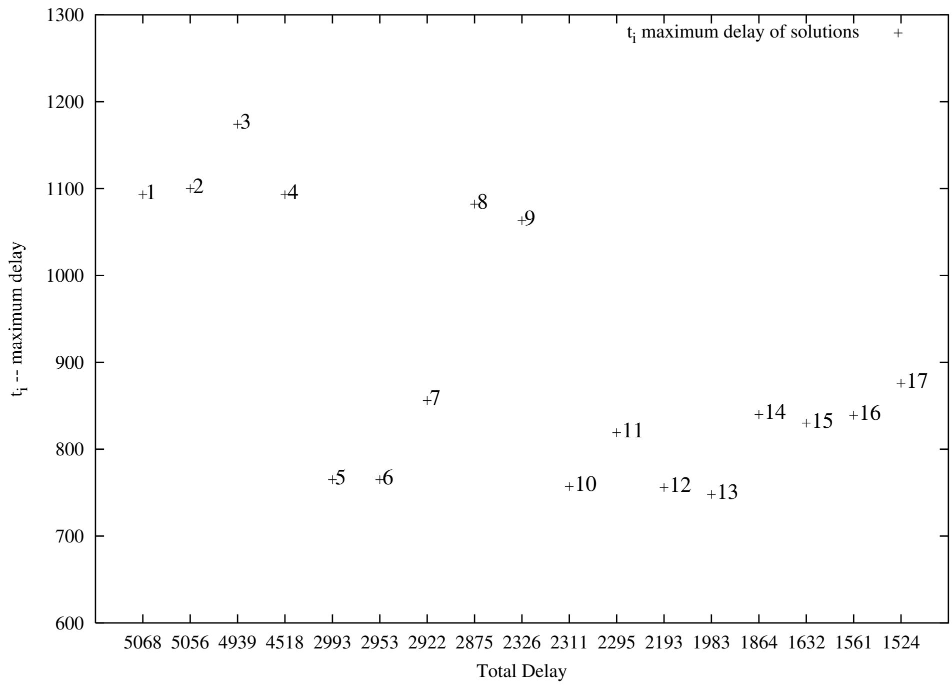


Figure 5.7: Progress of the Total Delay Solutions found for problem 20 and their respective t_i values.

tardiness of 5068, the most delayed job has t_1 of 1093, which is greater than *mDelay* (see Section 5.2.6.1). The second solution has total tardiness 5056 and t_2 is 1100.

On the rightmost side of the Figure, the optimal solution for the problem has total tardiness 1524 and t_{17} is 870. The results confirm that simply minimizing T_{max} will not necessary yield minimum total tardiness.

The results of applying the three preprocessing heuristics in sequence to solve the 21 problems are presented in Table 5.5.

In Table 5.5, *Minimizing* T_{max} column shows the result yielded by the heuristic which minimizes T_{max} while searching for improved total tardiness solutions, as described in Section 5.2.6.2. The time column shows only the time from the start of the heuristic up to the time no other solution could be found by the heuristic.

A dashed line in the *Delay* columns shows when the heuristics failed to find an improved solution.

Likewise, column *Maximizing* T_{max} shows the result of the heuristic which maximizes T_{max}, as described in Section 5.2.6.3. Again, the time column shows only the time from the start of the heuristic up to the time no other solution could be found by the heuristic.

CPU column shows the total time from the start of the first heuristic to the

Problem	mDelay	First Solution		Minimizing T_{max}		Maximizing T_{max}		CPU
		Time	Delay	Time	Delay	Time	Delay	
20	1080	0.06	2993	0.21	2154	0.67	1524	0.94
21	196	0.01	723	0.07	508	0.13	375	0.21
22	184	0.02	206	0.01	–	0.00	–	0.03
23	186	0.03	427	0.08	339	0.03	–	0.14
24	205	0.03	627	0.09	448	0.04	–	0.16
25	271	0.01	1173	0.04	305	0.02	–	0.07
26	120	0.02	474	0.00	–	0.12	390	0.14
27	168	0.02	320	0.04	212	0.01	–	0.07
28	150	0.02	441	0.00	–	0.05	348	0.07
29	200	0.00	633	0.08	357	0.02	–	0.10
40	198	0.03	357	0.10	329	0.04	–	0.17
41	245	0.03	766	0.12	–	0.92	–	1.07
42	307	0.05	1514	0.27	527	0.16	–	0.48
43	298	0.03	1007	0.38	766	0.78	–	1.19
44	312	0.03	950	0.16	571	0.21	562	0.40
45	215	0.02	993	0.07	408	0.14	358	0.24
46	311	0.04	1574	0.19	912	1.53	801	1.76
50	625	0.60	6275	1.38	1345	1714.36	1015	1716.34
51	615	0.34	2755	1.40	1110	525.32	970	527.06
60	296	2286.65	4502	51714.23	1884	0.70	–	54001.58
61	325	651.18	5767	53350.98	5756	2.32	–	54004.48

Table 5.5: Results of solving 21 problems using the preprocessing heuristics. The times (in seconds) and the cost found for each of the heuristics are given. One heuristic finds an initial solution, one minimizes the T_{max} while searching for improved solutions, and another maximizes the T_{max}.

completion of the third heuristic. As in previous experiments, we have set a time limit of 15 hours for each problem.

The results presented in Table 5.5 are greatly improved both in terms of time performance and quality of the solutions, when compared to those presented in Table 5.4. The optimal solution for problem 50 is now reached in less than 2 minutes, but proved to be the optimal in less than 30 minutes. The solution found when minimizing T_{max} for problem 51 is very near to the optimal solution and it is found in less than 2 seconds.

Although the time to find an initial solution for problem 60 has increased, its quality did improve with the use of the first heuristic described in Section 5.2.6.1.

The result with problem 61 is disappointing, as the second heuristic took most of the time to produce just a little improvement from 5767 to 5756 in the cost of the solutions. However, this is an improvement in time and quality performance over the previous algorithm results, where a worse solution of cost 8402 was only found after nearly 15 hours.

Although they are heuristics, we devised them such that if appropriately used together, as it is the case here, the optimal solution will always be found.

5.3 Second Model

Although the first model is applicable to scheduling passenger trains in practice, the second model is a more realistic approach for the train scheduling problem in general. The difference between these two models is basically that in the first model the delay can only occur at the departure of a trip. On the other hand, in the second model, a train can be delayed at any point in the course of its trip to resolve a conflict on a track segment.

In this second model, the constraint which allows no delay between successive tasks in the first model is relaxed. Hence, expression 5.1 in Section 5.2 is replaced by the following expression:

$$\overline{d}_{i(j+1)} \geq \overline{d}_{ij} + p_{ij}, \forall j \in \{1, 2, \ldots, k = \mid J_i \mid\}. \tag{5.4}$$

An example is given in Section 5.2 to illustrate the elements described in the first model as well as the algorithms proposed for the first model. The given desired timetable was also presented using the Time-space Diagram in Figure 5.3 and the optimal solution for that problem when applying the first model is that presented in Figure 5.6. When the second model is applied, there is no need for re-timing a trip at its departure time to resolve a conflict (see expression 5.4): the conflicting operations involved are ordered at the point where the conflict occurred, without affecting the previous stretch of the trip. This new solution is as shown in Figure 5.8, an optimal solution for the problem previously presented in Figure 5.3.

The first strategy adopted (Section 5.2.2, pp. 64) for solving the single-line track railway scheduling problem has the drawback of dealing with very many choice points when deciding the order for each task on each resource. The result of dealing with many alternative orderings is that only the smallest problems could be solved, as discussed in Section 5.2.3. In order to overcome this difficulty, a procedure consisting of just adding a precedence constraint between conflicting tasks is employed in the second algorithm (Section 5.2.4, pp. 76). This way it is not necessary to set up an

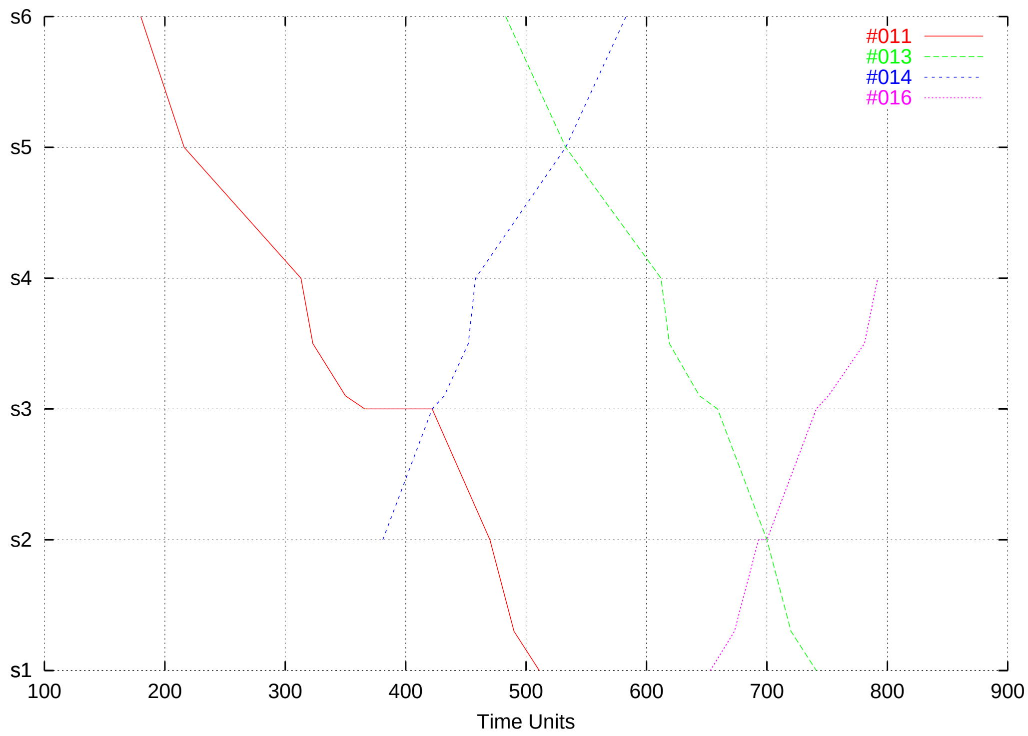


Figure 5.8: The optimal solution for the timetable given in Figure 5.3 when applying the second model.

order to those operations which do not break any constraint. This second algorithm is used as that basis of a complete search method for the remaining part of this thesis.

