DEA

INFORMATIQUE

ET

RECHERCHE OPERATIONNELLE

Resource Constraints for Preemptive and Non-Preemptive Scheduling

University of Paris VI

Institut Blaise Pascal

October 1995

Supervisor

Professor Philippe Chretienne

Philippe Baptiste

ILOG S.A.

9 rue de Verdun - BP 85

94253 Gentilly Cedex, France

E-mail : baptiste@ilog.fr

Contents

Abstract

ILOG, the company in which this thesis has been done, develops and markets ILOG SCHEDULE, a C++ library aimed at simplifying the representation and the resolution of scheduling problems. The SCHEDULE library is itself based on ILOG SOLVER, a generic software tool for object-oriented constraint programming.
Two main topics have been studied in the work presented here:

- **A theoretical study of constraint propagation algorithms for preemptive and non-preemptive scheduling:**

  1. ILOG SCHEDULE is currently limited to activities that execute without interruption and require the same resources from the beginning to the end of their execution. In this context, interruptible activities can be represented as sets of non-interruptible sub-activities. We show how to build a direct representation of interruptible activities and we describe several resource constraint propagation for preemptive scheduling.
  2. A theoretical comparison of two constraint propagation algorithms: The edge-finder algorithm and an algorithm based on energetic reasoning rules. This study led us to modify the edge-finder so that it incorporates some energy-based deduction rules.
  3. We also introduce a new characterization of the edge-finder and we show how this algorithm can be extended to become a "mixed" resource constraint propagation algorithm (i.e., a constraint propagation algorithm for resources required by interruptible and non-interruptible activities).

- **An experimental study of constraint propagation algorithms:**

  1. The study of an approximation algorithm for the Job-Shop Scheduling Problem. We present constraint-based optimization algorithms and a constraint-based approximation algorithm for the job shop scheduling problem. An empirical performance analysis shows that both the optimization algorithms and the approximation algorithm perform well. Especially the approximation algorithm is among the best algorithms known to date. We, furthermore, show that we can improve the performance of the optimization algorithms by combining them with the approximation algorithm. In particular, this combination allows us to solve an instance that was not yet reported to be solved.
  2. A computational study of the Preemptive Job-Shop Scheduling Problem. This led us to compare experimentally some constraint propagation algorithms for preemptive scheduling. This allowed us to solve some instances of the Preemptive Job-Shop Scheduling Problem including MT10, a well-known instance of the Job-Shop Scheduling Problem. To the best of our knowledge, the optimal makespan for these instances was not yet published anywhere.

Acknowledgments

This MSc thesis is the conclusion of a six-months training period at Ilog. This work has been very exciting and I want to thank the many people who helped me during this period. I especially think to Claude Le Pape, Sr Developer of Ilog Schedule, who advised me in all my work and whose excitment encourages me to carry on my studies in the scope of scheduling and more generally of combinatorial optimization. Thank you very much to Wim Nuijten, Developer of Ilog Schedule whose advices were always of great interest. I want also to thank Jean-Francois Puget Sr Developer of Ilog Solver and the all Solver and Schedule team Thank you very much to my supervisor, Professor Philippe Chretienne (Paris VI), and to the researchers of the DEA "Informatique et Recherche Opeŕationnelle".

Chapter 1

Introduction

Scheduling is the process of assigning activities to resources in time. Basically, the three main things to consider when building a scheduling system are:

- **The complexity of the scheduling problem.** Indeed, most scheduling problems are known to be NP-hard. Informally speaking, NP-hard problems are problems for which it is conjectured that there exists no algorithm enabling to optimally solve the problem in an amount of time bounded by a polynomial function of the size of the data. For a more precise definition, see for instance [Garey 79]. In practice, this means that one must design robust approximate algorithms, to generate appropriate (possibly optimal but often sub-optimal) solutions in a bounded amount of time.

- **The specificity of the problems to address.** Indeed, different manufacturing environments induce different scheduling constraints, some of which may be very specific to the problem under consideration.

- **The integration with the overall manufacturing system.** Indeed, a scheduling system must get its data from the information system globally in use in the factory, and must return its results (i.e., the constructed schedule) for factory-floor execution.

ILOG, the company in which this thesis has been done, develops and markets ILOG SCHEDULE, a C++ library aimed at simplifying the representation and the resolution of scheduling problems [Le Pape 94a]. The SCHEDULE library is itself based on SOLVER, a generic software tool for object-oriented constraint programming developed and marketed by ILOG [Puget 91] [Puget 92] [Puget 94] [Caseau 94a]. Each customer of ILOG SCHEDULE develops his or her own scheduling application, which implements the specific constraints and the specific problem-solving strategies that correspond to the type of manufacturing environment under consideration (see for instance [Le Pape 94b] [Baptiste 95a] for an example). Being based on the C++ programming language, the application is normally easy to integrate with databases and other interfaces required to operate the scheduling system.

The work presented here follows a previous training period at ILOG reported in [Baptiste 94], [Baptiste 95a], [Baptiste 95b], [Baptiste 95c]. In the following, we will often rely on this previous

work which include (1) a bibliographical study of constraint propagation algorithm for unary resources and (2) a description of a prototype for interruptible activities relying on a time-tabling mechanism. In the following report, two main topics have been studied:

- **A theoretical study of constraint propagation algorithms for preemptive and non-preemptive scheduling:**

  1. ILOG SCHEDULE is currently limited to activities that execute without interruption and require the same resources from the beginning to the end of their execution. In this context, interruptible activities can be represented as sets of non-interruptible sub-activities. We show how to build a direct representation of interruptible activities and we describe several resource constraint propagation for preemptive scheduling.
  2. A theoretical comparison of two constraint propagation algorithms: The edge-finder algorithm and an algorithm based on energetic reasoning rules. This study led us to modify the edge-finder so that it incorporates some energy-based deduction rules.
  3. We also introduce a new characterization of the edge-finder and we show how this algorithm can be extended to become a "mixed" resource constraint propagation algorithm (i.e., a constraint propagation algorithm for resources required by interruptible and non-interruptible activities).

- **An experimental study of constraint propagation algorithms:**

  1. The study of an approximation algorithm for the Job-Shop Scheduling Problem. We present constraint-based optimization algorithms and a constraint-based approximation algorithm for the job shop scheduling problem. An empirical performance analysis shows that both the optimization algorithms and the approximation algorithm perform well. Especially the approximation algorithm is among the best algorithms known to date. We, furthermore, show that we can improve the performance of the optimization algorithms by combining them with the approximation algorithm. In particular, this combination allows us to solve an instance that was not yet reported to be solved.
  2. A computational study of the Preemptive Job-Shop Scheduling Problem. This led us to compare experimentally some constraint propagation algorithms for preemptive scheduling. This allowed us to solve some instances of the Preemptive Job-Shop Scheduling Problem including MT10, a well-known instance of the Job-Shop Scheduling Problem. To the best of our knowledge, the optimal makespan for these instances was not yet published anywhere.

The remainder of this report is divided in 6 chapters. In chapter 2, we briefly present ILOG, and the two C++ libraries of interest to us, ILOG SOLVER and ILOG SCHEDULE. Chapter 3 presents a theoretical comparison of 2 constraint propagation algorithms and several flow-based algorithms for "preemptive" resource-constraints are detailed in chapter 4. The mixed

edge-finder is presented in chapter 5. Chapter 6 is devoted to an approximation algorithm for the Job-Shop Scheduling Problem, while chapter 7 presents a computational study of the Preemptive Job-Shop Scheduling problem.

Chapter 2

General Presentation

2.1 Presentation of the Company

ILOG was created in 1987 to industrialize the expertise of INRIA, Europe's largest computer research center in the field of symbolic computer languages and object-oriented environments. ILOG is gathering worldwide a team of over one hundred people, including approximatively eighty engineers. ILOG's Research and Development center, the department in which the work corresponding to this report has been done, operates with some forty high-skilled software engineers, including fifteen holders of doctorates in computer science.

ILOG develops and markets software development tools for object-oriented applications. The aim of these tools is to allow ILOG customers to design highly portable and maintainable code at a faster pace. These components are used to develop Graphical User Interfaces, decision support systems, and groupware applications.

- ILOG SOLVER is a C++ object-oriented constraint reasoning tool for solving highly combinatorial problems such as configuration, planning and scheduling.
- ILOG SCHEDULE is a C++ library added on ILOG SOLVER. Its aim is to ease the representation and resolution of scheduling and resource allocation problems.
- ILOG VIEWS is a C++ library of graphical objects dedicated to the development of standard graphical interfaces interfaces.
- ILOG BROKER is a C++ tool for developing client/server and distributed applications.
- ILOG SERVER is a C++ object server for building groupware applications. A notification mechanism ensures the consistency across multiple users of C++ objects.
- ILOG DB LINK is a C++ library to connect RDBMS such as Sybase, Oracle, Ingres, Informix ..., to C++ applications.
- ILOG RULES is a C++ tool for data monitoring in real time environments.

- ILOG TALK is an object-oriented dynamic language offering a seamless integration with C++ class libraries.

As said previously, ILOG SCHEDULE is a C++ library added on ILOG SOLVER. Thus, we will firstly describe ILOG SOLVER and the concept of "Constraint Programming". Afterwards, more details will be given on ILOG SCHEDULE.

2.2 Constraint Programming

2.2.1 Constraint Propagation

In this report, we refer to a constraint as a relation between the values of variables. For instance, if x is an integer variable, $x < 10$ is a constraint on the integer x. A constraint problem is made up of a set of variables and a set of constraints on these variables. Solving a constraint problem consists in finding for each variable a value that satisfies all the constraints posted on it.

The novelty of constraint programming lies in exploiting the constraints to reduce the amount of computation needed to solve the problem. The constraints are used to deduce new values and to detect impossibilities as rapidly as possible by reducing the domains. This deductive process is called **constraint propagation**.

For example, from $x < y$ and $x > 8$, we deduce, if x and y denote integers, that the value of y is at least 10. If later we add the constraint $y \leq 9$, a contradiction is immediately detected. Without propagation, the "$y \leq 9$" test could not be performed before the instantiation of y: no contradiction would be detected at this stage of the problem-solving process.

2.2.2 Backtracking

For complexity reasons, constraint propagation is usually **incomplete**. This means that some but not all the consequences of posted constraints are deduced. In particular, constraint propagation cannot detect all inconsistencies. Consequently, heuristic search algorithms must be implemented to explore possible refinements of the constraints (e.g., order any two activities that require the same unary resource) and exhibit solutions that are guaranteed to satisfy the constraints.

Backtracking is a mechanism used to implemement such algorithms: the algorithm is said to backtrack when it returns to a previous problem-solving state. ILOG SOLVER offers a few non-deterministic control primitives that allow its user to implement backtracking search algorithms.

2.3 ILOG SOLVER: An Overview

ILOG SOLVER is a C++ library for constraint-based programming. It includes predefined classes of variables (section 2.3.1), predefined classes of constraints together with a mechanism to implement new constraints (section 2.3.2), predefined search algorithms (section 2.3.3) and

non-deterministic control primitives that can be used to implement heuristic search algorithms (section 2.3.4).

2.3.1 Predefined Classes of Constrained Variables

In SOLVER, a logical variable is a C++ object. When the variable is not known (i.e., when it is not bound), a **domain** is associated to it. This domain represents the set of possible values for that variable. Constraint propagation reduces such domains by removing values that can be proven inconsistent with the constraints.

SOLVER provides several classes of variables:

- integer variables, whose domain is either an interval, or an enumeration of integers;
- enumerated variables, whose domain is a user-defined finite set of C++ object addresses;
- floating point variables, whose domain is an interval of floating point numbers;
- Boolean variables, whose domain is the set $\{0, 1\}$;
- set variables, whose domain is a set of sets. The value of a set variable is a finite set. When the variable is not bound, the domain is represented by its two bounds: the greatest lower bound (i.e., the intersection) of the possible values of the variable, and the least upper bound (i.e., the union) of the possible values of the variable. It is also possible to constrain the cardinality of such a variable.

According to C++ inheritance principles, it is possible to define new classes of variables that inherit from or combine variables of the above classes.

2.3.2 Constraints

Stating a constraint is done by a mere C++ function call. For some of the constraints, a C++ operator is also defined. Let us take the addition constraint $x = y + z$. The following are 3 equivalent ways to state this constraint:

```
IlcEqAdd(x, y, z);                  // relational form
IlcEq(x, IlcAdd(y, z));             // functional form
x == (y + z);                       // operator   form
```

Basic constraints include the following: $=$, $\neq$, $\leq$, $\geq$, $<$, $>$, $+$, $-$, $*$, $/$, subset, superset, union, intersection, member, boolean or, boolean and, boolean not, boolean xor. Some generic versions of these constraints are available, such as `IlcAllDiff` which states that all the variables in a given array are different, or `IlcArraySum` that returns a variable equal to the sum of the variables in a given array. Such generic constraints are useful since only one constraint is allocated, whatever the number of variable is.

The very important point is that ILOG SOLVER allows users to define their own constraints which are relevant to some special cases. A constraint may be defined by **predicates**, that

is, functions which test whether a set of given values satisfy a constraint, and **demons**, i.e., functions that reduce the domains of some variables when the domains of other variables change. When defining a new class of constraints, the user has access to the very mechanism used for predefined constraints.

2.3.3 Predefined Algorithms

To solve a constraint satisfaction problem, SOLVER uses a branch and bound algorithm. This may be done automatically by SOLVER which creates recursively choice points after a propagation phase. In case of failure, SOLVER backtracks on the last choice-point where one value at least is unexplored.

2.3.4 Search Control Primitives

ILOG SOLVER also provides a set of control primitives that allow its user to implement his/her own heuristic search algorithm. The main concept that is used is the concept of a non-deterministic goal which can be decomposed into a conjunction or a disjunction of sub-goals. For example, the following program states that to satisfy goal `f`, one must satisfy either the conjunction of goals `f1` and `f2` or the conjunction of goals `f3` and `f4`.

```
ILCGOAL0(f) {
  return IlcOr(IlcAnd(f1(), f2()),
               IlcAnd(f3(), f4()));
}
```

Goals may receive parameters as arguments. The syntax for calling a goal is identical to the syntax used to call regular C++ functions.

2.4 Scheduling with ILOG SCHEDULE

ILOG SCHEDULE is an "add-on" to ILOG SOLVER. It is a C++ library which allows users to represent scheduling problems and provides efficient propagation algorithms. This section, describes the main features of ILOG SCHEDULE. The semantics of the ILOG SCHEDULE model are very simple. Basically the elements of the model are the following ones: a **schedule** (section 2.4.1) which includes both **resources** (section 2.4.2) with limited capacity and **activities** (section 2.4.3).

This model includes also some predefined constraints:

- Temporal constraints which express precedence relationships between activities.
- Resource availability constraints which express the conditions under which a resource can be made available for use.
- Resource utilization constraints which specify how activities can use and share resources.

Section 2.4.4 presents an overview of the resource constraints available in ILOG SCHEDULE.

2.4.1 The Schedule

A schedule is represented by a local object that is an instance of the `IlcSchedule` class. The activities and the resources which are created are always reefering to an instance of the `IlcSchedule` class.

A `IlcSchedule` object has a time origin and a time horizon. These are used as defaults to initialize the earliest start time and the latest end time of activities.

2.4.2 Resources

ILOG SCHEDULE provides four different types of resources:

- `IlcDiscreteResource`: A `IlcDiscreteResource` represents a resource of discrete capacity. Capacity varies with time: at any time `t`, capacity represents the number of copies or instances of the resource that are available (e.g., the number of milling machines available in a manufacturing shop, the number of bricklayers at work on a construction site). "Discrete" means that capacity is defined to be a positive integer. Each activity may require (or provide) some amount (e.g., one milling machine, three bricklayers) of the resource capacity.

- `IlcUnaryResource`: A `IlcUnaryResource` represents a resource whose capacity is one.

- `IlcDiscreteEnergy`: A `IlcDiscreteEnergy` is similar to a `IlcDiscreteResource`, but its *energetic* capacity — as opposed to *instantaneous* capacity — is defined with respect to given time intervals (e.g., days, months, years) as the amount (e.g., in watt hours, in human-months) that can be made available over those intervals.

- `IlcStateResource`: A `IlcStateResource` represents a resource (a priori of infinite capacity) the state of which can vary over time. Each activity may, throughout its execution, require a state resource to be in a given state (or in any of a given set of states). Consequently, two activities may not overlap if throughout their execution they require incompatible states.

2.4.3 Activities

Activities are represented by instances, or combination of instances, of the `IlcIntervalActivity` class. An activity that executes without interruption from its start time to its end time, and uses the same resources from the beginning to the end of its execution, can be represented by a single instance of the `IlcIntervalActivity` class. Other type of activities may be defined by combining several such instances.

An activity is defined by its start time, end time, and duration. The values of these parameters may be unknown and are represented by the mean of constrained variables.

2.4.4 Resource Constraints

We need first to give a brief overview of the time-table mechanism used in some of the resource constraint propagation algorithms of ILOG SCHEDULE.

Time-Tables

ILOG SCHEDULE represents resources through time-tables of constrained variables. A time-table of constrained variables can be thought of as a table where each entry corresponds to a time unit, or a sequence of contiguous time units, and contains a constrained variable associated to that time unit or sequence of contiguous time units. Two types of time-tables are available:

- The first type assumes a discrete representation of time and memorizes the status of the variable (current value, current domain, current constraints) for each instant t in an interval $[a\ b)$. Information in the time-table is accessed in constant time, but the modification of the table from a date c to a date d (e.g., to reserve a resource for a given operation) requires time proportional to $d - c$. This type of implementation is particularly appropriate when the durations of operations are not much larger than the precision required in building the schedule.

- The second type does not make any assumption about the discrete or dense nature of time and memorizes the instants in time at which the status of the variable changes. Information in the time-table is accessed in time proportional to the number n of status changes (a slightly more complex implementation would allow an access time in $log(n)$) and a modification of the table from a date c to a date d requires time proportional to n. This type of implementation is particularly appropriate when there are very few operations to consider but the operations have to be positioned very precisely on the time-line.

These two implementations are referred to as **discrete array** and **sequential table**.

In addition to distinguishing discrete arrays and sequential tables, the user can specify that the value $v(t)$ is constant over intervals of a given size g, called the **grain** of the table. The default value of g is 1. The ratio between the schedule **horizon** (the duration of the overall time period to schedule) and the **grain** is a good indicator of the interest of a discrete array compared to a sequential table: when the ratio is small (e.g., a month in days), a discrete array is more appropriate; on the opposite, when the ratio is large (e.g., a month in seconds), a sequential table constitutes the best representation.

One of the main advantages of the underlying theoretical model is that it allows the management of discrete arrays and sequential tables of any type of variable: a discrete array of integer variables is built from a prototype integer variable; a sequential table of floating point variables is built from a prototype floating point variable; etc. In SCHEDULE:

- discrete arrays and sequential tables of integer variables are used to implement discrete resources;

- discrete arrays and sequential tables of Boolean variables are used to implement unary resources;
- discrete arrays and sequential tables of integer variables are used to implement energetic resources;
- discrete arrays and sequential tables of enumerated variables are used to implement state resources.

Discrete Resources Constraints

For discrete resources, there are two methods to take into account the constraints concerning the requirement or the provision of a resource.

- The first method allows capacity to vary over time: at any time t, a maximal capacity can be defined representing the number of instances of the resource that are available at t.
- The second method deals only with requiring (or providing) activities: it consists of posting a generic *disjunctive* constraint to insure that the theoretical capacity is satisfied. Furthermore, it insures that each *pair* of activities that require (or provide) a combined capacity that exceeds the capacity of the resource are not scheduled at the same time.

  When this second method is used, you can further increase the level of propagation: rather than considering only pairs of activities A_1, A_2 to prove that A_1 must precede A_2 or vice-versa, the constraint propagation process can consider arbitrary tuples $A_1, A_2, ..., A_n$ of activities to prove that some activity A_i must execute first (or must execute last) among $A_1, A_2, ..., A_n$.

Unary Resources Constraints

An instance of the `IlcUnaryResource` class represents a resource the capacity of which is one. There are three methods to take into account the constraints concerning the requirement or the provision of a unary resource.

- The first method specializes (for efficiency!) the method used for discrete resources. It allows capacity to vary with time: at any time `t`, the resource may or may not be in use. As there are only two possible values (in use or not), the discrete arrays and sequential tables that are used in the context of a `IlcUnaryResource` are arrays and tables of Boolean variables (`IlcBoolVar`).
- In contrast, the second method deals only with requiring (or providing) activities: it consists of posting a generic "disjunctive" constraint to ensure that the time intervals over which two activities require (or provide) the unary resource cannot overlap in time.[1]

[1] For instance, if a resource is required (or provided) by two activities throughout two time intervals `[ti1 ti2)` and `[tj1 tj2)`, the disjunctive constraint states that either `ti2` is less than or equal to `tj1`, or `tj2` is less than or equal to `ti1`.

No time-table needs to be created if this method is used.

- Last but not least, the third method is based on the edge-finder algorithm [Pinson 88] [Carlier 89], an extension of the disjunctive constraint.

The differences between these representations are highlighted below:

- The disjunctive and the edge-finder representation deal only with requiring (or providing) activities: to specify that the resource is not available over a given time interval, the user must create a "fake" activity that requires (or provides) the resource over that interval. The use of the disjunctive representation may therefore prove costly (in CPU time and memory space) if a big collection of "fake" activities is created.

- The disjunctive representation is, a priori, more CPU-time consuming, but the propagation of generic disjunctive constraints often results in more precise time-bounds than the propagation of the corresponding time-table constraints. In the context of a particular scheduling application, more CPU-time may be spent propagating the disjunctive constraints, but this extra propagation may result in a better exploration of the search space and, consequently, in a drastic improvement of the overall CPU time.

- When no "fake" activity is created, the disjunctive representation requires significantly less memory space than the time-table representation.

Hence the user of SCHEDULE can choose between three ways of propagating the constraints, knowing that the edge-finder propagates more than the disjunctive constraint which in turn propagates more than the time-table constraint. Depending on the application, the most cost-effective formulation may then be either (1) the time-table formulation, (2) the current disjunctive formulation or (3) the edge-finder formulation.

Chapter 3

A Theoretical Study of Two Constraint Propagation Algorithms for Disjunctive Scheduling

In the literature, many algorithms have been studied to provide interesting time-bounds for activities requiring the same unary resource. Three of them are available in ILOG SCHEDULE (cf. chapter 2). One of these methods consists in determining whether an activity A

- must,
- can,
- or cannot,

be the first (or the last) to execute among a set of activities S that require the same resource. Carlier and Pinson are well-known for their success with this technique. Among other impressive results, their edge-finder algorithm, was the first to exactly solve an instance of the job-shop scheduling problem introduced 25 years earlier by Fischer and Thomson [Muth 63] and known as MT10 [Carlier 89] [Carlier 90] [Pinson 88]. Similar ideas were since then used by Applegate and Cook [Applegate 91] and, in the form of a constraint propagation technique, by Nuijten, Aarts, van Erp Taalman Kip and van Hee [Nuijten 93] and by Caseau and Laburthe [Caseau 94b] and by Baptiste and Le Pape [Baptiste 95a].

An other interesting technique consists in comparing the amount of resource energy required over a time interval [*start end*) to the amount of energy that is available over the same interval. Many variants of this form of energetic propagation exist:

- Erschler, Lopez and Thuriot provide a number of "rules" that can potentially be used [Lopez 91] [Erschler 91].

- Beck defines a data structure called "habograph" to perform energetic propagation [Beck 92].
- ILOG SCHEDULE relies on this type of propagation in the case of energetic resources [Le Pape 94a].

There are basically two approaches to compare the edge-finder algorithm ([Carlier 89], [Carlier 90], [Pinson 88], [Nuijten 93]) and the algorithm based upon the energetic rules described in [Lopez 91], [Erschler 91]: An experimental and a theoretical approach. Since an experimental study has been carried in [Baptiste 94], we propose here to compare theoretically these algorithms. As a result of this study, we propose a slight modification of the edge-finder algorithm, which enables to incorporate some energy-based deduction rules. In seek of clarity, we recall the edge-finder algorithm (section 3.1) and the energy-based algorithm (section 3.2).

3.1 The Edge-Finder Algorithm

3.1.1 A Fundamental Proposition

Let us consider a set K of activities, subjected to both release and due-dates, to be sequenced on a single unary resource R (typically a machine). The following proposition holds:

Proposition 1

$$\max_{i \in K}(d_i) - \min_{i \in K}(r_i) \geq \sum_{i \in K}(p_i) \tag{3.1}$$

where r_i, d_i, and p_i respectively denote the release-date, the due-date, and the processing time (duration) of activity i.

Proof: This proposition states mathematically the fact that between the minimal release-date of the activities of K and the maximal due-date of the activities of K, there must be at least enough time to sequence all the activities of K.

3.1.2 Input and Output of a Clique

Definitions:

- A **clique** is a subset of the set of the activities to be scheduled on the same unary resource, that contains at least two elements.
- In a given solution, an activity is called the **input** of the clique if and only if all the other activities of the clique are scheduled after this one.
- In a given solution, an activity is called the **output** of the clique if and only if all the other activities of the clique are scheduled before this one.
- In a given solution, an activity is called a **middle** of the clique if and only if it is neither an input nor an output of the clique.