Using the problem's second model as a basis, we present a Disjunctive Graph formulation for the problem. Such a formulation is also often used for the job-shop scheduling problem in the literature. The new formulation for the problem is presented in the following section.

5.3.1 A Disjunctive Graph Model

The job-shop scheduling problem is a class of combinatorial problem well known in OR and, as discussed in Chapter 3, usually defined as follows. Given are a set of jobs J. For each job $J_i \in J$ a set of operations $J_i = \{o_{i1}, o_{i2}, \ldots, o_{ik}\}$ is also specified. Each of these operations requires processing on a unique machine $r_i \in \mathcal{R}$, the set of machines. The processing time p_{ij} for each operation o_{ij} is also given as input for the problem. In addition, each job has its release date d_i and its expected completion date c_i. If C_i denotes the actual completion time for job J_i, then the tardiness T_i of job J_i is defined as $max(C_i - c_i, 0)$. The aim may be, for instance, to minimize the total tardiness given by the expression:

$$D = \sum_{1 \leq i \leq n} T_i \tag{5.5}$$

i.e. D is the overall delay of jobs.

This problem can elegantly be represented by a disjunctive graph [71]. Consider a directed graph $G = (N, A, B)$, where N is the set of all the processing operations o_{ij} to be performed of each J_i job of J. A is a set of *conjunctive* arcs which represent the sequence of operations on a particular job, whereas B is the set of pairs of *disjunctive* arcs linking tasks being performed on the same machine.

The conjunctive arcs of A are in fact precedence constraints so that, for each pair of successive operations $(o_{ij}, o_{i(j+1)})$ of job J_i, o_{ij} must be performed before $o_{i(j+1)}$, but not necessarily immediately before it, as in the first model described in Section 5.2, pp. 60. Therefore, the release time of consecutive operations in this formulation is as already pointed out by the expression 5.4.

Each machine can only perform a single operation at a time, therefore, any two operations which are performed on the same machine are linked by a pair of directed arcs in the set B. The semantic of this pair of *disjunctive* arcs is so that one arc is to set the possible precedence order between two pair of tasks as $(o_{ij} \rightarrow o_{i'j'})$, and another arc is to set the alternative order $(o_{i'j'} \rightarrow o_{ij})$. The two alternative orders are set this way until an order between the tasks on the machine is determined.

The conjunctive arcs $(o_{ij}, o_{i(j+1)}) \in A$ linking two consecutive operations of job J_i have associated to them the processing time p_{ij}, that is the processing time of task o_{ij}. Associated to the arc linking two disjunctive operations, for instance $(o_{ij}, o_{i'j'})$, is p_{ij} and associated to the alternative arc linking the same pair of task $(o_{i'j'}, o_{ij})$ is $p_{i'j'}$, in case $o_{i'j'}$ is instead performed before o_{ij} on the machine.

To the graph G we add a dummy node S as the source node. This node represents a dummy task and links to the first task of each job. The processing time associated to the arc between these two tasks is r_i, the release time of job J_i.

Besides the set of conjunctive and disjunctive arcs which exist due to the characteristics of the problem, we may also have in the graph G some other fixed additional arcs introduced by the special constraints described in Chapter 2.

In the graph G we have $n = \mid J \mid$ dummy nodes N_i representing the sink nodes. Each of these nodes denote the actual completion time of job J_i. The reason for having n sink nodes rather than one is because we are minimizing the total tardiness instead of a unique maximum completion time (maximum tardiness), as is usual when the objective function is the *makespan*. Moreover, a change in the order of an operation on a machine can also cause a change in completion time of the other jobs.

In order to illustrate the model we are proposing here, consider a desired timetable with four trains as presented in Figure 5.3. The single-line railway considered is that shown in Figure 5.2 and the characteristics are as described in Section 5.2.1. The same planned four services are to be offered. The services are presented in Table 5.6.

Trip ≡ Job	Machines' Sequence	d_i	p_{ij}
11	$r_8, r_7, r_6, r_5, r_4, r_3, r_2, r_1$	180	36,97,10,27,16,48,20,21
13	$r_8, r_7, r_6, r_5, r_4, r_3, r_2, r_1$	483	50,79, 7,25,15,41,20,21
14	$r_3, r_4, r_5, r_6, r_7, r_8$	381	41,10,20, 6,73,52
16	$r_1, r_2, r_3, r_4, r_5, r_6$	653	20,20,41,10,30,11

Table 5.6: A given desired timetable (reproduced from Table 5.1).

In the example shown in Figure 5.3, we have trip $J_{11} = \{o_{11,1}, o_{11,2}, o_{11,3} \ldots o_{11,7}, o_{11,8}\}$ (see Table 5.6). Each operation $o_{11,i}$ signifies part of the trip J_{11} consisting of either travelling from one point to another along the track, or a dwell time at a passing point. Because the operations $o_{11,i}$ are performed consecutively (but not necessarily immediately after one another as is the case in the first model), a conjunctive arc is set between each pair of them. This way, the conjunctive precedence constraints are such that $o_{11,1} \rightarrow o_{11,2}$, $o_{11,2} \rightarrow o_{11,3}$, and so on.

The disjunctive arcs are placed between tasks which are performed on the same machine. In this example each track segment $r_i \in \mathcal{R}$ correponds to a machine or a resource in job-shop scheduling problem. For instance, for the operations performed on the track segment between station s_2 and s_3, we would have disjunctive arcs as depicted in Figure 5.9.

The example in Table 5.6 is an infeasible schedule because the operations ($o_{11,6}$, $o_{14,1}$) $\in B$ of G, for instance, do not have their precedence order determined yet, in other words, their unique execution time interval has not been yet determined. Hence, G is a cyclic graph, as is shown by the subgraph of G in Figure 5.9.

In order to have a feasible schedule (see Section 3.3, pp. 17) it is necessary to transform the given disjunctive graph of the given timetable into an acyclic graph. This is performed by choosing an order for those activities linked by disjunctive arcs.

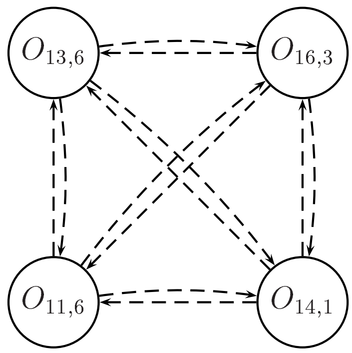


Figure 5.9: Disjunctive subgraph of G for operations being performed on the track section between station s_2 and s_3.

This means, in the train context, to choose which service is going to use first the track segment in dispute.

Any operation o_{ij} whose execution time coincides, either in total or in part, with another operation being performed on the same track segment, is considered a conflicting task. This is the case with ($o_{11,6}$, $o_{14,1}$) and ($o_{13,6}$, $o_{16,3}$), operations being performed on the track segment between stations s_2 and s_3 in Figure 5.3. To arbitrate an order between any conflicting pair of operations we need to use a dispatch rule to decide which trip uses that track segment first. This way only one of the disjunctive arcs is left in the disjunctive graph. The consequence of sequencing a conflicting pair of tasks is that one of the tasks is delayed from its planned departure time. However, the order of unconflicting activities is preserved because they do not have any impact on the total delay (see Equation 5.2) we want to minimize.

Whatever the criteria chosen to arbitrate the order for the conflicting activities, once the unique time interval for execution of each operation in the graph is determined, some operations will have an important role in the graph structure: they will be the *critical operations*. The following section discusses the structure of the critical path for this problem and its relevance during the search process.

5.3.2 Critical Path Structure

The *critical path* is formed by particular operations. Given a feasible solution, the activities whose schedule order cannot be changed without also changing the value of the objective function [71] are those of the *critical path*. This criterion can also be applied when the total tardiness is the objective function, as is the case in this paper.

During the process of scheduling the conflicting activities $(o_{ij}, o_{i'j'})$, one of them will have to be shifted in time. This is done by adding a precedence constraint between them $(o_{ij} \rightarrow o_{i'j'})$. The delayed activity $(o_{i'j'})$ is scheduled after that activity chosen to be scheduled first (o_{ij}). In this case these activities may have an *adjacent* schedule order in the solution. Although this situation of *adjacency* can be changed if a successor activity of the same job of the delayed activity is delayed later on in the course of the algorithm execution, we will keep track of previous adjacent operations. The reason is because they may have contributed to the subsequent conflicts. Hence, the set of all adjacent operations will constitute the critical path and the arcs between two adjacent activities are the *critical arcs*.

The algorithm to schedule these activities is thus based upon choosing appropriate *critical arcs* for the conflicting activities so that a feasible schedule is produced while minimizing the overall delay. The algorithm structure is described in the following two sections. We start by describing the neighbourhood structure we use for the local search part of the algorithm, which is based on the critical arcs of a given feasible solution. In the sequence, the complete three-stage structure of the algorithm is described.

5.3.3 Neighbourhood Structure

There are many neighbourhood definitions available for the job-shop scheduling problem in the literature, as discussed in Section 3.5.2.1, pp. 25. However, few references can be found with the total tardiness as the objective function [81, 82].

The neighbourhood proposed in this paper is based on the critical arc structure described in Section 5.3.2 for a problem whose objective function is to minimize the total tardiness of the jobs. This neighbourhood was also used by Kreipl in [57]. As in Kreipl's works the number of neighbours which it is possible to generate is the number of critical arcs in a solution, *i.e.* adjacent operations in a solution.

Critical arcs are used here as starting points to change a given solution with the aim of finding better solutions. The change consists of reversing an arc's orientation with the intent of improving the current solution cost.

Hence, a neighbour of a given solution is defined by a pair of critical operations, say $(o_{ij} \rightarrow o_{i'j'})$, with the arc between them in reversed orientation $(o_{i'j'} \rightarrow o_{ij})$ from what it was in the given solution. The process of swapping two critical operations can take some operations off the list of the critical operations, but also can add others.