Let K be a clique of a unary resource R. [Pinson 88] proved the following propositions.

$$\forall k \in K, \left[\max_{i \in K-\{k\}} (d_i) - r_k < \sum_{i \in K} (p_i) \right] \Rightarrow [k \textit{ is not the input of } K]$$

$$\forall k \in K, \left[d_k - \min_{i \in K-\{k\}} (r_i) < \sum_{i \in K} (p_i) \right] \Rightarrow [k \textit{ is not the output of } K]$$

$$\forall k \in K, \left[\max_{i \in K-\{k\}} (d_i) - \min_{i \in K-\{k\}} (r_i) < \sum_{i \in K} (p_i) \right] \Rightarrow [k \textit{ is not a middle of } K]$$

These propositions are based upon the same principle as the "fundamental" one (equation 3.1). The first one, for instance, sets the fact that if a given activity is the input of the clique then, between its minimal completion time and the maximal due-date of all the activities, there must be enough time to schedule all the other activities. According to these propositions, we can make the following deductions:

$$\forall k \in K, \begin{cases} \max_{i \in K-\{k\}} (d_i) - \min_{i \in K-\{k\}} (r_i) < \sum_{i \in K} (p_i) \\ d_k - \min_{i \in K-\{k\}} (r_i) < \sum_{i \in K} (p_i) \end{cases} \Rightarrow [k \textit{ is the input}]$$

$$\forall k \in K, \begin{cases} \max_{i \in K-\{k\}} (d_i) - r_k < \sum_{i \in K} (p_i) \\ \max_{i \in K-\{k\}} (d_i) - \min_{i \in K-\{k\}} (r_i) < \sum_{i \in K} (p_i) \end{cases} \Rightarrow [k \textit{ is the output}]$$

$$\forall k \in K, \begin{cases} \max_{i \in K-\{k\}} (d_i) - r_k < \sum_{i \in K} (p_i) \\ d_k - \min_{i \in K-\{k\}} (r_i) < \sum_{i \in K} (p_i) \end{cases} \Rightarrow [k \textit{ is a middle}]$$

The proof of these propositions is obvious. For instance, the first proposition states that if an activity is neither an output nor a middle, then it is an input. These propositions allow us to deduce new temporal constraints and to update the release-dates as well as the due-dates:

1. If an activity is the input (or output) of a clique, then temporal constraints may be added between it and the other elements of the clique.

2. If an activity is a middle of the clique, then its earliest start time can be set to the minimal completion time of those of the other activities which can be first. Similarly, its latest end time can be set to the maximal start time of those of the other activities which can be last.

3. If an activity is not the input (output) of the clique then its earliest start time (latest end time) can be set to the minimal (maximal) completion time (start time) of those of the other activities which can be first (last).

This set of propositions known as the **edge-finding** technique, is powerful. An intuitive approach would be to test all these propositions for all cliques of a resource. The problem is that there are 2^n cliques therefore, this method would not be feasible for large scale scheduling problems. [Pinson 88] describes a method that performs all of the possible adjustments corresponding to case (1) in polynomial time. The next section summarizes the main results.

3.1.3 Jackson's Preemptive Schedule

Jackson's preemptive schedule is the preemptive schedule obtained by applying the rule reefered to as "Preemptive Earliest Due-Date rule" in [Morton 92]. This schedule minimizes the maximal lateness (and maximal tardiness) of n interruptible activities, subjected to release and due-dates constraints, and executing on a unique unary resource.[1] Morton and Pentico [Morton 92] describe the algorithm as follows:

1. Always schedule the activity with the earliest due-dates.
2. If an activity i becomes available while activity j is in process, stop activity j and start activity i if the due-dates of i is strictly smaller than the due-dates of j; otherwise continue activity j.

To determine whether it is sure that an activity A_i is scheduled after all the activities of a set K, Carlier and Pinson use the following algorithm:

1. Compute Jackson's preemptive schedule (JPS).
2. For each activity A_i, compute the set K of the activities which are not finished at $t = r_i$ on JPS. Let p_k^* be the residual duration on the preemptive schedule of the activity A_k at the time $t = r_i$. Take then the activities of K in decreasing order of due-dates and select the first activity A_j such that:

$$r_i + p_i + \sum_{A_k \in K - \{i\} / d_k \leq d_j} p_k^* > d_j \tag{3.2}$$

   If such an A_j exists then post the new temporal constraint:

$$\forall A_k \in K - \{i\} | d_k \leq d_j, A_k \prec A_i$$

   Which leads to:

$$r_i \geq \max_{A_k \in K - \{i\} / d_k \leq d_j} e_{JPS}(A_k) \tag{3.3}$$

   where $e_{JPS}(A_k)$ is the completion time of the activity A_k on Jackson Preemptive schedule.

For proof of this algorithm, see [Pinson 88] [Carlier 90]. Note that the symmetric version of this algorithm (JPS is computed backward: schedule as late as possible the activity of latest release-date...) will be reefered in the following as the dual version of the edge-finder.

[1] In particular, it is worth noting that when the maximal lateness obtained is strictly positive, it is impossible to schedule the activities without violating at least one constraint.

An example

Let us now detail an example taken from [Carlier 90]. The previous propositions are going to be applied on a one-machine problem where six tasks have to be scheduled:

A ONE-MACHINE PROBLEM			
Activity	duration	release-date	due-date
A	6	4	32
B	8	0	27
C	4	9	22
D	5	15	43
E	8	20	38
F	8	21	36

Applying the JPS algorithm, the following result is obtained:

```
BBBBBBBBACCCCAAAAADDEFFFFFFFFEEEEEEEDDD
```

Let us apply the algorithm at the step where the activity D is concerned. The following sums are computed:

$$r_D + p_D + p_A^* + p_E^* + p_F^* = 15 + 5 + 3 + 8 + 8 = 39$$

$$\max(d_A, d_E, d_F) = \max(32, 38, 36) = 38$$

We are then in the case where:

$$r_D + p_D + p_A^* + p_E^* + p_F^* > \max(d_A + d_E + d_F)$$

Therefore we can update the release-date of the activity D by setting

$$r_D = \max(C_A^{JPS}, C_E^{JPS}, C_F^{JPS}) = 36$$

We can then recompute JPS:

```
BBBBBBBBACCCCAAAAA--EFFFFFFFFEEEEEEEDDDDD
```

And iterate...

3.2 Energetic Reasoning

The second type of method consists in comparing the amount of resource energy required over a time interval [*start end*) to the amount of energy that is available over the same interval.

In [Erschler 91] and in [Lopez 91], the authors analyse the effect of time and resource constraints on the admissibility of schedules. They study how activity characteristics and discrete resource constraints may induce new constraints which allow to restart the propagation. Although Lopez and Erschler have worked on discrete resources, our study will focus on unary resources. Let us define the concept of **required energy consumption** $W_A^{[t_1\ t_2)}$ of an activity

A over an interval $[t_1\ t_2)$. This is actually the smallest amount of time during which A will be executed on the interval.

$$W_A^{[t_1,t_2)} = \max(0, \min(p_A, t_2 - t_1, r_A + p_A - t_1, t_2 - d_A + p_A))$$

Let us now describe two important deduction rules:

1. The first deduction rule is based upon the idea of picking two activities A, B and to try to find a contradiction when sequencing A before B. To achieve this, the required consumption of all the activities over the interval $[r_A\ d_B)$ is computed and compared to the provided amount of resource during the same interval.

$$\left[d_B - r_A < \sum_{\alpha \notin \{A,B\}} W_\alpha^{[r_A, d_B)} + p_A + p_B \right] \Rightarrow [B \prec A] \tag{3.4}$$

2. The aim of the second deduction rule is to update straightly the release-dates and due-dates of the activities. Let us choose an activity A and an integer x in the interval $[r_A\ d_A)$. We are going to check on the interval $[r_A\ x)$ whether A can or cannot start at its earliest start-time. If it cannot, then r_A will be increased. Let us first define the quantity $TotalW$.

$$TotalW = \sum_{\alpha \neq A} W_\alpha^{[r_A, x)} + \min(p_A, x - r_A)$$

The deduction rule is then:

$$[x - r_A < TotalW] \Rightarrow \left[r_A = r_A + \sum_{\alpha \neq A} W_\alpha^{[r_A, x)} \right] \tag{3.5}$$

In fact, Erschler, Lopez and Thuriot provide $x + TotalW - (x - r_A)$ as a bound for updating the earliest end time of A. For a non-interruptible activity with fixed duration, this leads to $r_A + \sum_{\alpha \neq A} W_\alpha^{[r_A, x)} + \min(p_A, x - r_A) - p_A$ as a new value for r_A. The rationale for adopting this bound is that a minimal energy of $TotalW - (x - r_A)$ must be flushed out of the $[r_A\ x)$ interval. However, when $x - r_A$ is smaller than p_A, the $p_A - (x - r_A)$ energy is already consumed out of the $[r_A\ x)$ interval. Hence, it is correct to replace $x + TotalW - (x - r_A)$ by $x + \sum_{\alpha \neq A} W_\alpha^{[r_A, x)} + p_A - (x - r_A)$. This leads to the bound given above.

Obviously, there are many values of x for which this rule could be used. Erschler, Lopez and Thuriot suggest that the appropriate values of x are the earliest and latest start and end times of activities (which seems reasonable but is yet to be formally proven).

3.3 Edge-Finder does not Subsume Energetic Rule

The small example described bellow illustrates that the edge-finder algorithm does not subsume the "first" Energetic rule described in [Lopez 91]. Indeed, time-bounds deduced by the edge-finder may be weaker than those deduced of the application of energetic rules.

A ONE-MACHINE PROBLEM			
Activity	duration	release-date	due-date
A	2	1	10
B	2	0	5
C	1	2	5

- **Edge Finder Algorithm**

  Let us build the Jackson Preemptive Schedule:

```
0 1 2 3 4 (time)
B B C A A
```

  Let us apply the edge-finder algorithm. Table 3.1 details the six tests that are computed by the edge-finder. The logical value located at column i, line j corresponds to:

$$r_i + p_i + \sum_{A_k \in K - \{i\} / d_k \leq d_j} p_k^* > d_j \tag{3.6}$$

  Note that an automatic adjustment is achieved if and only if at least one of the tests described in tables 3.1 and 3.2 is true.

EDGE-FINDER TESTS (primal version)			
Activity	A	B	C
	$r_A = 1,\ p_A = 2$ $p_B^* = 1,\ p_C^* = 1$	$r_B = 0,\ p_B = 2$ $p_A^* = 2,\ p_C^* = 1$	$r_C = 2,\ p_C = 1$ $p_A^* = 2,\ p_B^* = 0$
A $d_A = 10$	–	$0 + 2 + 1 + 2 = 5 > 10$? No deduction	$2 + 1 + 2 = 5 > 10$? No deduction
B $d_B = 5$	$1 + 2 + 1 + 1 = 5 > 5$? No deduction	–	$2 + 1 = 3 > 5$? No deduction
C $d_C = 5$	$1 + 2 + 1 + 1 = 5 > 5$? No deduction	$0 + 2 + 1 = 3 > 5$? No deduction	–

Table 3.1: Primal Edge-Finder

  Let us also verify that the dual version of the edge-finder does not deduce anything either. The dual JPS is:

```
0 1 2 3 4 5 6 7 8 9 (time)
- - B B C - - - A A
```

  The following table sums-up the six tests that are computed by the edge-finder. The logical value located at column i, line j corresponds to:

$$d_i - p_i - \sum_{A_k \in K-\{i\}/r_k \geq r_j} p_k^* < r_j$$

EDGE-FINDER TESTS (dual version)			
Activity	A	B	C
	$d_A = 10,\ p_A = 2$ $p_B^* = 2,\ p_C^* = 1$	$d_B = 5,\ p_B = 2$ $p_A^* = 0,\ p_C^* = 1$	$d_C = 5,\ p_C = 1$ $p_A^* = 0,\ p_B^* = 2$
A $r_A = 1$	–	$5-2-1=2<1$? No deduction	$5-1-0=4<1$? No deduction
B $r_B = 0$	$10-2-2-1=5<0$? No deduction	–	$5-1-2=2<0$? No deduction
C $r_C = 2$	$10-2-1=7<2$? No deduction	$5-2-1=2<2$? No deduction	–

Table 3.2: Dual Edge-Finder

- **Energetic Rule**

  Let us now focus on the first energetic rule 3.4. The value of the required energy consumption is:

$$\sum_{\alpha \notin \{A,B\}} W_\alpha^{[r_A, d_B)} = W_C^{[1,5)} = 1$$

  Consequently, B is before A because the left member of 3.4 holds.

$$[5-1 < 1+2+2] \Rightarrow [B \prec A]$$

  The time-bound adjustment is then $r_A = 2$.

This example has shown us that the edge-finder algorithm does not subsume the first "energetic" rule of [Lopez 91]. However, this rule does not either subsume the edge-finder algorithm: We propose in the next section an other example which illustrates this.

3.4 Energetic Rule does not Subsume Edge-Finder

Let us consider the following instance of the one-machine problem. And let us apply the edge-finder and the energetic rules:

A ONE-MACHINE PROBLEM			
Activity	duration	release-date	due-date
A	6	0	17
B	4	1	11
C	3	1	11

Let us now apply the two algorithms:

- **Edge Finder Algorithm**

  The Jackson Preemptive Schedule corresponding to this instance is:

```
0 1 2 3 4 5 6 7 8 9 10  11
A B B B B C C C A A A A A
```

  The edge-finder is able to deduce that the activity A cannot execute before the date 8. Indeed, the logical test (corresponding to equation 3.6) applied to activities A, C is true and consequently an adjustment is achieved:

$$[0 + 6 + 4 + 3 > 11] \Rightarrow [r_A \geq 8]$$

- **Energetic rule**

  Let us show that the propagation algorithm relying on the energetic rules (equations 3.4 3.5) is unable to deduce the new time-bound $r_A = 8$. In the following table, each cell (column i, line j) stands for the logical test which triggers any updating (cf. 3.4).

FIRST ENERGETIC RULE			
Activity	A $r_A = 0$	B $r_B = 1$	C $r_C = 1$
A $d_A = 17$	– –	$4 + 6 + 3 > 17 - 1$? No deduction	$3 + 6 + 4 > 17 - 1$? No deduction
B $d_B = 11$	$6 + 4 + 3 > 11 - 0$? $A \not\prec B$	– –	$3 + 4 + 0 > 11 - 1$? No deduction
C $d_C = 11$	$6 + 3 + 4 > 11 - 0$? $A \not\prec C$	$4 + 3 + 0 > 11 - 1$? No deduction	– –

The deductions of the first energetic rule are then $B \prec A$ and $C \prec A$; which leads to the adjustment $r_A = 5$. Let us apply again the same energetic rule with this new time bound:

FIRST ENERGETIC RULE			
Activity	A $r_A = 5$	B $r_B = 1$	C $r_C = 1$
A $d_A = 17$	– –	$4 + 6 + 3 > 17 - 1$? No deduction	$3 + 6 + 4 > 17 - 1$? No deduction
B $d_B = 11$	$6 + 4 + 0 > 11 - 5$? $A \not\prec B$	– –	$3 + 4 + 0 > 11 - 1$? No deduction
C $d_C = 11$	$6 + 3 + 0 > 11 - 5$? $A \not\prec C$	$4 + 3 + 0 > 11 - 1$? No deduction	– –

We can see that the first energetic rule cannot deduce more than $B \prec A$ and $C \prec A$. Let us now apply the second energetic rule forward and backward. In the following tables,

each cell (column i, line j) stands for the logical test which triggers any updating (cf. 3.5).

SECOND ENERGETIC RULE (forward)			
Activity	A	B	C
	$r_A = 5$	$r_B = 1$	$r_C = 1$
A	$0 + \min(6, 17-5) > 17-5$?	$9 + \min(4, 17-1) > 17-1$?	$10 + \min(3, 17-1) > 17-1$?
$d_A = 17$	No deduction	No deduction	No deduction
B	$0 + \min(6, 11-5) > 11-5$?	$3 + \min(4, 11-1) > 11-1$?	$4 + \min(3, 11-1) > 11-1$?
$d_B = 11$	No deduction	No deduction	No deduction
C	$0 + \min(6, 11-5) > 11-5$?	$3 + \min(4, 11-1) > 11-1$?	$4 + \min(3, 11-1) > 11-1$?
$d_C = 11$	No deduction	No deduction	No deduction

SECOND ENERGETIC RULE (backward)			
Activity	A	B	C
	$d_A = 17$	$d_B = 11$	$d_C = 11$
A	$0 + \min(6, 17-5) > 17-5$?	$0 + \min(4, 11-5) > 11-5$?	$0 + \min(3, 11-5) > 11-5$?
$r_A = 5$	No deduction	No deduction	No deduction
B	$7 + \min(6, 17-1) > 17-1$?	$3 + \min(4, 11-1) > 11-1$?	$4 + \min(3, 11-1) > 11-1$?
$r_B = 1$	No deduction	No deduction	No deduction
C	$7 + \min(6, 17-1) > 17-1$?	$3 + \min(4, 11-1) > 11-1$?	$4 + \min(3, 11-1) > 11-1$?
$r_C = 1$	No deduction	No deduction	No deduction

The deduction of the first and the second energetic rules, is then $r_A = 5$; which illustrates the fact that the energetic rules do not subsume the edge-finder. It is interesting to notice that the energetic rules do not commute. Indeed, let us change the order in which we apply the deduction rules: Apply (1) the second energetic rule and (2) the first energetic rule.

SECOND ENERGETIC RULE (forward)			
Activity	A	B	C
	$r_A = 0$	$r_B = 1$	$r_C = 1$
A	$7 + \min(6, 17-0) > 17-0$?	$8 + \min(4, 17-1) > 17-1$?	$9 + \min(3, 17-1) > 17-1$?
$d_A = 17$	No deduction	No deduction	No deduction
B	$7 + \min(6, 11-0) > 11-0$?	$3 + \min(4, 11-1) > 11-1$?	$4 + \min(3, 11-1) > 11-1$?
$d_B = 11$	$r_A \geq 7$	No deduction	No deduction
C	$7 + \min(6, 11-0) > 11-0$?	$3 + \min(4, 11-1) > 11-1$?	$4 + \min(3, 11-1) > 11-1$?
$d_C = 11$	$r_A \geq 7$	No deduction	No deduction

The second energetic rule applied forward deduces that $r_A \geq 7$! Let us verify that the second energetic rule applied backward and that the first energetic rule do no deduce anything else:

SECOND ENERGETIC RULE (backward)			
Activity	A	B	C
	$d_A = 17$	$d_B = 11$	$d_C = 11$
A	$0 + \min(6, 17-7) > 17-7$?	$0 + \min(4, 11-7) > 11-7$?	$0 + \min(3, 11-7) > 11-7$?
$r_A = 7$	No deduction	No deduction	No deduction
B	$7 + \min(6, 17-1) > 17-1$?	$3 + \min(4, 11-1) > 11-1$?	$4 + \min(3, 11-1) > 11-1$?
$r_B = 1$	No deduction	No deduction	No deduction
C	$7 + \min(6, 17-1) > 17-1$?	$3 + \min(4, 11-1) > 11-1$?	$4 + \min(3, 11-1) > 11-1$?
$r_C = 1$	No deduction	No deduction	No deduction

FIRST ENERGETIC RULE			
Activity	A	B	C
	$r_A = 7$	$r_B = 1$	$r_C = 1$
A	–	$4+6+3 > 17-1$?	$3+6+4 > 17-1$?
$d_A = 17$	–	No deduction	No deduction
B	$6+4+0 > 11-7$?	–	$3+4+0 > 11-1$?
$d_B = 11$	$A \not\prec B$	–	No deduction
C	$6+3+0 > 11-7$?	$4+3+0 > 11-1$?	–
$d_C = 11$	$A \not\prec C$	No deduction	–

The following table is a summary of the results presented in this section. In this table, EF stands for Edge-Finder, ER1 stands for Energetic Rule 1 and ER2 stands for Energetic Rule 2. We use the notation $EF \not\rhd ER1$ to state that the edge-finder does not subsume the first energetic rule.

Summary
$EF \not\rhd ER1$
$ER1 \not\rhd EF$
$ER2 \not\rhd EF$
$ER1 + ER2 \not\rhd EF$
ER1 and ER2 do not commute

3.5 An Energetic Edge-Finder

The examples described in sections 3.4 and 3.3 led us to incorporate an energy-based deduction rule to the edge-finder algorithm. From now on the term "energetic edge-finder" will denote the resulting algorithm. We claim that the complexity of the energetic edge-finder is quadratic and that it allows to deduce more than the edge-finder.

3.5.1 Basic Ideas

To simplify the presentation of our algorithm, we consider in this section that given two activities A_i and A_j requiring the same resource, the due-date of A_i and the due-date of A_j are different (which can be done by adding $i * \epsilon$ to each activity's due-date d_i; where $\epsilon \ll 1$).

Let us recall 3.2, the fundamental edge-finding test.

$$r_i + p_i + \sum_{A_k \in K - \{i\} / d_k \leq d_j} p_k^* > d_j$$

Two cases may occur:

1. **Either** this test succeeds and then, the activity A_i has to be executed after all the other activities of K, which leads to the time-bounds adjustments 3.3.
2. **Or** this test fails and the algorithm tries to find an other activity A_j which will make 3.2 succeed.

Let us now focus on case 2: None of the activities A_j has triggered a time-bound adjustment. Intuitively, it seems interesting to test if at least, a deduction (of course less powerful than 3.3) like $A_j \prec A_i$ can be made.

Proposition 2

$$\left[r_i + p_i + \sum_{A_k \in K - \{i\} / d_k < d_j k \neq j} p_k^* + p_j > d_j \right] \Rightarrow [A_j \prec A_i] \tag{3.7}$$

Proof: Let us suppose that there exists Ω, a schedule of activities $A_1, \cdots, A_n$ meeting release-dates and due-dates, such that A_i is scheduled before A_j on Ω. Let us then consider a new set of activities $A_1^{'}, \cdots, A_n^{'}$ whose release-dates ($r_k^{'}$), due-dates ($d_k^{'}$) and processing times ($p_k^{'}$) are the same as those of $A_1, \cdots, A_n$ except for the release-date of $A_j^{'}$ which is $r_j^{'} = \max(r_j, r_i + p_i)$.

On Ω, A_i is scheduled before A_j then A_j cannot start before $r_i + p_i$. Ω is then meeting release-dates and due-dates of this new problem. Moreover, $A_i^{'}$ is scheduled before $A_j^{'}$ then, according to 3.2, the following inequality holds (otherwise the edge-finder would deduce that $A_i^{'} \prec A_j^{'}$):

$$r_i^{'} + p_i^{'} + \sum_{A_k^{'} / d_k^{'} \leq d_j^{'}} p_k^{'*} \leq d_j^{'} \tag{3.8}$$

Let us then compute the values of $p_k^{'*}$. Two cases have to be distinguished:

- **Either** $k \neq j$ and then A_k is more urgent than A_j (its due-date is strictly smaller). Since the modification of the release-date of A_j does not modify the dates at which the activities whose due-dates are strictly smaller than d_k, the following equality holds:

$$p_k^{'*} = p_k^*$$

- **Or** $k = j$. In such a case, the value of $p_j^{'*}$ must take into account the fact that A_j cannot start before $r_i + p_i$; which means that $p_j^{'*} = p_j$.

Equation 3.8 now becomes:

$$r_i + p_i + \sum_{A_k / d_k < d_j} p_k^* + p_j \leq d_j^{'}$$

Absurd since equation 3.7 holds.

3.5.2 Description of the Energetic Edge-Finder

The results described in section 3.5.1 led us to study a resource constraint propagation algorithm combining (1) the edge-finder and (2) the deduction rule 3.7. The primal version of the resulting algorithm can be written as follow:

1. Sort activities in increasing order of due-dates (we suppose in the remaining part of this algorithm that the activities $A_1, A_2, \cdots, A_n$ are sorted in increasing order of due-dates).
2. Build the JPS of $A_1, A_2, \cdots, A_n$.
3. Iterate on activity A_i for $i = 1$ to $i = n$:
   (a) Compute the values of p_j^* (for $j = 1$ to n).
   (b) Compute $\Psi = \sum_k p_k^*$
   (c) Iterate on activity A_j for $j = 1$ to n:
      i. Update the value of Ψ to $\Psi - p_j^*$.
      ii. Perform the basic edge-finding test (3.2).

$$r_i + p_i + \Psi > d_j \tag{3.9}$$

      If it succeeds, update the value of r_i according to 3.3:
      iii. If (3.9) fails perform the following test

$$r_i + p_i + \Psi + p_j > d_j$$

      In case of success, update r_i to $\max(r_i, r_j + p_j)$

   End Iterate

End Iterate

Proposition 3 *The energetic edge-finder runs in $O(n^2)$.*

Proof: Step (1) runs in $O(n * \log(n))$, as well as step (2) (see for instance [Carlier 84]). Step (3) consists of one inner loop and one outer loop; which leads to a quadratic complexity. The overall theoretical complexity of the algorithm is then $O(n^2)$.

3.5.3 An Example

The one-machine problem used as an example in section 3.3 shows that the edge-finder algorithm does not subsume the first energetic rule 3.4. The energetic edge-finder algorithm described in section 3.5.2 is able to make (al least) some energetic deductions. Indeed, applying it to the example 3.3, allows us to update the release-date of activity A, which could not be achieved by the edge-finder.

A ONE-MACHINE PROBLEM			
Activity	duration	release-date	due-date
A	2	1	10
B	2	0	5
C	1	2	5

The primal (JPS) associated to this instance is:

```
0 1 2 3 4 (time)
B B C A A
```

Applying 3.7 with $A_i = A$ and $A_j = B$ leads to test whether the following equation holds:

$$1 + 2 + \sum_{A_k \in K - i,j \,/\, d_k \leq 2} p_k^* + 2 > 5$$

Which is true since $1+2+1+2 > 5$. This leads to $B \prec A$, which in turn allows to update the release-date of A to $r_B + P_B$: $r_A = 2$. This adjustment was not achieved by the edge-finder.

3.6 Conclusion

The energetic-edge-finder proposed in the previous section is an extension of the edge-finders algorithm. It seems interesting because its theoretical complexity is $O(n^2)$ the same as the complexity of the "classical" version of the edge-finder (a more sophisticated edge-finder running in $O(n * \log n$ is detailed in [Carlier 94]). However, one can ask several questions about this algorithm:

- The energetic edge-finder is theoretically more efficient than the edge-finder. However, as we did not find time to make any experiment, it is difficult to see how efficient, in practise, this energetic-edge-finer is.
- The edge-finder respects "fixpoint semantics" (In general, a set of propagation rules is said to respect "fixpoint semantics" if the results of the overall propagation do not depend on the order in which the propagation steps are executed) but it is not clear whether the energetic-edge-finder does.
- It is also difficult to order in terms of theoretical efficiency the energetic-edge-finder and the algorithm based on the energetic rules 3.4 and 3.5. One open question is "May the algorithm based on 3.4 still outperform the energetic-edge-finder?". Intuitively, it seems to us that the answer is yes.