5.3.4 A Hybrid Algorithm for the 2^{nd} Model

The Job-shop scheduling problem is a NP-hard problem as already stated above. Therefore an enumerative scheme alone is computationally expensive as the problem size increases.

Because conventional enumerative strategies are very time-consuming, many meta-heuristic strategies have been applied to this class of problems with a reasonable success [89]. In this case the aim is to quickly reach a near-optimal solution, which is a good upper-bound on the cost.

The algorithm described in the following sections takes advantage of the best of the local search strategies, in terms of walking quickly through better and better solutions, and also of the enumerative strategies for thoroughly searching the search space seeking for the optimal solution.

Algorithm 5.3.1 describes the three-level structure we propose to increase the performance of the algorithm in finding optimal solutions for the single-line railway scheduling problem discussed in this paper.

Like the previous algorithm implementations, each stage of this algorithm was also developed within the ILOG Scheduler [48, 49] framework.

Algorithm 5.3.1

```
begin
 solution := True

 1. Use a fast heuristic to find a good initial solution for the problem.

 while solution == True do

       while stopping criteria == False do
             2. Take a not yet selected neighbour and swap the pair of activities.
              Re-examine the remaining critical activities in order to adjust
              them to the new configuration of activities. Any new conflict
              is resolved using the SPT dispatch rule.
       od

       With the best solution value found so far;
       3. Use a Branch-and-bound algorithm to look for a better solution

```

```
25          if a better solution is found
26             then  solution := True
27             else  solution := False; //The optimal solution has been found.
28          fi
29    od
31 end
```

The algorithm's first stage (line 5) is to find a feasible initial solution. Though speed in constructing the first solution is the priority at this stage, the quality of the solution is not to be neglected.

After having extracted the critical path from the initial solution, the next stage (line 14) is responsible for generating an improved solution. This process goes on as long as a better solution can be found using the strategy employed by the local search.

From the local minimum reached by the local search strategy, a branch-and-bound procedure is used to find a better solution. The main objective at this point is to get out of the local minimum. If no better solution can be found at this stage, it will mean that the last local minimum point was in fact the optimal solution. Otherwise, the algorithm returns to the local search phase.

The following sections describe in more detail how each one of these procedures actually perform their task.

5.3.4.1 Finding the First Solution

There are no limits imposed on the problem in terms of maximum overall delay initially. This is because it is not possible to guess what a potential solution configuration is going to look like beforehand, given that the system may have constraints (*e.g.* those discussed in Section 2) which need to be satisfied all together.

Once the constraints are posted in the system, the variable domains are pruned as those constraints are propagated through the system.

However, the constraints which determine the order of each task on a machine are not posted yet. The disjunctive graph described in Section 5.3.1 is not acyclic therefore.

In order to make the graph acyclic, the first solution algorithm creates a list of pairs of conflicting activities. From this list it picks up a conflicting pair of tasks by a chronological order and sequences them according to the *shortest processing time* (SPT) dispatch rule [71]. New conflicts may come up after ordering a conflict in the list. A new list is then created and the process continues until there are no more conflicting tasks to be sequenced, which means the graph is completely acyclic. The unconflicting operations are performed at their earliest processing time.

This procedure for finding the first solution is actually one pass procedure of the algorithm described in Section 5.2.4. Here, once a solution is found the algorithm stops, whereas the one described in Section 5.2.4 backtracks when a solution cost is not within the current upper-bound.

The SPT dispatch rule used serves the purpose of minimizing the delay over each pair of conflicting operations. However, this procedure is done locally, only taking into account the pair of tasks involved, which may not be the best option taking a broad view of the configuration of the conflicts in the system.

5.3.4.2 Walking Through Solutions

Many variants of local search have been proposed in the last decade for dealing with the job-shop scheduling problem [89]. Usually a local search strategy is used as an alternative to enumerative algorithms due to computational high cost of the complete search. Unlike this usual approach, Baptiste *et al.* [8] presented a combined approximation and enumerative algorithm to speed up the process of reaching near-optimal solutions.

The structural difference between their approach and ours is that their local search works up to a certain number of iterations without improvement and then stops. The enumerative algorithm is then launched to search for the optimal solution. Our enumerative algorithm is instead used to free the local search from the local minimum, if a better solution can be found, returning the control to the local search afterwards.

The local search we propose here is a hill-climbing strategy based upon the introduction of a *perturbation* to the critical path of a solution.

A perturbation to the critical path (see description in Section 3.4) is provoked by changing the orientation of the arc of a pair of adjacent operations. To carry out this perturbation each neighbour is taken sequentially until a better solution than the current solution is found.

The reversal of the arc is then propagated onto the set of operations. As a result of this modification, some critical operations may no longer be critical. Therefore, the algorithm revisits each of the initial critical operations in order to assess whether their orientation needs to be either kept, changed, or discarded in order to minimize the overall delay.

Moreover, some new conflicts might be created due to the modifications carried out as described above. To deal with the new conflicts the algorithm sets up an orientation between each of the conflicting pair of activities using the chosen dispatch rule. When all the remaining conflicts are resolved, the cost of this new solution is then compared against the current cost. An improved solution is taken as the new current solution, or a new neighbour is chosen otherwise.

The hill-climbing algorithm is trapped in a local minimum when all the neighbours have been tried without leading to any improvement. The algorithm then hands the control over to the enumerative algorithm, described in the following Section, to search for a better solution, if there is one.

5.3.4.3 Proving Optimality

The branch-and-bound algorithm devised to resolve the conflicts iterates chronologically through the list of conflicting operations. It fixes an arc orientation between the conflicting activities, so that we can have an acyclic graph (see Section 5.3.1) after having all the conflicts resolved. A similar chronological way of resolving the conflicts was also used by Higgins [45].

This way a solution is found when the algorithm reaches a leaf of the search tree. Our first solution is generated by this algorithm up to the point it finds a solution, as explained in Section 5.3.4.1.

The optimization process of the algorithm consists of incrementally adding a new upper-bound constraint to the problem, each time a solution is found. This new upper-bound for the cost of the next solution constrains the algorithm to find an improved solution, if there is one.

During the search, any time the algorithm reaches a partial solution whose cost exceeds the current upper-bound, the algorithm backtracks from that point to the latest orientation assignment so that a different arc orientation can be tried towards finding a better solution. When no better solution can be found, that will mean the previous solution was in fact the optimal solution.

Those operations, which are not conflicting with any other operation, are simply ordered according to their earliest possible departure time. For those which conflict, the *Shortest Processing Time* heuristic [71] (see Algorithm 5.2.2) is used to schedule conflicting tasks and construct a partial solution by adding a precedence constraint between them. When a chosen task cannot be scheduled after those which have already been scheduled on the resource, because it would increase the already known total delay, the program backtracks in order to choose another orientation for the conflicting pair of operations. When the alternative orientation cannot be applied without exceeding the current upper-bound, it backtracks to an upper level of the search tree, choosing a different conflicting pair of tasks to reorient it.

5.3.5 Solving 21 Real-Life Problems – 2^{nd} Model Results

In this Section we present the results of solving the problems described in Section 5.2.3, now using the second model explained in Section 5.3. We use the combined structure of algorithms to solve the problems.

The hybrid program described in Section 5.3.4 is used here to solve a set of problems from [45], whose characteristics are shown in Table 5.2. Within this set of problems there are none of the special constraints we suggested in Section 2, although our program can handle them. Therefore, the aim here is twofold. First, to evaluate the performance of the proposed hybrid algorithm on solving this set of real-world problems. Second, again to compare the performance with that obtained by Higgins [45].

We used our hybrid program to solve the 21 problems described in Table 5.2 and the result yielded is depicted in Table 5.7. These experiments were performed on a networked PC Pentium III.

The *First Solution* columns show the time to find the first feasible solution to a problem and its respective delay. Likewise, the *Best Solution* columns show the result to reach the optimal solution for that particular problem and the overall minimum delay.

The *Iterative Improvement* columns show the first local minimum solution reached by the local search method proposed here and described in Section 5.3.4.2. The *Time*, in this column, is the necessary time to find the first improved local minimum solution and the *Delay* shows its value. When, for each problem, no better solution cost can be found, *i.e.* when the heuristic remained trapped in a local minimum, the last solution cost found is shown instead. The *LM* number shows how many times the local search got trapped in a local minimum and the third part of the algorithm (the complete algorithm) was called to free the heuristic from it (see Algorithm 5.3.1).

In the *CPU* column is shown the total time in seconds including the time to find the best solution and prove its optimality.

The result presented in Table 5.7 is, thus, an improvement over that presented when using only the 2^{nd} algorithm (see Section 5.2.4). First, there is an improvement

	First Solution		Iterative Improvement			Best Solution		
Problem	Time	Delay	Time	Delay	LM	Time	Delay	CPU
20	0.03	1815	0.37	1775	14	5.22	1224	5.27
21	0.03	253	0.14	210	1	0.14	175	0.15
22	0.03	294	0.08	238	2	0.14	137	0.15
23	0.02	116	0.06	151	1	0.06	116	0.07
24	0.03	517	0.12	391	2	0.22	349	0.25
25	0.03	410	0.12	285	1	0.12	285	0.14
26	0.02	344	0.07	352	2	0.13	338	0.15
27	0.02	303	0.09	310	2	0.16	161	0.17
28	0.02	249	0.07	257	1	0.07	249	0.09
29	0.03	409	0.12	321	2	0.23	223	0.24
40	0.02	350	0.09	282	2	0.15	215	0.16
41	0.02	199	0.11	334	1	0.11	199	0.12
42	0.01	441	0.16	439	4	0.51	388	0.58
43	0.03	716	0.15	690	4	0.83	560	1.03
44	0.01	427	0.14	476	2	0.23	389	0.26
45	0.03	359	0.09	291	1	0.09	228	0.11
46	0.02	526	0.12	526	1	0.12	526	0.21
50	0.17	805	3.64	565	2	5.98	555	22.85
51	0.21	1445	18.27	810	6	406.13	650	463.78
60	0.90	504	92.88	318	–	–	–	–
61	1.09	800	51.83	345	–	–	–	–

Table 5.7: Results of solving 21 problems using the hybrid algorithm. The respective times to find the first solution, to find the first solution by the iterative method, to find the best solution and to prove a solution is optimal, are all given in seconds. For each of that the correspond delay (*i.e. cost function*) is shown.

in terms of time performance for the large problems, as we can now find good solutions for all the 21 problems within less than 10 minutes. Second, because of the new model applied here we can also find solutions with smaller total delay for each problem compared with the first model. For instance, the optimal solutions with this new model for problems 50 and 51 are 555 and 650, respectively.