Chapter 4

Interruptible Activities and Preemptive Resource Constraints

ILOG SCHEDULE is currently limited to activities that execute without interruption and require the same resources from the beginning to the end of their execution. Interruptible activities can be modeled as sets of non-interruptible sub-activities. The user who requires interruptible activities has to implement such a model. In this chapter, we discuss the feasibility of a direct representation and describe a prototype implementation of interruptible activities in ILOG SCHEDULE. This work follows [Baptiste 94] which describes a prototype for interruptible activities relying on a time-tabling mechanism. The approach described in this chapter is very different from the previous one since it relies on Operations Research's techniques.

Section 4.1, presents some of the reasons which have motivated this work and enumerates the functionalities that would be worth offering in a future version of ILOG SCHEDULE. Section 4.2 describes (1) a modelization of interruptible activities and (2) some basic constraints that ensure the internal consistency of this model. The remaining sections (4.3 and 4.4) are dedicated to the propagation algorithms which have been developed to represent unary resources and discrete resources.

4.1 Functionalities

It is interesting to deal with interruptible activities for two main reasons. On the one hand, they would be very useful for modeling some scheduling problems. In particular, it has been noticed that ILOG customers could benefit from interruptible activities in the following cases:

- **To model preemption.** There are a few applications for which it is actually possible to start an activity A, interrupt it in favor of another activity B, and restart A after B is finished.

- **To model shifts.** In a factory, it often occurs that different machines operate according to different shifts (e.g., one ten-hour shift for machine M_1, two eight-hour shifts for machine M_2). Depending on the type of product being manufactured, it may or may not be possible to start an operation during one shift and end it during another shift. Interruptible activities could simplify the representation of such situations.

- **To model periodic or pseudo-periodic activities.** For example, it can be the case that a particular piece of equipment must be regularly maintained. To represent such a maintenance constraint with non-interruptible activities, one must determine the maximal number n of maintenance periods and create an activity for each of these periods. The same constraint could alternatively be represented with a unique interruptible activity representing the overall maintenance task. A similar situation occurs when allocating tasks to individual people each of which is assumed to rest on given days each week. On one particular application, ILOG has evaluated that interruptible activities could result in sparing up to 8 Megabytes of memory!

- **To model breaks.** For example, [Le Pape 94b] presents an application in which moulding operations can be interrupted once for a break (e.g., a lunch break) but only under certain conditions. To model the existing rules with the current version of SCHEDULE, it has been necessary to create up to four non-interruptible activities for each operation. The overall scheduling model would have been much simpler and probably less space-consuming if interruptible activities had been available.

On the other hand, a preemptive problem may often constitute a tractable relaxation of a non-preemptive problem. For instance, the one-machine problem (i.e., the problem of sequencing n independent tasks subjected to release-dates and due-dates on a unique resource) is NP-hard [Garey 79] in the non-preemptive case. The same problem can be solved in polynomial time when preemptions are allowed [Garey 79]. Hence, interruptible activities could be used to easily compute a lower bound for the optimal cost of a non-preemptive problem. As shown in [Pinson 88] and [Carlier 90], interruptible activities could also be used to deduce characteristics of non-preemptive problem solutions.

To represent efficiently the previously described problems, many functionalities should be integrated in a future version of ILOG SCHEDULE. The functionalities described below seem to be quite exhaustive and in the developed prototype, some of them only have been implemented. It shall be possible:

1. As for non-interruptible activities, to constrain the earliest start time (StartMin), earliest end-time (EndMin), latest start-time (StartMax), latest end-time (EndMax), minimal processing time (DurationMin), and maximal processing time (DurationMax), of an interruptible activity.

2. To force an interruptible activity to execute during some intervals chosen by the user (the "required intervals").

3. To force an interruptible activity not to execute during some intervals chosen by the user (the "forbidden intervals").

4. To constrain the duration of an interruption to be lower than (or equal to) a given value (InterruptionDurationMax) and greater than (or equal to) another value (InterruptionDurationMin). For example, it may not be acceptable to stop a production line for less than half an hour.

5. To constrain the duration of each execution interval (between two interruptions) to be lower than (or equal to) a given value (ExecutionPeriodDurationMax) and greater than (or equal to) another value (ExecutionPeriodDurationMin).

6. To constrain the number of interruptions to be lower than (or equal to) a given value (InterruptionNumberMax) and greater than (or equal to) another value (InterruptionNumberMin). For example, it may be considered unacceptable to interrupt more than twice the production of a batch of parts.

7. To decide whether during interruptions, the resource can be active or not. In some cases, a manufacturing operation may be stopped only for a break but not to let another operation execute on the same resource.

Only points 1, 2, 3 and 4 have been dealt with in the prototype. Note that some of the functionalities above can already be implemented using the current version. For example, point 3 can be implemented by adding a new (artificial) resource which is (a) required by the activity and (b) not available over the forbidden intervals. It is obvious that a direct representation is more convenient and more efficient.

4.2 Interruptible Activities

A non-interruptible activity can be defined by three SOLVER variables: *start*, *end* and *duration*. One "internal" constraint is posted to ensure that:

$$start + duration = end$$

This model does not suit to Interruptible activities. Indeed, the three variables *start*, *end* and *duration* still make sense but, such a model does not allow the user to state that, for instance, a given activity cannot execute in a given time interval. Sections 4.2.1 and 4.2.2 describe respectively a generic model for interruptible activities and, "internal" constraints which ensure the "internal" consistency of such a model.

4.2.1 A Generic Model: Required Intervals and Forbidden Intervals

A very general model for interruptible activities consists in associating to each activity A_i:

- a SOLVER variable p_i which represents the duration of the activity (note that, in the following, p_i often denotes the minimal value of the duration variable).

- A set $R_i = R_i^1, R_i^2, \cdots, R_i^{req_i}$ of **required** time-intervals i.e., time-intervals over which the activity is required to execute (req_i denotes the number of non-overlapping time-intervals in R_i).
- A set $F_i = F_i^1, F_i^2, \cdots, F_i^{forb_i}$ of **forbidden intervals** i.e., time-intervals over which the activity is not executed ($forb_i$ denotes the number of non-overlapping time-intervals in F_i).

This model enables a straight decision-making process: an activity can be required or "forbidden" to execute in a given time-interval.

Definition Given an interval I, let $s(I)$ and $e(I)$ be respectively the start and the end of I (i.e., $I = [s(I)\ e(I))$.

4.2.2 Internal Constraints

Some constraints have to be posted on each activity. Indeed, the constraint propagation process must ensure that, for a given activity A_i,

1. at least the minimal duration of the activity is available between its for bidden intervals. This can be written as follow:
$$\sum_{k=1}^{k<req_i} \left(s(R_i^{k+1}) - e(R_i^k)\right) \geq p_i$$
2. R_i and F_i do not overlap.

Constraining Time-Bounds

Interruptible activities as described in section 4.2.1 do not allow to deal straightly with time-bounds of activities. This led us to add to this model, two solver variables per activity: the start and end variables. To guarantee the consistency of these extra-variables with the model described in section 4.2.1, some "internal"constraints have to be posted on each interruptible activity. The number of events that may occur (modification of a time-bound, addition of an extra-required interval ...) is large but the constraint propagation methods that handle them are similar. We only describe two of them.

- When a new required interval $[a\ b)$ is added to the set R_i of A_i, the constraint propagation algorithm must (1) check that $[a\ b)$ does not overlap with any forbidden intervals of F_i and (2) update the value of the maximal start-time of A_i to a (if the current maximal start-time is greater than a), and (3) update the value of the minimal end-time of A_i to b (if the current minimal end-time is lower than b).
- When the minimal start-time of a given activity increases, the set of forbidden intervals has to be updated.
- ...

Constraining Interruptions and Executions

As said in section 4.1, it seems interesting to allow the user of ILOG SCHEDULE to constrain the duration of each execution interval (between two interruptions) to be (1) lower than a given value and (2) greater than another value. However the current prototype is limited to point (2). Notice that the symmetric constraint (1) as well as constraints dealing with the durations of interruptions can be handled by the same type of constraint propagation algorithms.

Let $minExec_i$ be the minimal duration of each execution interval for a given activity A_i. Let R_i^k be a required interval of A_i, whose size is lower than $minExec_i$ and let F_i^l be the forbidden time-interval immediately before R_i^k (i.e. $F_i^l \prec R_i^k \prec F_i^{l+1}$). The basic ideas of the propagation algorithm can be summarized as follow:

- **Proposition 4** *If $e(F_i^{l+1}) - s(F_i^l) < minExec_i$, a failure is triggered.*

  Proof: Indeed, A_i has to be executed between F_i^l and F_i^{l+1} but, the maximal available duration between these two forbidden intervals is strictly lower than $minExec_i$.

- **Proposition 5** *If $s(F_i^l) + minExec_i > e(R_i^k)$ then the required interval R_i^k has to be enlarged to $[s(R_i^k), s(F_i^l) + minExec_i)$.*

  Proof: Let Ω be a preemptive schedule meeting (1) all release-dates and due-dates and (2) the "minimal execution" constraint. Moreover, let us suppose that $\exists t \in [e(R_i^k), s(F_i^l) + minExec_i)$ such that A_i is not executed at time t. Let then t_{min} be the minimal date in $[e(R_i^k), s(F_i^l) + minExec_i)$ such that A_i is not executed at time t_{min}. The minimal execution-time of A_i is $minExec_i$, A_i is executed at $t_{min} - 1$ and A_i is not executed at t_{min} then, there is at least a chunk of A_i which starts at $t_{min} - minExec_i$ and ends at $t_{min} - 1$. Since $t_{min} \leq s(F_i^l) + minExec_i$, there is at least a chunk of A_i which starts before $s(F_i^l) - 1$. This chunk and the forbidden interval F_i^l overlap. Absurd.

- The same deduction rule can be applied in the "reverse direction": If $s(F_i^{l+1}) - minExec_i < s(R_i^k)$ then the required interval R_i^k has to be enlarged to $[s(F_i^{l+1}) - minExec_i, e(R_i^k))$.

4.3 Unary Resource Constraints

The prototype provides a new type of resource: the `IlcPreemptiveUnaryResource`. It represents a resource whose capacity is one and which can be required by interruptible activities. Three constraint-propagation algorithms have been developed to ensure that at any time, at most one interruptible activity is executed on a given resource. The following sections describe these algorithms and provide complexity results. These complexity results are related to one run of the constraint propagation algorithm for a given preemptive resource of capacity one. They are computed in function of n (the number of activities requiring the same resource), of r (the maximal number of required intervals among all activities), of f (the maximal number of forbidden intervals among all activities) and eventually of $\max_i p_i$ (the largest duration amongs durations of activities).

4.3.1 Disjunctive Constraint

Description

An obvious resource constraint propagation algorithm for `IlcPreemptiveResource` has been implemented. It states that, when a time-interval is required by a given activity, this time-interval becomes forbidden for the other activities sharing the same resource. Note that such a constraint is very similar to the classical disjunctive constraint [Erschler 76] [Le Pape 88].

Complexity

The complexity of the algorithm is $O(n * f)$: Indeed, the interval is made forbidden for the n activities requiring the resource and each time, at most f intervals of time have to be checked to make the interdiction. Note that a more careful implementation using a binary tree representation of the intervals for each activity could lead to an overall complexity of $O(n * log(f))$. This complexity is low but results in a poor reduction of the search space (c.f. the experimental study carried out in chapter 7).

Since the success of constraint-based applications often relies on the "pruning power" of constraint propagation algorithms [Baptiste 95b], [Regin 94], we tried to improve the algorithm described in previous section. It seemed to us that an interresting issue was to find an algorithm which could ensure global consistency (i.e. an algorithm which would be able, at each step of the decision making process, to check whether there exists a preemptive schedule meeting the constraints related to a resource). Section 4.3.2 details an algorithm based on Jackson Preemptive Schedule (JPS). This algorithm is however unable to deal with all types of "forbidden intervals", which led us to work on flow-based algorithms detailed in sections 4.3.3 and 4.3.4.

4.3.2 A JPS-Based Resource Constraint

Description

Jackson Preemptive Schedule is the particular schedule associated to the **MWR** (**M**ost **W**ork **R**emaining) priority dispatching rule. Let $A_1, A_2, \cdots, A_n$ be n non-interruptible activities requiring the same unary resource. The following proposition holds:

Proposition 6 *JPS can be built if and only if there is a preemptive schedule meeting the release-dates and the due-dates of all activities.*

Proof: See for instance [Carlier 84].

It seems interesting to try to extend the previous proposition to interruptible activities, subjected to due-dates, release-dates and to "forbidden" and "required" intervals. However, we have been unable to modify (JPS) so that it takes required and forbidden intervals into account. Unformally speaking, this may be explained by the fact that introducing forbidden intervals makes the due-dates criteria unreliable when building the (JPS). Let us however consider the preemptive schedule called **MJPS** (**M**odified **JPS**) built as follow:

1. Schedule all required intervals.
2. Build the preemptive schedule associated to the following dispatching rule: Schedule as soon as possible the unscheduled activity of minimal due-date.