The performance of our hybrid algorithm is also depicted in Figure 5.10. The graph shows the result of running the program using the three phases and also using the branch-and-bound (described in Section 5.2.4) part of the program alone.

The hybrid algorithm did not show superior performance on problems 20-50, as is shown in Figure 5.10. However, its potentiality starts to be noticed from problem 51. The hybrid algorithm finds the optimal solution within 410s, whereas the optimal solution is found over 685s when running the complete part of program alone.

The hybrid algorithm also managed to find good solutions for problems 60 and

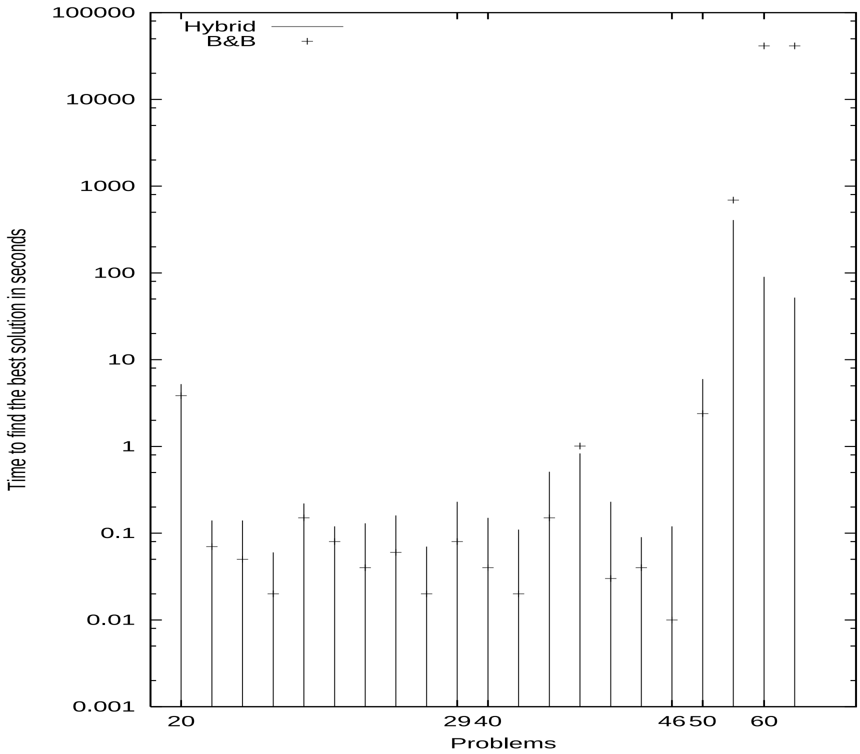


Figure 5.10: The results comparison between the Hybrid and Branch-and-Bound Algorithm running alone. The Branch-and-Bound was interrupted after 15 hours, for problems 60 and 61, without reaching a solution as good as that by the Hybrid algorithm.

61 in about 3min. These values are not reached by the complete algorithm in a period less than 15 hours, the program's limit time. Hence, the challenge is still to search for the optimal solutions for problems 60 and 61, in case the solutions found with costs of 318 and 345, respectively, are not the optimal.

The results of problems 20 and 51 call attention to the number of times the iterative improvement strategy was trapped in a local minimum and the corresponding amount of time to reach the best solution, 5.22 and 406.13 respectively. In addition to showing that the local search part of the program did not much help the algorithm in walking smoothly through improved solutions, these results also show, thus, the poor quality of the neighbourhood of these problems on yielding improved solutions.

Among these 21 problems, the optimal solution was promptly found for four problems, problems 23, 28, 41 and 46. However, the iterative method did manage to find the optimal solution only for problems 25 and 46 among these problems.

The hybrid algorithm proposed here did improve the performance on finding better solutions for those large problems (51, 60, and 61). However, it is still difficult to reach the optimal solution and prove optimality for problems 60 and 61 within a maximum of 15 hours.

Higgins [45] reported that problems of the size of 50 and 51 were solved, using a tailored MIP approach, in about 46 seconds and problems of the size of 60 and 61 within 10min. The result presented here is still far from being comparable with those in [45] in terms of performance. Nonetheless, our approach can handle more complex constraints, which are likely to be found in practice. Moreover, we can easily combine different approaches within the same structure of program and, therefore, get the best of both approaches, as shown with the hybrid algorithm presented here.

5.3.6 Using the Special Constraints: Some Small Examples

The experiments carried out both with the first model and the second model presented in the previous sections show the time performance of our algorithms. As pointed out in Section 5.2.3, the datasets used for the experiments, provided by Higgins [45], do not contain within any of these special constraints we presented in Chapter 2. However, the time performance of our algorithms would not be deteriorated by the inclusion of these constraints. On the contrary, the addition of these constraints may prune further the search space, as discussed in Chapter 4.

Therefore, to exemplify the utilization of the constraints described in Chapter 2, we are going to use the same problem used in Section 5.2.1, however, now under the second model. The optimal solution for that problem under the second model is shown in Figure 5.8, pp. 87. We are going to separately introduce one by one of those special constraints into the problem and present the schedule yielded in the new optimal solution. Thereafter, we are going to solve the problem mixing both models. For that, we are going to fix one of the trains in the problem as a passenger train, while the other three trains will be kept as in the second model.

The *Meet* Constraint

This constraint guarantees that two trains meet each other at a specified station. In the previous optimal solution for the problem used here as an example, the train #11 is delayed at station s_3 to resolve a conflict. This is shown in Figure 5.8.

Consider now that trains #11 and #14 need to meet each other at station s_3 in order to have a common dwell for at least 10 time units. The introduction of this constraint into the problem and its consequent propagation through the problem's variables ceased the conflict between these two trains. Hence, the problem to be solved now is one with only two conflicts: those between trains #13 and #14, and trains #14 and #16. The new schedule, whose trains #11 and #14 meet at station s_3, is shown in Figure 5.11.

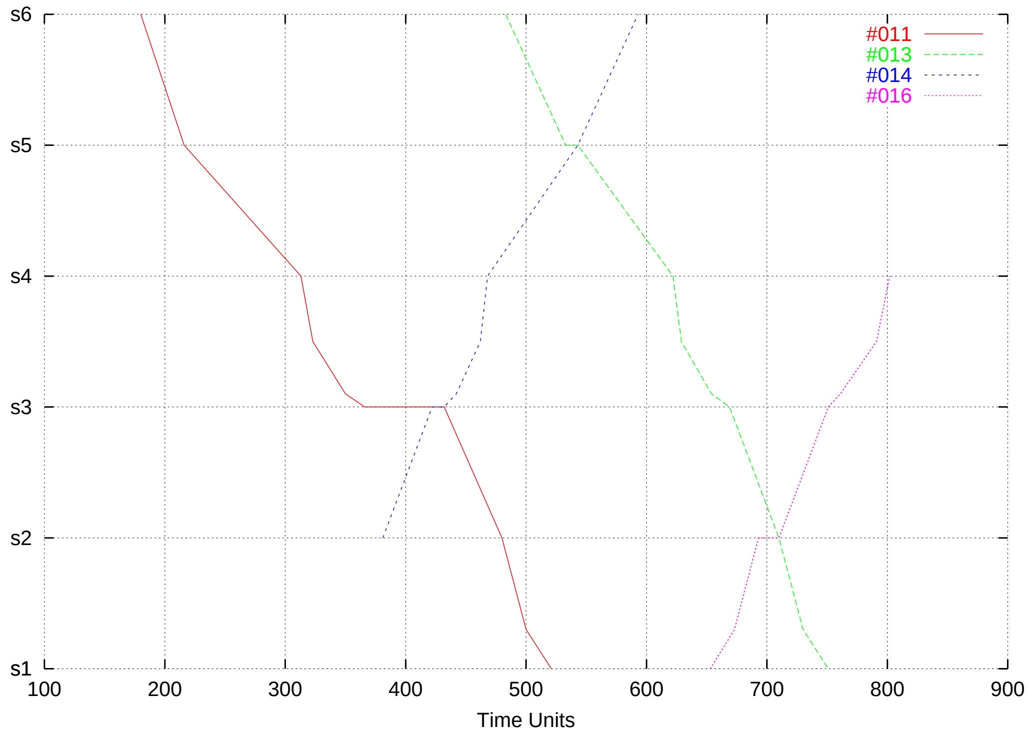


Figure 5.11: The result when a *Meet* constraint between trains #11 and #14 is added to the problem in Figure 5.3.

The *Form* Constraint

In the solution shown in Figure 5.8, one may want the vehicle used for train #13 to be also used for train #16. If this new constraint is introduced into the problem, the new optimal solution must have train #16 only departing from station s_1 after train #13 ends its journey at that station. Thus, to illustrate the use of this constraint, let us consider that trip #16 ought to start only after trip #13 ends, plus the necessary time to prepare the vehicle for the next journey.

As in the previous example, once this constraint is posted into the problem and the propagation of the constraints occurs, the conflict between train #13 and #16 ceases to exist. The problem is simplified to a problem with only 2 rather than 3 conflicts. The optimal solution for this new problem is shown in Figure 5.12.