The following propositions hold:

Proposition 7 *If there is a preemptive schedule meeting the release-dates, the due-dates and the "required intervals" constraint then, MJPS can be built.*

Proof: Let Ω be a preemptive schedule meeting the release-dates, the due-dates and the "required intervals" constraint. Let $A_1^{'}, A_2^{'}, \cdots, A_n^{'}$ be n interruptible activities such that:

- $\forall i, p_i^{'} = p_i - \sum_{k=0}^{k=req_k} \left(e(R_i^k) - s(R_i^k)\right)$
- $\forall i, r_i^{'} = r_i$
- $\forall i, d_i^{'} = d_i$
- $\forall i, R_i^{'} = \emptyset$
- $\forall i, F_i^{'} = \emptyset$

Let $A_{n+1}^{'}, A_{n+2}^{'}, \cdots, A_{n+\sum_{i=0}^{n} req_i}^{'}$ be extra interruptible activities corresponding to the required intervals $R_i^k \in R_i$ (for instance, the activity $A_{n+1}^{'}$ corresponds to the required interval R_i^1 and, $r_{n+1}^{'} = s(R_i^1)$, $d_{n+1}^{'} = e(R_i^1)$, $p_{n+1}^{'} = e(R_i^1) - s(R_i^1)$ etc). Let $\Omega^{'}$ be a preemptive schedule of activities $A_1^{'}, A_2^{'}, \cdots, A_{n+\sum_{i=0}^{n} req_i}^{'}$ build according to the following rules:

- If no activity is executed at t on Ω, no activity is executed at t on $\Omega^{'}$;
- If a "required interval" is executed at t on Ω then, the activity corresponding to this interval is executed at t on $\Omega^{'}$;
- If a non-required chunk of A_i is executed on Ω at time t. Then $A_i^{'}$ is executed at t on $\Omega^{'}$.

By construction, $\Omega^{'}$ is a preemptive schedule meeting release-dates and due-dates of the activities $A_1^{'}, A_2^{'}, \cdots, A_{n+\sum_{i=0}^{n} req_i}^{'}$. Then, the JPS related to the same activities can be built (Proposition 6). Let $\Omega^{''}$ be this preemptive schedule. Since $\forall i > n, r_i + p_i = d_i$, the activities corresponding to required intervals are scheduled from their release-date to their due-date. This schedule corresponds exactly to MJPS. Indeed, the required intervals correspond to the fake activities $A_{n+1}^{'}, A_{n+2}^{'}, \cdots$ and the other activities are scheduled according to the MWR rule.

Proposition 8 *If MJPS is admissible and if $\forall i, \forall k, \exists j, \exists l$ such that $F_i^k = R_j^l$, there is a preemptive schedule meeting release-dates, due-dates, required intervals constraints and forbidden intervals constraints.*

Proof: MJPS meets all release-dates, all due-dates, and all required intervals constraints. Moreover, forbidden intervals of a given activity are required intervals of another activity. Consequently, the forbidden intervals constraints are satisfied. MJPS is itself a preemptive schedule meeting release-dates, due-dates, required intervals constraints and forbidden intervals constraints.

As a consequence of the previous propositions, the following result holds:

Proposition 9 *If* $\forall i, \forall k, \exists j, \exists l$ *such that* $F_i^k = R_j^l$, *there is a preemptive schedule meeting release-dates, due-dates, required intervals and forbidden intervals if and only if MJPS is admissible.*

Proof: Obvious according to 7 and 8.

This last result led us to use MJPS in a constraint propagation algorithm. When all forbidden intervals are required intervals (i.e. when all the interdictions are due to requirements), this algorithm enables to achieve global consistency.

Complexity

This constraint propagation algorithm runs in $O(n * r * \log(n * r))$ steps. Indeed, MJPS can be build as follow:

1. Create all activities $A_1^{'}, A_2^{'}, \cdots, A_n^{'}$ and $A_{n+1}^{'}, A_{n+2}^{'}, \cdots, A_{n+\sum_{i=0}^{n} req_i}^{'}$ (cf previous section). This can be done in $O(n + n * r)$ steps.
2. Build the JPS of these activities $A_1^{'}, \cdots, A_{n+\sum_{i=0}^{n} req_i}^{'}$. This can be achieved in $O(n * r * \log(n * r))$ steps (see for instance [Carlier 84]).

However, when the user states interdictions, this algorithm does not ensure that a preemptive schedule meeting all the constraints exists. This led us to consider a new type of resource-constraint algorithm based on a network-flow model.

4.3.3 A Network-Flow Based Resource Constraint

[Regin 94] describes an algorithm to achieve the global consistency of the `all-different` constraint. This constraint of difference is defined on a set of variables and constrains these variables to have pairwise distinct values. [Regin 94] also presents a filtering algorithm for this `all-different` constraint. Based on matching theory, it achieves arc-consistency (A constraint $c(v_1, \cdots, v_n)$ is said to be arc-consistent if and only if for any variable v_i and any value val_i in the domain of v_i, there exist values $val_1, \cdots, val_{i-1}, val_{i+1}, \cdots, val_n$ in the domains of $v_1, \cdots, v_{i-1}, v_{i+1}, \cdots, v_n$, such that $c(val_1, \cdots, val_n)$ holds). [Leconte 95] highlights the similitudes between this constraint of difference and the unary resources constraints and also presents a propagation algorithm for the `all-different` constraint based on the edge-finder techniques.

We present in this section three resource constraint propagation algorithms based on network flow techniques. These techniques are indeed similar to matching techniques and it would be interesting to make a comparison between the algorithms presented in [Regin 94], [Leconte 95] and the following ones.

Description

Let $A_1, \cdots, A_n$ be n interruptible activities requiring the same resource. Let us consider the bipartite graph $g = (X, Y, E)$ such that:

- $X = \{x_1, x_2, \cdots, x_n\}$ and let x_i be the vertex associated to A_i.
- $Y = \{y_1, y_2, \cdots, y_m\}$. Each vertex y_j is associated to a time-interval $[y_j^s, y_j^e)$ such that:
  - $\forall R_i^k, \forall j$ either $R_i^k \cap y_j = \emptyset$ or $R_i^k \cap y_j = y_j$.
  - $\forall F_i^k, \forall j$ either $F_i^k \cap y_j = \emptyset$ or $F_i^k \cap y_j = y_j$.
- If e_{x_i,y_j} denotes the edge from x_i to y_j then,

$$[e_{x_i,y_j} \in E] \Leftrightarrow [\forall k F_i^k \cap y_j = \emptyset]$$

The network-flow graph associated to g is $G = (V', E')$ (see figure 4.1) where,

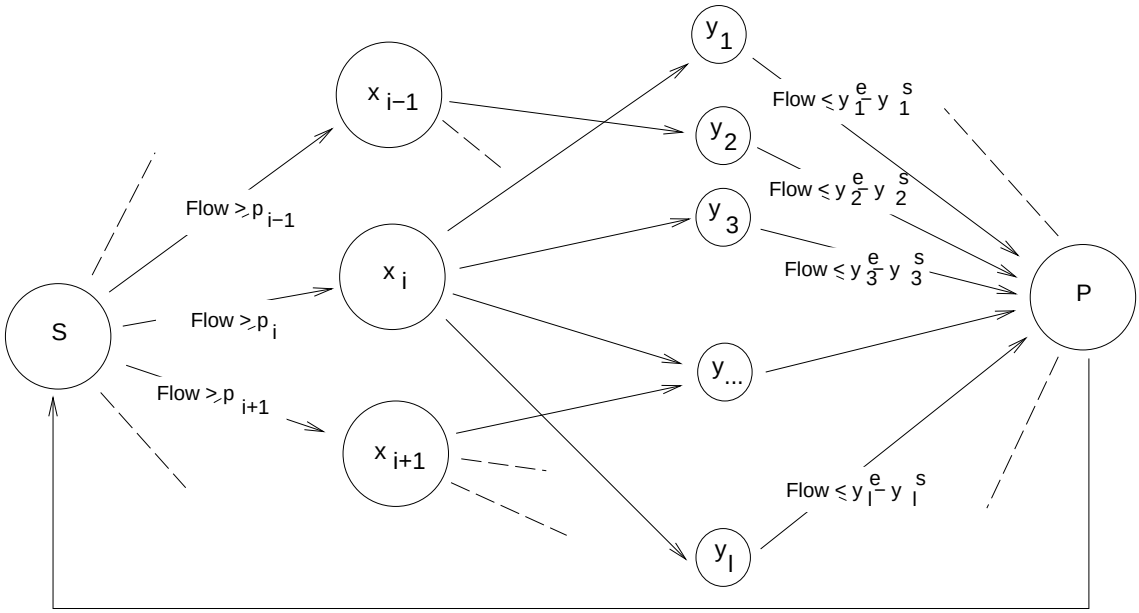

Figure 4.1: Graph G'

- $V' = X \cup Y \cup \{s, p\}$
- $E' = E \cup \{e_{s,x_i} \| i \in \{1, 2, \cdots, n\}\} \cup \{e_{y_j,p} \| j \in \{1, 2, \cdots, m\}\} \cup \{e_{p,s}\}$

Each edge e of G is valuated by two integers e^{min} and e^{max}, respectively the minimal and the maximal value of a compatible flow on e:

- $e^{min}_{p,s} = 0$ and $e^{max}_{p,s} = +\infty$
- $\forall i \in \{1, 2, \cdots, n\}, e^{min}_{s,x_i} = p_i$ and $e^{max}_{s,x_i} = +\infty$
- $\forall j \in \{1, 2, \cdots, m\}, e^{min}_{y_j,p} = 0$ and $e^{max}_{y_j,p} = y^e_j - y^s_j$
- $\forall i \in \{1, 2, \cdots, n\}, \forall j \in \{1, 2, \cdots, m\}$,
  - **either** y_j belongs to a required interval of A_i (i.e. $\exists k$ such that $y_j \subset R^k_i$); then, $e^{min}_{x_i,y_j} = y^e_j - y^s_j$ and $e^{max}_{x_i,y_j} = y^e_j - y^s_j$.
  - **or** y_j is not included in any required interval and neither in any forbidden interval; then $e^{min}_{x_i,y_j} = 0$ and $e^{max}_{x_i,y_j} = y^e_j - y^s_j$.

The resource constraint propagation algorithm is now obvious: It consists in searching for a compatible flow in the graph G'. The proof of this algorithm is completed by the following proposition:

Proposition 10 *There is a preemptive schedule Ω meeting the release-dates, the due-dates, the required and forbidden interval constraints if and only if there is a compatible flow in the valuated graph G'.*

Proof:

- **Necessary Condition:** Let us (1) valuate each edge of G by an integer $f_{x,y}$ and (2) prove that this valuation is a compatible flow.
  1. Definition of f.

$$\begin{cases} \forall x_i \in X, \forall y_j \in Y, f_{x_i,y_j} = \left|\{t \backslash A_i \text{ executed at } t \text{ on } \Omega\} \cap [y^s_j, y^e_j)\right| \\ \forall x_i \in X, f_{s,x_i} = p_i \\ \forall y_j \in Y, f_{y_j,p} = \sum_i f_{x_i,y_j} \\ f_{p,s} = \sum_i p_i \end{cases}$$

  2. The valuation $f_{x,y}$ represents a compatible flow. Indeed, Kirshoff's law holds:
     - For all node x_i, let us compute $\sum_j f_{x_i,y_j}$. According to the definition of f_{x_i,y_j},

$$\sum_j f_{x_i,y_j} = \sum_j \left|\{t \backslash A_i \text{ executed at } t \text{ on } \Omega\} \cap [y^s_j, y^e_j)\right|$$

       Which can also be written

$$\sum_j f_{x_i,y_j} = \left|\{t \backslash A_i \text{ executed at } t \text{ on } \Omega\} \cap \left(\bigcup_j [y^s_j, y^e_j)\right)\right|$$

       which leads to

$$\sum_j f_{x_i,y_j} = |\{t \backslash A_i \text{ executed at } t \text{ on } \Omega\}| = p_i$$

- For all node y_j, it is obvious that, according to the definition of $f_{y_j,p}$, Kirchoff's conservative law holds.
- To check that Kirchoff's law holds at node e, let us compute $\sum_j f_{y_j,p} - f_{p,s}$:

$$\begin{aligned}\sum_j f_{y_j,p} - f_{p,s} &= \sum_j f_{y_j,p} - \sum_i p_i \\ &= \sum_j \sum_i f_{x_i,y_j} - \sum_i p_i \\ &= 0\end{aligned}$$

since $\sum_j f_{x_i,y_j} = p_i$.

- According to the definition of $f_{p,s}$, it is obvious that Kirchoff's law holds at node s.

3. The flow f is compatible:
   - $\forall i, f_{s,x_i} = p_i$ and $e^{min}_{s,x_i} = p_i$ which leads to $e^{min}_{s,x_i} \leq f_{s,x_i}$
   - let us recall the definition of f_{x_i,y_j}:

$$f_{x_i,y_j} = \left|\{t \backslash A_i \; executed \; at \; t \; on \; \Omega\} \cap [y^s_j, y^e_j)\right|$$

   Let us distinguish again three cases:
   - **either** y_j belongs to a required interval of A_i; then, $e^{min}_{x_i,y_j} = y^e_j - y^s_j$ and $e^{max}_{x_i,y_j} = y^e_j - y^s_j$. Since Ω is an acceptable schedule, A_i is executed in the time interval y_j and thus, according to the definition of f_{x_i,y_j}, the value of the flow in the edge e_{x_i,y_j} is $y^e_j - y^s_j$; which is an acceptable value of the flow.
   - **or** y_j belongs to a forbidden interval of A_i; then, $e^{min}_{x_i,y_j} = 0$ and $e^{max}_{x_i,y_j} = 0$. Since Ω is an acceptable schedule, A_i is not executed in y_j; which proves that the flow is acceptable.
   - **or** y_j is not included in any required or forbidden interval; then $e^{min}_{x_i,y_j} = 0$ and $e^{max}_{x_i,y_j} = y^e_j - y^s_j$; which of course makes the flow f_{x_i,y_j} acceptable.