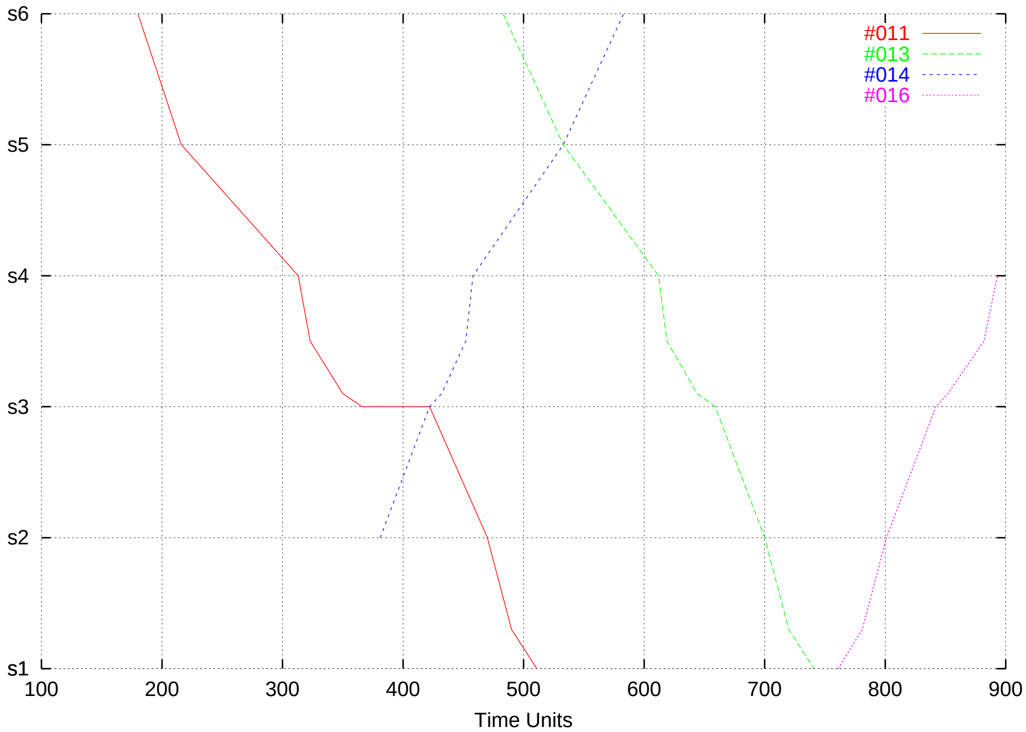


Figure 5.12: The result when a *Form* constraint between trains #13 and #16 is added to the problem in Figure 5.3.

The *Blocking* Constraint

The blocking constraint is to prevent any service from being scheduled to a track segment within a time interval. This is suitable if one wants to set a track segment to maintenance, for instance.

Hence, consider that the track segment between stations s_5 and s_6 is blocked within the time interval from 500 to 520.

The result of introducing the blocking constraint into the problem is shown in Figure 5.13. Note that how train #14 was greatly delayed due to this constraint. It is now departing after 400 time units and completing its journey not before 600 time units.

The *Headway* Constraint

The headway constraint proposed in this thesis sets a minimum time between two trains according, for instance, to their characteristics.

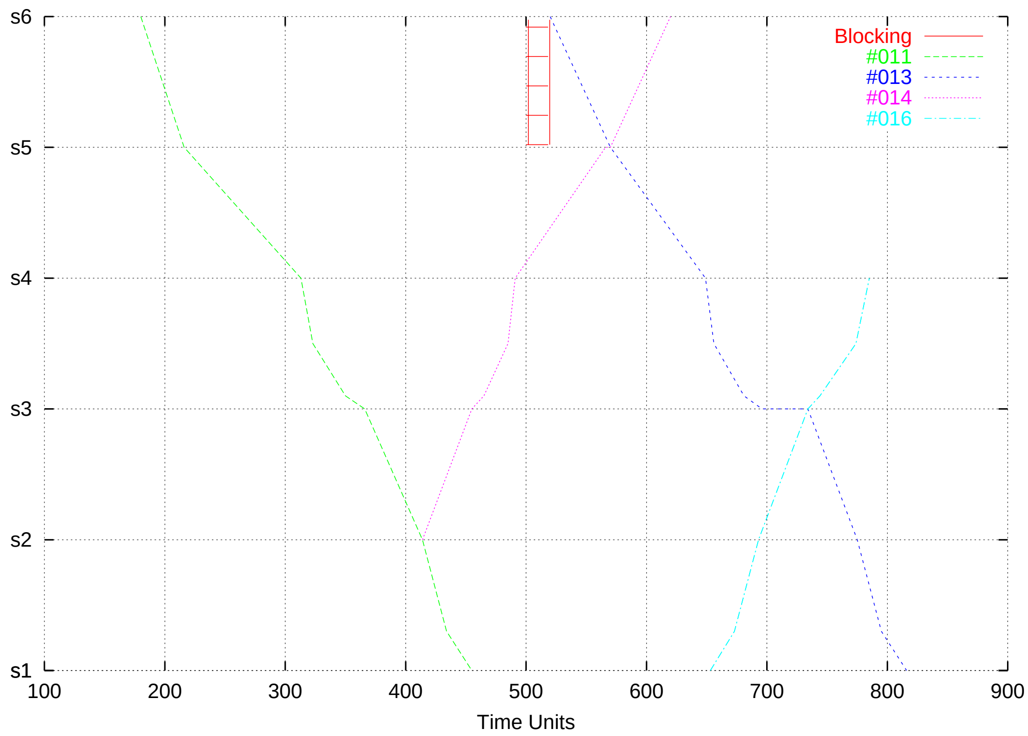


Figure 5.13: The result when a *Blocking* constraint between stations s_5 and s_6 is added to the problem in Figure 5.3.

To illustrate the use of this constraint we are going to add another service #20 to the problem example we have used so far. This freight service carrying a dangerous load is, for the sake of this example, arbitrarily planned to depart at 583 from station s_5 and arrive at 633 at station s_6. Hence, this service is overlapping with the train #14's last part of journey (see Figure 5.8, pp. 87).

The solution for this new problem, if the headway constraint is not posted, can have train #20 being scheduled before train #14. However, due to operational reasons, if train #20 departs first from the station s_5 a minimum time headway of $d_{20\_14} = 70$ is set between these two trains, or $d_{14\_20} = 60$ time units otherwise. Note that both trains take 50 time units to travel from station s_5 to station s_6.

Therefore, in order to yield the minimum total delay train #14 may (or may not) be chosen to wait for train #20 to complete its journey at the station s_6. This will, of course, depend on the configuration of the entire schedule. The solution for the example problem described above is to schedule train #20 after train #14 increasing, this way, the total delay by only 10 time units. This result is shown in Figure 5.14.

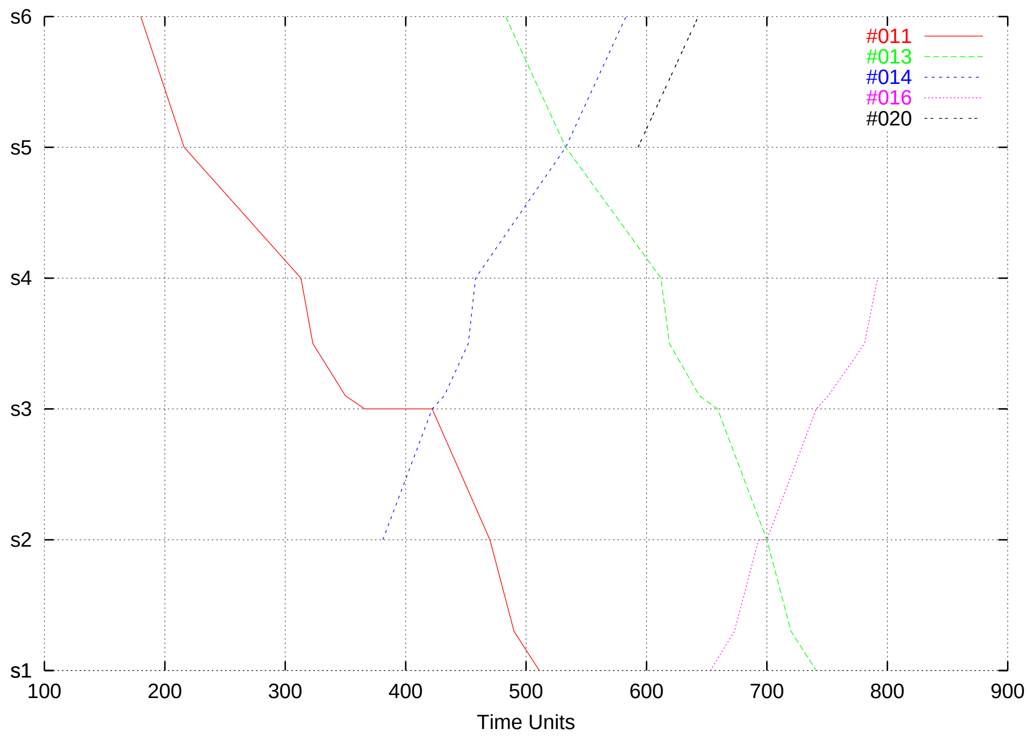


Figure 5.14: The result when a *Headway* constraint between trains #13 and #14 is added to the problem in Figure 5.3.

Coping with both Models

Both models described in Sections 5.2 and 5.3 can be handled together by the algorithms described in this thesis. The difference between these two models is in the way the constraints between each pair of tasks are dealt with as described previously.

In order to illustrate how the combined model would be possible, we are going to set the constraint between each pair of tasks of train #11 to be as in the first model, while the other trains will remain as described for the example in Figure 5.8.

The solution for this combined-model problem is shown in Figure 5.15. Note that train #11 is no longer delayed at station s_3, whereas train #16 is still possible to be dalayed.

5.4 Summary

In this chapter we presented two models and four main methods to solve the single-track railway scheduling problem.

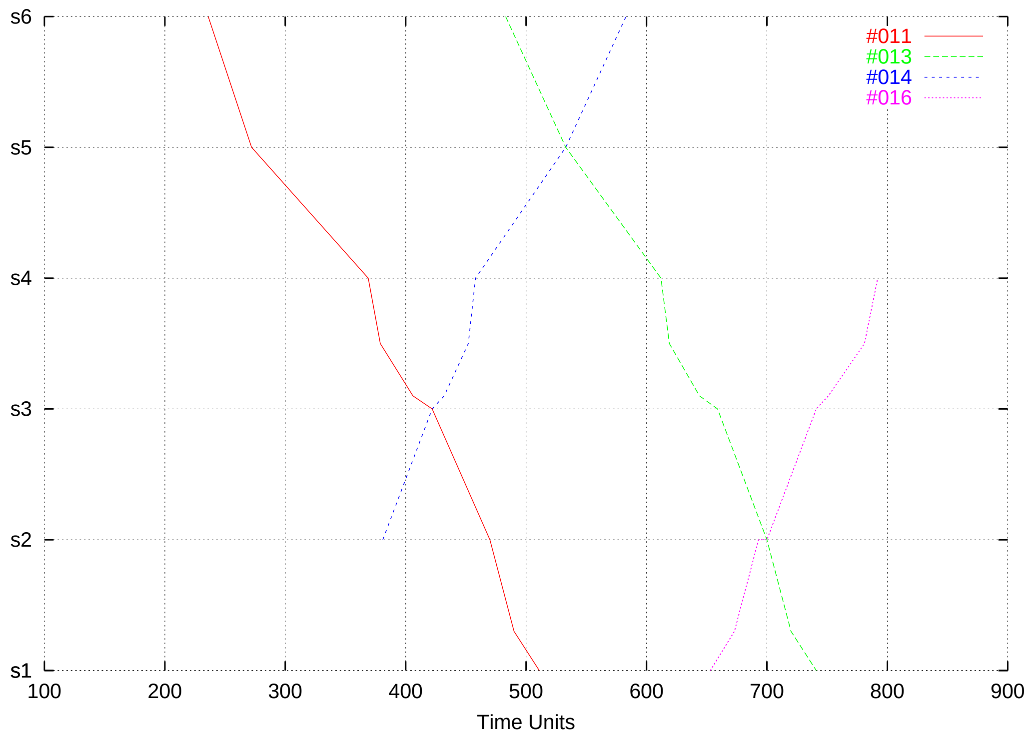


Figure 5.15: The result when train #11 is made a passenger train, as in the 1^{st} Model – combining the models.

These two models map the Single-Track Railway scheduling problem to a special case of Job-Shop Scheduling problem. We discussed their applicability in practice, and pointed out that they differ from each other in the way the conflicts are resolved. For instance, in the first model a conflict is resolved by re-timing one of the trips at its departure time up to the point the conflict is resolved, whereas, in the second model, a conflict is resolved by delaying only the conflicting piece of one of the two trips (and subsequent pieces of that trip). In this way, the section of the trip before the conflicting point does not need to be re-timed.

In order to solve the 21 problems considered in this thesis, we proposed four main methods. The first and the second methods are a branch-and-bound like search. The third method is based on the implementation of the second method: however, extra bounds are applied to the problem. The fourth strategy is a hybrid algorithm combining a local search with a complete search.

The first method did not work well on solving the 21 problems considered. It took too long to find solutions for the problems when using the first model and, in addition, this first method was not good enough to find optimal solutions for problems 50 and 51 within 15 hours [69], a maximum limit time to run the program used along this thesis. We could not even find a first solution for problems 60 and 61 using this first algorithm.

The second proposed algorithm improved on the first algorithm in terms of time performance, as discussed in Section 5.2.4. Using this second strategy, for the first proposed model, we were able to find optimal solutions for 19 out of 21 problems. In addition to increasing the number of problems for which we could find optimal solutions, we were also able to find initial solutions for problems 60 and 61. However, within the limit time of 15 hours, only one solution was found for each of these two problems.

The third algorithm consists in fact of a group of three heuristics, which use the second algorithm as basis. The idea behind each of these heuristics is to set a limited region in which to search for solutions, as discussed in Section 5.2.6. We do this by adding bound constraints on the maximum tardiness variable.

Although they are heuristics, we devised them such that if appropriately used together, the optimal solution will always be found. This is shown in Section 5.2.6, and the results on finding better solutions did further improve. The times to find optimal solutions were greatly reduced, again using the first proposed model, and we were able to find better solutions for problems 60 and 61, but not yet prove their optimality.

In the second proposed model, the constraints that link subsequent tasks in a trip are relaxed, so that subsequent tasks can have a time gap between them in order to resolve a conflict, as discussed in Section 5.3. This way of resolving conflicts is similar to that adopted by Higgins [45].

Furthermore, we represent the single-track railway problem by a disjunctive graph, a representation that is often used for the job-shop problem [71]. Using this representation, the fourth algorithm is proposed. It combines the best of a local search strategy, in terms of walking quickly through improved solutions, and also of the enumerative strategies for thoroughly searching the search space seeking for the optimal solution.

The results show the superiority of the hybrid algorithm over the 2^{nd} algorithm on the large problems: 51, 60 and 61. Good solutions for problems 60 and 61 were now found within 3 minutes. However, we are not yet able to prove optimality for these problems.

Chapter 6

Conclusions

6.1 Introduction

In this chapter, the main contributions and results presented in this thesis are summarized. In addition, some of the practical details which differentiate this research from others in single-track railway scheduling are once more highlighted, and future research based on the outcome of the experiments carried out in this thesis is suggested.

6.2 Contributions

6.2.1 The Models

In this thesis two models for the single-track railway scheduling problem are presented. Both models map the Single-Track Railway scheduling problem to a special case of the Job-Shop Scheduling problem.

In addition to the models for the problem, we have introduced a more realistic approach when modelling the physical representation of a railway. This is done by taking into account the actual signals placed along the track delimiting each track segment. These signals control whether a train can or cannot go on that particular track segment, avoiding thus the possibility of trains running into each other. Previous authors adopted an approximation to what is found in practice by adding the headway constraints into their model [45, 50, 54]. Moreover, we do not restrict the model of our network to have only one track segment between any two passing points, as was the case in [58].

Therefore, the work presented in this thesis separates the network model from the problem model. The separation of the two definitions allows the method used to solve the problem to be general, rather than specific for a particular network characteristic.

The two models proposed specifically to map the problem, have their applicability in practice, and they differ from each other by the way the conflicts are resolved. For instance, in the first model a conflict is resolved by re-timing one of the trips at its departure time up to the point the conflict is resolved.

In the second model, on the other hand, a conflict is resolved by delaying only the conflicting piece of one of the two trips (and subsequent pieces of that trip). In this way, the section of the trip before the conflicting point does not need to be re-timed. This second way of resolving the conflicts allows us to find better solutions in terms of their cost.

Both models were tackled in this thesis separately. However, we can also deal with more flexible type of models. We can handle both types of operation together: trains that cannot be delayed at any intermediate passing point (1^{st} Model), those that can be delayed at any passing point (2^{nd} Model), and trains that can only be delayed at a set of pre-specified passing points. This has been demonstrated on small problems in Section 5.3.6, because each constraint between two consecutive tasks needs to be individually specified. It would be straightforward to automate this, however.

Modelling the single-track railway scheduling problem as a special case of the job-shop scheduling problem opens up the possibility to the first problem of taking advantage of solving methods that have been investigated for decades in the scheduling scientific community, as pointed out in Chapter 3. This allowed us to explore, for the single-track railway scheduling problem, one of the techniques often used successfully for job-shop scheduling problems. This is further detailed when discussing the contributions in the solving methods below.

6.2.2 The Special Constraints

Among some advantages provided by the CP paradigm is its ease to code complex constraints. Another advantage, also related to the capability of coding complex constraints, is that these constraints have an important and active role within the solving method, as discussed in Chapter 4.

In this thesis we have presented four special types of constraints which have hitherto been left aside by the previous authors. The previous authors often dealt only with simple temporal constraints: constraints to determine the sequence of a trip, and the disjunctive constraints of trains requiring the same track segment at an overlapping time [46, 54, 86].

However, the special constraints presented in this thesis have great applicability in practice. The *meet* constraint discussed in Chapter 2 can be, for instance, used as a feature provided to the user in a (re)scheduling system, such as that proposed in [26], to impose a required meeting for two trains at a particular station. Similarly, the *blocking* constraint can be a feature to impose an emergency track maintenance. Note that none of these special constraints restricts the method to be used to solve the problem, although they may act on reducing the search space of the problem.

6.2.3 The Solving Methods

Four main strategies to solve the single-track railway scheduling problem were proposed in Chapter 5. The first and the second methods are a branch-and-bound-like searching method. The third method is based on the implementation of the second method, however, extra bounds are applied to the problem. The fourth strategy is a hybrid algorithm combining a local search with a complete search. Each of the methods proposed in this thesis is not restricted to a specific type of solution, such as pre-fixed trains overtaking, as that proposed by Jovanovic [54].

To improve the time performance on searching for optimal solutions, we proposed the third method, which uses a different objective function so that the propagation is speeded up. This alternative way consists of bounding the maximum tardiness variable, as this is immediately propagated to all the individual trips delay. On the other hand, bounding the total tardiness is much weaker, and often tighter bound on the total tardiness will not lead to tighter bounds on the individual trip's delay.

Using the maximum tardiness objective of a problem as a means to achieve the total tardiness objective was shown to be very effective for our problems. This has no precedent in the scheduling literature (to the author's best knowledge), and no doubt this is the first time that this procedure has been applied for the single-track railway scheduling problem. It could be applied in other contexts in any optimization problem where the objective does not give effective constraint propagation and there is a suitable surrogate objective available.

As already stated, in constraint programming constraints of any type do not restrict the method used to solve problems. In the light of this, we proposed a method, adapted from the job-shop literature, to solve the single-track railway scheduling problem. This method uses one of the features often used to solve job-shop problems: the critical path of a problem's solution. Hence, we devised a local search heuristic, which by swapping the critical activities, looks for improved solutions for the problem. This is the first time (to the author's knowledge) that this procedure has been applied for the single-track railway scheduling problem.

Because we can apply general algorithms to the single-track railway scheduling problem, we can handle, as a result of that, more flexible type of models, as already stated when discussing the contributions in the models above.

6.3 Future Work

There are two major areas that can deserve further research: the model and solving methods for the problem. We can split the model of the problem into two other sub-aspects: the generalization of this model to a multi-track context, and also the improvements in modelling the network in order to let it match even more realistically what is encountered in practice.