- **Sufficient Condition:** Let f be a compatible flow and let us build an acceptable schedule Ω meeting the release-dates, the due-dates, the required and/or forbidden constraints. Ω is build as follow: For each vertex y_j, we have to schedule in $[y^s_j, y^e_j)$ the activities A_i such that $f_i > 0$. Let us suppose that Ω has been built until time y^s_j. Let t be the current date and let us then iterate on A_i as follow:
  - schedule A_i in the time-interval $[t, t + f_{x_i,y_j})$;
  - increase t of f_{x_i,y_j};

  Since f is a compatible flow and according to the construction of G, Ω is obviously acceptable.

An example

Let us consider the following problem: Four interruptible activities A, B, C, D have to be scheduled on the same unary resource. Note that activities B, C, D cannot execute in some given time-intervals.

A ONE-MACHINE PROBLEM				
Activity	duration	release-date	due-date	forbidden interval
A	0	5	10	
B	0	4	12	[3 6)
C	5	3	15	[6 12)
D	0	2	15	[3 6)

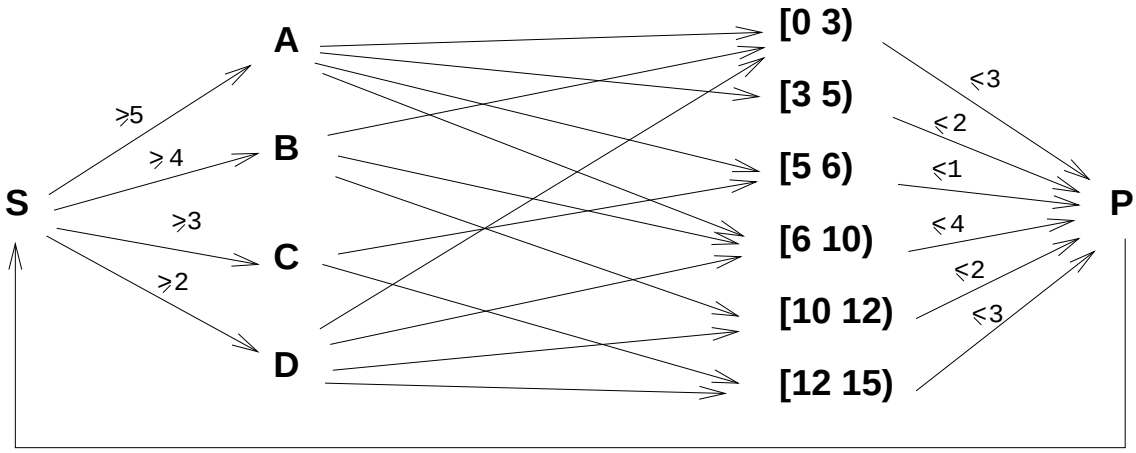

Figure 4.2: The network-graph associated to A, B, C, D

According to the model described in the previous section, let us build the network-graph associated to the four activities under consideration. Note that on figure 4.2, only some of the edges' valuations have been represented. For instance, the symbol ≥ 5 on edge $e_{s,A}$ stands for $e^{min}_{s,A} = 5$. Figure 4.3 presents a compatible flow.
This compatible flow corresponds, for instance, to the following schedule:

```
B A A A A A B B B D D   C C C
```

Reduction of G

We have seen that, as far as interruptible activities are concerned, a straight way of making decisions consists of stating that an activity is required (or "forbidden") to execute in a given time-interval. When considering the graph G, it seems that stating these requirements and interdictions may increase the size of G. This can be avoided by reducing G.

Consider a required interval $[a, b)$ of an activity A_i. G is equivalent to a graph G' build as follow: (1) all vertexes corresponding to the time-interval included in $[a, b)$ are removed of G, (2) the minimal acceptable flow through e_{s,x_i} can be decreased of $b - a$. If the new value of this minimal acceptable flow through e_{s,x_i} becomes null, the vertex x_i can also be removed of G.

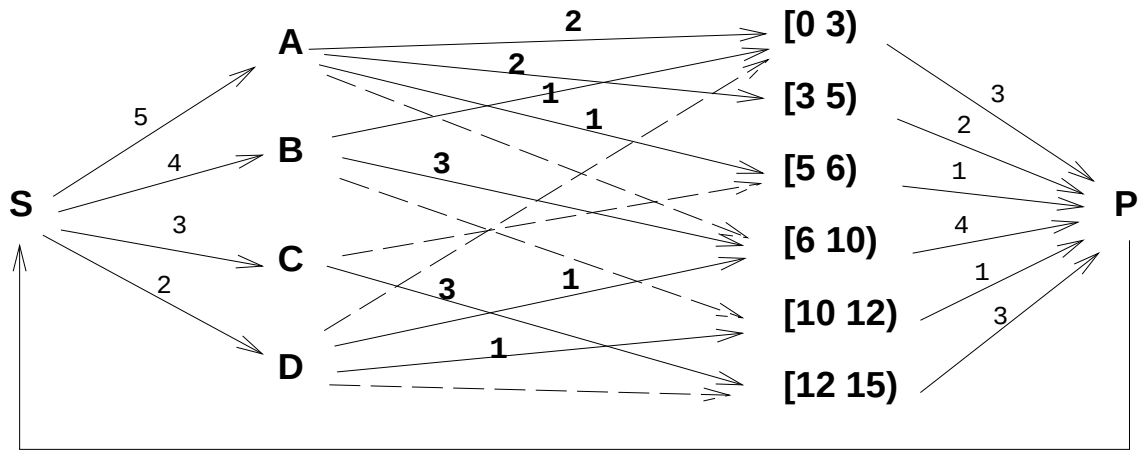

Figure 4.3: A compatible Flow

Proposition 11 *There is a preemptive schedule Ω meeting the release-dates, the due-dates, the required intervals and the forbidden intervals if and only if there is a compatible flow in the valuated graph $G^{'}$.*

Proof: (sketch)

- **Necessary Condition:** Since 10 holds, there is a compatible flow in the valuated graph G. Let us then modify G step by step:
  - For each required interval y_j of G, there is one and only one edge e_{x_i,y_j} whose flow is non null. Let us then consider the cycle $\sigma = (s, x_i, y_j, p, s)$. and let us reduce the values of the flows through each edge of the cycle σ of $y_j^e - y_j^s$. This new flow still respects Kirshoff's law but, the edge e_{s,x_i} is violated. Let us then decrease e_{s,x_i}^{min} of $y_j^e - y_j^s$; which makes the flow acceptable.
  - The vertex y_j can be obviously removed.
  - iterate.

  The modified graph is exactly $G^{'}$.
- **Sufficient Condition:** if there is a compatible flow in the valuated graph $G^{'}$, we can then (1) compute the corresponding flow in G and (2) make it compatible by increasing the flow passing through the cycles (s, x_i, y_j, p, s) (for each y_j belonging to a required interval).

Complexity

We propose in this section three evaluations of the complexity of our algorithm. The first (1) one is a basic evaluation, the second (2) one relies on the reduced graph $G^{'}$ and the third one (3) shows that by a slight modification of the algorithm, its theoretical complexity can be improved.

Basic Evaluation of the Complexity

Let us compute the theoretical complexity of this network-flow based resource constraint. The two main steps of this algorithm are:

1. **The building of the graph** G'. This can be achieved in $O(n * r + n^2 * f)$:
   - The required intervals have to be "removed"; which can be done in $O(n * r)$ steps.
   - The "left" vertexes corresponding to the n activities can be built in linear time.
   - There are potentially $O(n * f)$ right vertexes (at most $f + 1$ vertexes per activity).
   - The graph is bipartite and thus, there are at most $O(n^2 * f)$ edges.
2. **The Search for a compatible flow**. Many algorithms have been designed to achieve this step of the algorithm. [Gondran 84] presents Herz's algorithm which **either** computes a compatible flow **or** proves that no compatible flow exists. Given a graph ?, this algorithm runs in $O(M^2 * C)$, where M denotes the number of edges in ? and C denotes the maximal absolute value of the minimal (and maximal) acceptable flow of one edge over all edges of ?. Applying this algorithm to the graph G leads to a theoretical complexity of $O(n^4 * f^2 * \max_i p_i)$.

Using G' to Bring Complexity Down

Let us recall **Herz's algorithm**: Let $? = (\xi, U)$ a connected graph. To each arc, $\upsilon \in U$ let b_υ and c_υ $(b_\upsilon \leq c_\upsilon)$ be two integers attached to υ. The goal of the algorithm is to find a flow φ in ? compatible with the constraints:

$$\forall \upsilon \in U, b_\upsilon \leq \varphi_\upsilon \leq c_\upsilon$$

The basic idea of the algorithm is to introduce the function d which represents for each flow φ the sum of the distances from each interval $[b_\upsilon, c_\upsilon]$ to φ_υ.

$$d(\varphi) = \sum_{\upsilon \in U} \max(0, b_\upsilon - \varphi_\upsilon, \varphi_\upsilon - c_\upsilon)$$

It is then obvious that φ is compatible if and only if $d(\varphi) = 0$. At each step of the algorithm a labeling method on ? is used to find a cycle in ? which allows to modify the current flow φ so that d_φ decreases (for more details, see [Gondran 84]). This labeling method runs in $O(M)$. Since b_υ and c_υ have integer values, $d(\varphi)$ decreases **at least of one unit at each step**. This leads to theoretical complexity bounded by $O(M * d(\varphi^0))$ where φ^0 is the initial flow.

Let us now consider G' and let us compute the cost of searching for a compatible flow f in G'. We can chose the null flow as an initial flow $(\forall i, \forall j, f_{x_i, y_j} = 0)$. Let us find out which edges are violated:

- Since $e^{min}_{s,x_i} = d_i > 0$, the "left" edges e_{s,x_i} are violated.
- The "right" edges are not violated since $e^{min}_{y_j,p} = 0$.
- the "middle" edges are not violated since (1) there are no more edge (cf, building of G') corresponding to a required interval and (2) for the remaining edges, $e^{min}_{x_i,y_j} = 0$.

We can then compute an upper bound of $d(\varphi^0)$:

$$\begin{aligned} d(\varphi^0) &\leq \sum_{i=1}^{n} e^{min}_{s,x_i} \\ &\leq \sum_{i=1}^{n} p_i \\ &\leq n * \max_i p_i \end{aligned}$$

This leads in turn to an overall complexity of $O(M * n * \max_i p_i)$. Since $M \leq n^2 * f$, the complexity can also be written $O(n^3 * f * \max_i p_i)$.

Using JPS to Compute an Initial Flow

Since the initial value $d(\varphi^0)$ is crucial, let us slightly modify the algorithm to improve the initial flow. The first step of the algorithm now consists in building MJPS (4.3.2). This can be done in $O(n.r.\log(n.r))$. We also have to build an initial φ'^0 flow in G' thanks to this schedule (same type of flow-building than in proposition 10). Either (MJPS) is fully acceptable and the algorithm stops or, (MJPS) is not acceptable, which means that there are activities which are executed during forbidden intervals. We then have to search in G' for a compatible flow. Since MJPS has been used to build φ'^0, we can upper-bound the value of $d(\varphi'^0)$:

$$d(\varphi'^0) = \sum_{i=1}^{n} \sum_{k=1}^{forb_i} \left(e(F_i^k) - s(F_i^{k+1}) \right)$$

Let us note $\mathcal{F} = d(\varphi'^0)$, the sum of the sizes of forbidden intervals. The theoretical complexity of the overall algorithm is then $O(n*r*\log(n*r) + M*\mathcal{F}) = O(n*r*\log(n*r) + n^2*f*\mathcal{F})$.

4.3.4 A Flow-Based Technique for Adjustment of Time-Bounds

The resource constraint algorithm described in the previous sections allows to trigger a failure immediately but does not perform any updating of start-times, due-dates or of sets of required/forbidden intervals is performed. In this section, we propose some algorithms deducing new time bounds, new requirements (new required intervals) and new interdictions. We will see that the the price to pay for efficient deductions is often a high theoretical complexity of the algorithm. Hopefully, this is often balanced by a drastic reduction of the search space.

Adjustments of the Edges' Capacities

A simple algorithm, which allows to update the minimal and maximal possible values of the flow passing through a given edge e_{x_i,y_j}, consists in looking for the minimal (maximal) value

of the flow passing through e_{x_i,y_j} such that there still exists a compatible flow. The following algorithm achieves the updating of the minimal value of the flow passing through e_{x_i,y_j}:

1. Save the value of $e^{max}_{x_i,y_j}$.
2. While there exists a compatible flow, decrease $e^{max}_{x_i,y_j}$ of one unit.
3. When exiting step (2), we know that the current value of $e^{max}_{x_i,y_j}$ is a lower bound of any compatible flow passing through e_{x_i,y_j}. Let us then set the value of $e^{min}_{x_i,y_j}$ to $e^{max}_{x_i,y_j} + 1$ and restore the value of $e^{max}_{x_i,y_j}$ saved at step (1).