A multi-track railway network can be thought of as a set of single-track corridors. However, resolving to optimality each of the corridor's problem (object of this thesis) does not necessarily lead to the global optimal for the general problem. Therefore, the solving method for the multi-track railway scheduling problem requires devising a more general algorithm. This algorithm, unlike those developed in this thesis, must take into account each railway corridor and resolve them all simultaneously. However, the insights from the algorithms developed for this thesis might also be of great use for the multi-track problem.

In order to make the model for the single-track network more realistic, one of the future improvements might be to provide the network model with constraints to impose restrictions on the type of services running on particular track segments at particular times. Such constraints may restrict certain services to use the track segment only during a pre-specified set of time windows. These constraints can be seen as a generalization of the *blocking* constraint (Chapter 2), by being selective on what type of services to block and when.

Another improvement to the single-track network model can be in the way passing points are handled. In this thesis, every train is considered capable of stopping at any passing point. However, we know that in practice, there are stations and sidings that cannot hold certain vehicles due to their large length, for instance. Therefore, those stations and sidings could not be used for resolving conflicts of these vehicles. Such a practical situation can be handled by extending the model for passing points, so that the characteristic of the vehicle would be taken into account. Note that, a particular vehicle being allowed or not to stop at a particular passing point would be in the definition of the network and not in the algorithm used to solve the problem.

Apart from the improvement that can be done on modelling, further research can be done with respect to the algorithms at least in two directions. The first would be towards improvements in the systematic algorithms for resolving conflicts. The second would be to apply more sophisticated metaheuristics, such as Tabu Search, to the process of quickly finding improved solutions: the second stage in the fourth proposed algorithm (see Section 5.3.4).

The systematic algorithms proposed in this thesis are based on chronologically resolving conflicts. The example evaluated in Section 5.2.2 teaches us that a non-chronological way of resolving the three conflicts in that example would result in fewer nodes being explored to reach the optimal solution. For example, if the second conflict, chronologically, were resolved first, that would show better options for how to resolve the other two remaining conflicts. Thus, that problem could be solved to optimality at once.

In order to have a non-chronological way of resolving the conflicts, it would be necessary to attribute a weight to each of the conflicts and then resolve first that conflict considered to be the most critical.

The second direction in order to improve the algorithms might be in applying a better metaheuristic to the fourth algorithm presented in Section 5.3.4.

The hill-climbing heuristic used as the metaheuristic in the three stage algorithm discussed in Section 5.3.4, is rather a simple metaheuristic, a more sophisticated one could be placed there instead. The Tabu Search implementation of Nowicki and Smutnicki [68] is considered to be the most efficient and effective implementation proposed for job-shop problem [13] and, thus it could be used to replace our hill-climbing.

Nowicki and Smutnicki incorporated within their implementation a mechanism to free their metaheuristic from a local minimum solution cost by backtracking to previous improved solutions in the tabu long-list memory. This is discussed in Section 3.5.2.2. Such a metaheuristic, if implemented within our fourth algorithm, would prevent the metaheuristic from easily getting irretrievably trapped in a local minimum solution, thus the systematic algorithm would be called fewer times.

Another area of possible improvement is in finding good bounds to impose on the problem so that propagations can occur, cutting off fruitless parts of the search space. The heuristics proposed for the third algorithm in Section 5.2.6 showed the potential a tight bound can have on the time performance of the algorithm. However, in order to find these bounds, more understanding is needed of the structure of the problem.

6.4 Achievements of the Research

The research carried out in this thesis has presented more realistic ways of modelling the single-track railway scheduling problem than have previously been considered. Two different models are proposed, appropriate to different types of rail operation. The problem is mapped into a special case of a well known class of problems in OR: the job-shop scheduling problem. In doing this, we open up an avenue to the application of job scheduling strategies to solve the first problem. In addition, the flexibility of constraint programming allows us to deal with more complex practical constraints than have been considered in the literature hitherto.

In this thesis, we have explored some of the techniques often applied in the job-shop context. The results yielded by our algorithms show the applicability of our approach to real-life problems, both in terms of performance and quality of the solutions.

Bibliography

[1] Adams, J., Balas, E., and Zawack, D. The Shifting Bottleneck Procedure for Job Shop Scheduling. *Operations Research*, (34):391–401, 1988.

[2] Allahverdi, A., Gupta, D., and Aldowaisan,T. A Review of Scheduling Research Involving Setup Considerations. *Omega*, 27(2):219–239, 1999.

[3] Allen, J. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[4] Applegate, D. and Cook, W. A Computational Study of the Job-Shop Scheduling Problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[5] Backer, B., Furnon, V., Shaw, P., Kilby, P., and Prosser, P. Solving Vehicle Routing Problems Using Constraint Programming and Metaheuristics. *Journal of Heuristics*, 6(4):501–523, 2000.

[6] Balas, E. Machine Sequencing via Disjunctive Graphs: an Implicit Enumeration Algorithm. *Operations Research*, 17:941–957, 1969.

[7] Balas, E. and Vazacopoulos, A. Guided Local Search with Shifting Bottleneck for Job Shop Scheduling. *Management Science*, 44(2):262–275, 1998.

[8] Baptiste, P., Le Pape, C., and Nuijten, W. Constraint-Based Optimization and Approximation for Job-Shop Scheduling. AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, IJCAI-95, Montreal Canada, 1995.

[9] Bater, W. M. Computer Aided Railway Engineering. In B. Mellit, R. J. Hill, J. Allan, G. Sciutto, and C.A. Brebbia, editors, *Computers in Railways VI*, pages 199–211, Lisbon, 1998. Sixth International Conference on Computer Aided Design, Manufacture and Operation in the Railway and other Advanced Mass Transit Systems, WIT press – Computational Mechanics Publications. Comreco Rail Ltd York, England – http://www.comreco-rail.co.uk.

[10] Beck, J.C., Davenport, A.J., and Fox, M.S. Five Pitfalls of Empirical Scheduling Research. In *Proceedings of CP'97*, pages 390–404, 1997.

[11] Bessiere, C. and Cordier, M.O. Arc-Consistency and Arc-Consistency Again. In *Proceedings of AAAI'93*, pages 108–113, 1993.

[12] Bessiere, C., Freuder, E.C., and Régin, J.-C. Using Inference to Reduce Arc Consistency Computation. In *Proceedings of IJCAI'95*, pages 592–598, 1995.

[13] Blażewicz, J., Domschke, W., and Pesch, E. The Job-Shop Scheduling Problem: Conventional and new Solution Techniques. *European Journal of Operational Research*, (93):1–33, 1996.

[14] Blażewicz, J., Pesch, E., and Sterna, M. The Disjunctive Graph Machine Representation of the Job Shop Scheduling Problem. *European Journal of Operational Research*, (127):317–331, 2000.

[15] Brailsford, Sally C., Potts, C. N., and Smith, Barbara M. Constraint Satisfaction Problems: Algorithms and Applications. *European Journal of Operational Research*, (119):557–581, 1999.

[16] Brucker, P., Hurink, Johann, Jurish, B., and Wöstmann, Birgit. A Branch and Bound Algorithm for the Open-Shop Problem. *Discrete Applied Mathematics*, 76:43–59, 1997.

[17] Brucker, P. and Jurish, B. A new Lower Bound for the Job-Shop Scheduling Problem. *European Journal of Operational Research*, (64):156–167, 1993.

[18] Brucker, P., Jurish, B., and Sievers, B. A Branch and Bound Algorithm for the Job-Shop Scheduling Problem. *Discrete Applied Mathematics*, 49:109–127, 1994.

[19] Cai, X. and Goh, C.J. A Fast Heuristic for The Train Scheduling Problem. *Computers and Operations Research*, 21(5):499–510, 1994.

[20] Caprara, A., Focacci, F., Lamma, E., Mello, P., Milano, M., Toth, P., and Vigo, D. Integrating Constraint Logic Programming and Operations Research Techniques for the Crew Rostering Problem. *Software – Practice & Experience*, 28:49–76, 1998.

[21] Carey, M. and Lockwood, D. A Model, Algorithms and Strategy for Train Pathing. *Journal of Operational Research Society*, 46:988–1005, 1995.

[22] Carlier, J. The one Machine Sequencing Problem. *European Journal of Operational Research*, (11):42–47, 1982.

[23] Carlier, J. and Pinson, E. An Algorithm for Solving the Job-Shop Problem. *Management Science*, 35(2):164–176, 1989.

[24] Carlier, J. and Pinson, E. Adjustment of Heads and Tails for the Job-Shop Problem. *European Journal of Operational Research*, 78:146–161, 1994.

[25] Caseau, Y. and Laburthe, F. Improved CLP Scheduling with Task Intervals. *Proceedings of the* 11^{th} *International Conference on Logic Programming*, 1994.

[26] Chiu, C.K., Chou, C.M., Lee, J.H.M, Leung, H.F., and Leung, Y.W. A Constraint-Based Interactive Train Rescheduling Tool. *Proceedings of CP*, pages 104–118, 1996.

[27] Colmerauer, A. Prolog in 10 Figures. In *Proceedings of IJCAI'83*, pages 487–499, 1983.

[28] Della Croce, F., Tadei, R., and Volta, G. A Genetic Algorithm for the Job Shop Problem. *Computers and Operations Research*, 22:15–24, 1995.

[29] Dell'Amico, M. and Trubian, M. Applying Tabu-Search to the Job Shop Scheduling Problem. *Annals of Operations Research*, 41:231–252, 1993.

[30] Deville, Y. and van Hentenryck, P. An Efficient Arc Consistency Algorithm for a Class of CSP Problems. In *Proceedings of IJCAI'91*, pages 325–330, 1991.

[31] Dexter, K. L. W. Scheduling an Urban Railway. In Wren, Anthony, editor, *Computer Scheduling of Public Transport: Urban Passanger Vehicle and Crew Scheduling'80*, pages 147–180, 1980.

[32] Dincbas, M., Simonis, H., and van Hentenryck, P. Solving a Cutting-Stock Problem in Constraint Logic Programming. In *Proceedings of the* 5^{th} *International Conference on Logic Programming*, pages 42–58, 1988.

[33] Esquirol,P., Lopez, P., Fargier, H., and Schiex, T. Constraint Programming. *JORBEL (Belgian Journal of Operations Research, Statistics and Computer Science)*, 35(2):5–36, 1995.

[34] Fernández, A. and Hill, Patricia M. A Comparative Study of Eight Constraint Programming Language. *Constraints: An International Journal*, 5:275–301, 2000.

[35] Freuder, E.C.. A Sufficient Condition for Backtrack Free Search. *Journal of the ACM*, pages 24–32, 1982.

[36] Fries, G. A. and Sprakelaar, J. van. Computer Aided Railway Engineering. In B. Mellit, R. J. Hill, J. Allan, G. Sciutto, and C.A. Brebbia, editors, *Computers in Railways VI*, pages 53–59, Holland Railconsult, Rail Traffic Systems, 1998. Sixth International Conference on Computer Aided Design, Manufacture and Operation in the Railway and other Advanced Mass Transit Systems, WIT press – Computational Mechanics Publications.

[37] Garey, M. R. and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Edited by Freeman, San Francisco, 1979.

[38] Glover, F. and Laguna, M. *Tabu Search.* Kluwer Academic Publishers, 1997.

[39] Goldberg, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, 1989.

[40] Golomb, S.W. and Baumert, L.D. Backtracking Programming. *Journal of the ACM*, (12):516,524, 1965.

[41] Hall, N.G. and Sriskandarajah, C. A Survey of Machine Scheduling Problems with Blocking and *no-wait* in process. *Operations Research*, 44(3):510–525, 1996.

[42] Haralick, R. and Elliot, G. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[43] Harvey, W.D. and Ginsberg, M.L. Limited Discrepancy Search. In *Proceedings of IJCAI'95*, 1995.

[44] Henz, M. and Würtz, J. Using Oz for College Timetabling. In *Proceedings of PATAT'95*, 1995.

[45] Higgins, A. *Optimisation of Train Schedules to Minimise Transit Time and Maximise Reliability.* PhD thesis, School of Mathematics/School of Civil Engineering, Queensland University of Technology Australia, 1997.

[46] Higgins, A., Kozan, E., and Ferreira, L. Optimal Scheduling of Trains on a Single Line Track. *Transportation Research*, 30(2):147–161, 1996.

[47] Hurley, J. Optimized Control of Railway Operations. In IMechE Conference Transactions, editor, *International Conference on Railways as a System: Working Across Traditional Boundaries*, May 1999.

[48] ILOG S.A. ILOG *Scheduler Reference Manual, 4.0 edition 1997.*

[49] ILOG S.A. ILOG *Scheduler User Guide Manual, 4.0 edition 1997.*

[50] Isaai, M.T. and Singh, M.G. An Intelligent Constraint-Based Search Method for a Single-Line Passenger-Train Scheduling Problems. *The Internation Conference on the Practical Application of Constraint Technologies and Logic Programming*, pages 79–91, 2000.

[51] Isaai, M.T. and Singh, M.G. An Object-Oriented, Constraint-Based Heuristic for a Class of Passenger-Train Scheduling Problems. *IEEE Transactions on Systems, Man and Cybernetics–Part C: Applications and Reviews*, 30(1):12–21, 2000.

[52] Jaffar, J. and Maher, M. J. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 20(19):503–581, 1994.

[53] Johnson, S.M. Optimal Two- and Three-Stage Production Schedules with Setup Times Included. *Naval Research Logistics Quartely*, 1:61–67, 1954.

[54] Jovanović, D. *Improving Railroad On-Time Performance: Models, Algorithms and Applications*. PhD thesis, The Wharton School, University of Pennsylvania, 1989.

[55] Kraay, D., Harker, P. T., and Chen, B. Optimal Pacing of Trains in Freight Railroads: Model Formulation and Solution. *Operations Research*, 39(1):82–99, 1991.

[56] Kravchenko, S.A. A Polynomial Algorithm for a Two-Machine *no-wait* Job-Shop Scheduling Problem. *European Journal of Operational Research*, 106:101–107, 1998.

[57] Kreipl, S. A Large Step Random Walk for Minimizing Total Weighted Tardiness in a Job Shop. *Journal of Scheduling*, 3:125–138, 2000.

[58] Kreuger, P., Carlson, M., Olsson, J., Sjöland, T., and Aströ,E. Trips Scheduling on Single Track Networks - The TUFF Train Scheduler. *CP'97 Workshop on Industrial Constraint – Directed Scheduling*, pages 1–12, 1997.

[59] Laarhoven, P.J.M. van, Aarts, E.H.L., and Lenstra, J.K. Job Shop Scheduling by Simulated Annealing. *Operations Research*, 40(1):113–125, 1992.

[60] Le Pape, C. Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-Based Scheduling Systems. *Intelligent Systems Engineering*, 2(3):55–66, 1994.

[61] Le Pape, C. and Baptiste, P. Incorporating Efficient Operations Research Algorithms in Constraint-Based Scheduling. In *1st International Joint Workshop on Artificial Intelligence and Operational Research*, 1995.

[62] Lhomme, O. Consistency Techniques for Numeric CSPs. In *Proceedings of IJCAI'93*, volume 1, pages 232–238, 1993.

[63] Manne, A.S. On the Job-Shop Scheduling Problem. *Operations Research*, 8:219–223, 1960.

[64] Minton, S., Johnston, M.D., Philips, A.B., and Laird, P. Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method. In *Proceedings of AAAI'90*, volume 1, pages 17–24, 1990.

[65] Mohr, R. and Henderson, T.C. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[66] Mohr, R. and Masini, G. Good Old Discrete Relaxation. In *Proceedings of ECAI'88*, pages 651–656, 1988.

[67] Nakano, R. and Yamada, T. Conventional Genetic Algorithm for Job-Shop Problems. pages 474–479. Proceedings of the 4th International Conference on Genetic Algorithms and their Applications, San Diego, USA, 1991.

[68] Nowicki, E. and Smutnicki, C. A Fast Taboo Search Algorithm for the Job Shop Problem. *Management Science*, 42(6):797–813, 1996.

[69] Oliveira, E. and Smith, Barbara M. A Job-Shop Scheduling Model for the Single-Track Railway Scheduling Problem. pages 145–160. 19th Workshop of the UK Planning and Scheduling Special Interest Group – PlanSIG, 2000. Also available at `http:/www.comp.leeds.ac.uk/sacm`.

[70] Oliveira, E. and Smith, Barbara M. A Hybrid Constraint-Based Method for the Single-Track Railway Scheduling Problem. Technical Report 04, School of Computing, University of Leeds, February 2001.

[71] Pinedo, M. *Scheduling : Theory, Algorithms and Systems*. Prentice Hall, 1995.

[72] Prosser, P. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.

[73] Puget, J.-F. PECOS A High Level Constraint Programming Language. In *Proceedings of SPICIS'92*, September 1992.

[74] Puget, J.-F. A C++ Implementation of CLP. In *Proceedings of SPICIS'94*, November 1994.

[75] Puget, J.-F. A Fast Algorithm for the Bound Consistency of *alldiff* Constraint. In *Proceedings of AAAI'98*, 1998.

[76] Puget, J.-F. and Leconte, M. Beyond the Glass Box: Constraints as Objects. In *Proceedings of ILPS'95*, 1995.

[77] Régin, J.-C. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proceedings of AAAI'94*, pages 362–367, 1994.

[78] Reynolds, M. G. Computer Aided Railway Engineering. In B. Mellit, R. J. Hill, J. Allan, G. Sciutto, and C.A. Brebbia, editors, *Computers in Railways VI*, pages 149–157, Lisbon, 1998. Sixth International Conference on Computer Aided Design, Manufacture and Operation in the Railway and other Advanced Mass Transit Systems, WIT press – Computational Mechanics Publications. Railtrack plc – http://www.railtrack.co.uk.

[79] Robinson, J. A. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12:23–44, 1965.

[80] Sahin, I. Railway Traffic Control and Train Scheduling Based on Inter-Train Conflict Management. *Transportation Research*, 7(33):511–534, 1999.

[81] Singer, M. and Pinedo, M. A Computational Study of Branch and Bound Techniques for Minimizing the Total Weighted Tardiness in Job Shops . *IIE Scheduling and Logistics*, 30:109–118, 1997.

[82] Singer, M. and Pinedo, M. A Shifting Bottleneck Heuristics for Minimizing the Total Weighted Tardiness in Job Shops . *IIE Scheduling and Logistics*, 46(1):1–17, 1999.

[83] Smith, A.W. *Scheduling a Steelplant Using Constraint Programming.* PhD thesis, School of Computer Studies, University of Leeds, March 2000.

[84] Smith, Barbara M. and Grant, S.A. Trying Harder to Fail First. In *Proceedings of ECAI'98*, pages 249–253, 1998.

[85] Smolka, G. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[86] Szpigel, B. Optimal Train Scheduling on a Single Track Railway. In Roos, M., editor, *Proceedings of IFORS Conference on Operational Research'72*, pages 343–352, August 1972.

[87] Tsang, E. Applying Genetic Algorithms to Constraint Satisfaction Optimization Problems. In *Proceedings of ECAI'90*, pages 649–654, 1990.

[88] Tsang, E. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[89] Vaessens, R.J.M., Aarts, E.H.L., and Lenstra, J.K. Job Shop Scheduling by Local Search. *INFORMS Journal on Computing*, 8(3):302–317, 1996.

[90] van Hentenryck, P. and Saraswat, V. Constraint Programming: Strategic Directions. *Constraints: An International Journal*, 2(1):7–33, 1997.

[91] Wallace, M. Practical Applications of Constraint Programming. *Constraints: An International Journal*, 1(1):139–168, 1996.

[92] Walsh, T. Depth-bounded Discrepancy Search. In *Proceedings of IJCAI'97*, 1997.

[93] Watson, K. J. Computer Aided Railway Engineering. In B. Mellit, R. J. Hill, J. Allan, G. Sciutto, and C.A. Brebbia, editors, *Computers in Railways VI*, pages 93–102, Lisbon, 1998. Sixth International Conference on Computer Aided Design, Manufacture and Operation in the Railway and other Advanced Mass Transit Systems, WIT press – Computational Mechanics Publications. Railtrack plc – http://www.railtrack.co.uk.