Symetrically, we could write an algorithm which updates the maximal value of the flow passing through e_{x_i,y_j}. This updating, for each edge e_{x_i,y_j}, of the minimal and maximal value of any compatible flow leads, in turn, to the modification of time-bounds of activities and eventually to the modification of the sets of required (and/or) forbidden intervals:

- **Proposition 12** *If $e^{min}_{x_i,y_j} > 0$ then, at least $e^{min}_{x_i,y_j}$ units of activity A_i are executed in $[y^s_j, y^e_j)$.*

  Proof: If there exists a preemptive schedule meeting all the requirements (release-dates, due-dates, ...) such that less than $e^{min}_{x_i,y_j}$ units of activity A_i are executed in $[y^s_j, y^e_j)$ then, according to 10, there is a compatible flow such that the flow passing through e_{x_i,y_j} is strictly lower than the value computed bellow. Absurd.

  Proposition 12 allows us to:

  - update the latest start-time of A_i:

    $$startMax(A_i) = \min(startMax(A_i), y^e_j - e^{min}_{x_i,y_j}) \tag{4.1}$$

  - update the earliest end-time of A_i:

    $$endMin(A_i) = \max(endMin(A_i), y^s_j + e^{min}_{x_i,y_j}) \tag{4.2}$$

  - test whether the interval $[y^s_j, y^e_j)$ is fully required by A_i or not:

    $$e^{min}_{x_i,y_j} = y^e_j - y^s_j$$

- **Proposition 13** *If $e^{max}_{x_i,y_j} = 0$ then $[y^s_j, y^e_j)$ must be added to the set of forbidden intervals of A_i.*

  Proof: Obvious

Note that, for the given edge e_{x_i,y_j}, at most $O(e^{max}_{x_i,y_j} - e^{min}_{x_i,y_j})$ calls to a procedure searching for a compatible flow are required to update the minimal (maximal) value of a compatible flow. This algorithm can be improved: instead of decreasing $e^{max}_{x_i,y_j}$ step by step, the minimal value of the flow passing through e_{x_i,y_j} can be reached by applying a dichotomizing procedure on the value of $e^{max}_{x_i,y_j}$.

Let u_s be the number of elementary steps for updating the minimal (maximal) value of a compatible flow on an edge e_{x_i,y_j} ($s = y_j^e - y_j^s$).

Proposition 14 *The following equation holds:*

$$u_{2s} = M * s + u_s \tag{4.3}$$

Proof: Let I be an interval of size $2s$. Let us recall that the dichotomizing algorithm (1) splits the interval I in two intervals I_1 and I_2 of size s. The algorithm then focus on one of the interval, say I_1, and (2) looks for a compatible flow such that the flow passing through e_{x_i,y_j} is lower than a value corresponding to the middle of I and (3) iterate the process on I_1 or I_2 (depending on the result of the search).

(1): This can be done in a constant number of steps.

(2): As we have said, the search for a compatible flow can start with any initial flow. Here, it is of course interesting to start the search with the previous compatible flow φ^0 (i.e., the flow which was compatible before splitting the interval I). We can then bound the value of $d(\varphi^0)$ by the half size of I; which leads to a number of steps proportional to $M * s$.

(3): The algorithm is iterated on I_1 or I_2. Since their sizes are equal to s, the remaining number of steps to execute before the end of the algorithm is u_s.

Proposition 15 $u_s^0 = M * s$ *is a solution of 4.3.*

Proof: Obvious ($M * 2 * s = M * s + M * s$).

Proposition 16 $\forall u_s$, *solution of 4.3.*

$$\exists \pi \setminus u_s = \pi + M * s$$

Proof: Let u_s be any solution of 4.3 and $v_s = u_s - u_s^0$. The following equation holds:

$$\begin{aligned} & v_{2s} + M * 2 * s = M * s + v_s + M * s \\ \Leftrightarrow & \\ & v_{2s} = v_s \end{aligned}$$

This leads in turn to: $\exists \pi \setminus v_s = \pi$ and, since $v_s = u_s - M * s$,

$$\exists \pi \setminus u_s = \pi + M * s$$

The number of steps of this algorithm (for updating the minimal (maximal) value of a compatible flow on the edge e_{x_i,y_j}) is then $O(M*(y_j^e - y_j^s))$. The overall complexity of the dichotomizing algorithm is then:

$$O\left(\sum_i \sum_j M * (y_j^e - y_j^s)\right)$$

Let us find an upper-bound of the previous expression:

$$\begin{aligned}\sum_i \sum_j (\pi + M * (y_j^e - y_j^s)) &\leq \sum_i \sum_j M * (y_j^e - y_j^s) \\ &\leq M * \sum_i (\max_i d_i - \min_i r_i) \\ &\leq M * n * (\max_i d_i - \min_i r_i) \\ &\leq n^3 * f * (\max_i d_i - \min_i r_i)\end{aligned}$$

The theoretical complexity of the dichotomizing algorithm is then

$$O\left(n^3 * f * (max_i d_i - min_i r_i)\right)$$

This complexity is high but, the updating of time bounds usually results in a better branching-strategy and in an interesting reduction of the search space.

Global Updating of Time-Bounds

The algorithm 4.3.4 provides interesting time-bounds but, the deductions are made edge by edge. This algorithm doesn't "look forward" the current edge on which it is working. We present in this section an algorithm deducing time-bounds of interruptible activities in a more "global fashion".

Let us focus on a given activity A_i. Let $y_{k_1}, y_{k_2}, \cdots, y_{k_t}, \cdots, y_{k_l}$ be the l time-intervals in which A_i can be executed ($y_{k_1} \prec y_{k_2} \prec \cdots \prec y_{k_l}$). Figure 4.4 illustrates these notations. Let

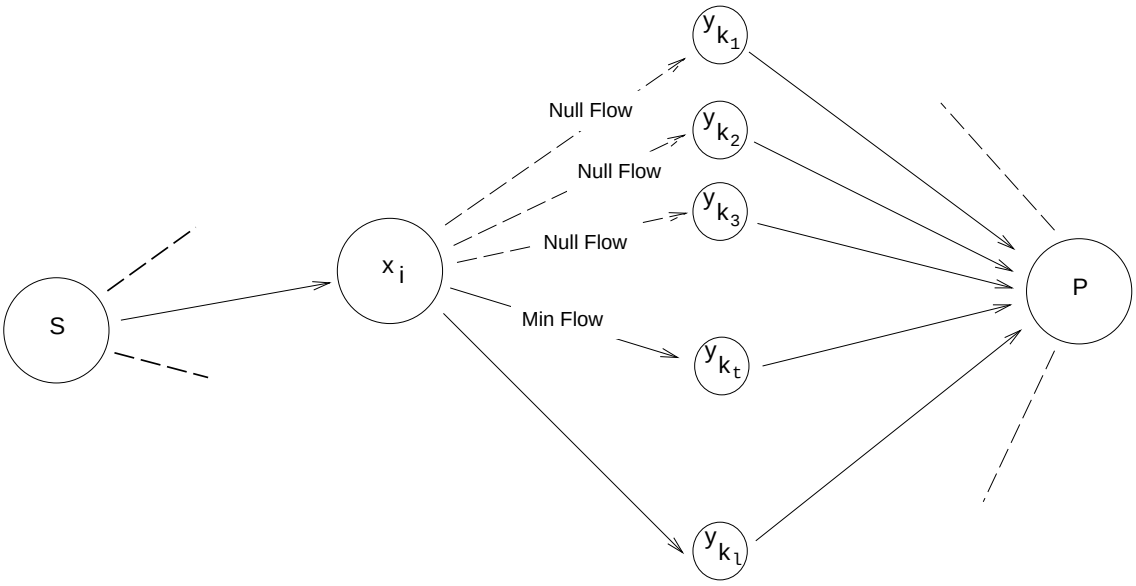

Figure 4.4: Updating of A_i's latest start-time

us now describe this algorithm:

1. Compute t, the maximal integer such that
   - there is a compatible flow in G';

- the flow passing through $y_{k_1}, y_{k_2}, \cdots, y_{k_t}$ is null;

2. Compute the minimal possible f_{min} value of the flow through $e_{x_i, y_{k_{t+1}}}$ such that

   - there is still a compatible flow in $G^{'}$;
   - the flow passing through $y_{k_1}, y_{k_2}, \cdots, y_{k_t}$ is still null;

3. The latest start-time of A_i can then be updated according to the following proposition.

Proposition 17

$$startMax(A_i) = \min\left(startMax(A_i), y^e_{k_{t+1}} - f_{min}\right)$$

Proof: Let us suppose that there is a preemptive schedule meeting release-dates, due-dates, required intervals and forbidden intervals and that the start of A_i is scheduled after $y^e_{k_{t+1}} - f_{min}$. Since 10 holds, there is a compatible flow such that (1) the flow passing through $y_{k_1}, y_{k_2}, \cdots, y_{k_t}$ is null and (2) such that the flow passing through $y_{k_{t+1}}$ is strictly smaller than f_{min}. Absurd.

Let us compute the theoretical **complexity** of this algorithm. For each activity A_i, the procedure searching for a compatible flow is called until the vertex y_{k_t} is found. Since we can suppose that the initial flow φ^0 is compatible with $G^{'}$, less than

$$M * \sum_{m=1}^{m=t+1} \left(y^e_{k_m} - y^s_{k_m}\right)$$

steps are necessary to compute f_{min}. We can upper-bound this value by $M * (d_i - r_i)$; which leads to an overall complexity of:

$$O(n^2 * f * n * \max_i(d_i - r_i)) = O(n^3 * f * \max_i(d_i - r_i))$$

4.4 Discrete Resource Constraints

The algorithms dedicated to unary-resource constraint described in section 4.3.3 rely on network-flow models. Amongst the advantages of this model, note that it can be easyly extended to discrete resource constraint, by modifying the valuations of the graph g.

Let us introduce the following notations. Let C be the capacity (i.e. the number of activities that can be executed on the same resource at the same time) of the discrete resource under consideration. Let c_i be the amount of resource used by activity A_i when being executed. Let us consider the graph g, valuated as follow:

- $e^{min}_{p,s} = 0$ and $e^{max}_{p,s} = +\infty$
- $\forall i \in \{1, 2, \cdots, n\}, e^{min}_{s,x_i} = p_i * c_i$ and $e^{max}_{s,x_i} = +\infty$
- $\forall j \in \{1, 2, \cdots, m\}, e^{min}_{y_j,p} = 0$ and $e^{max}_{y_j,p} = C * (y^e_j - y^s_j)$
- $\forall i \in \{1, 2, \cdots, n\}, \forall j \in \{1, 2, \cdots, m\}$, two cases can be distinguished to evaluate e_{x_i, y_j}.

- **either** y_j belongs to a required interval of A_i (i.e. $\exists k$ such that $y_j \subset R_i^k$); then, $e^{min}_{x_i,y_j} = c_i * (y_j^e - y_j^s)$ and $e^{max}_{x_i,y_j} = c_i * (y_j^e - y_j^s)$.
- **or** y_j is not included in any required interval and neither in any forbidden interval; then $e^{min}_{x_i,y_j} = 0$ and $e^{max}_{x_i,y_j} = c_i * (y_j^e - y_j^s)$.

Proposition 18 *There is a preemptive schedule meeting all the constraints if and only if there is a compatible flow in g.*

Proof: Same type of proof than the one described in section 4.3.3.

Chapter 5

A Mixed Resource Constraint

The resource constraints presented in the previous chapters are dedicated either to non-preemptive scheduling (chapter 3) or to preemptive scheduling (chapter 4). However, the same resource may be required by interruptible and non-interruptible activities. Currently, two models can represent such a "mixed" resource constraints:

- If on the same resource there is a large number of interruptible activities and few non-interruptible activities, a fully interruptible resource constraint (cf., chapter 4) can be used. It is sufficient to state for each non-interruptible activity that, the minimal duration of execution intervals ($minExec_i$) is equal to the duration of the activity.
- On the contrary, if the activities requiring the resource are mainly non-interruptible activities, it may be interesting to use a non-interruptible resource constraint and to model interruptible activities as sets of non-interruptible sub-activities.

In this chapter, we discuss the feasibility of a direct representation of this mixed resource constraint. Note that we only consider the case of a unary resource constraint. In the first part of this chapter (5.1) we give a characterization of the edge-finder which enable us to extend it. We then introduce (5.2) some deduction rules for interruptible activities which allow us to build a "mixed" edge-finder. To conclude, we show that the edge-finder characterization provided in section (5.1) still holds for the mixed edge-finder (5.3).

5.1 A Characterization of the Edge-Finder

Let us recall the characterization of the edge-finder algorithm provided in [Carlier 90]. Let $A_1, \cdots, A_n$ be n activities requiring the same unary resource and let r_i, d_i, and p_i respectively denote the release-date, the due-date, and the processing time (duration) of activity A_i.

Proposition 19 *The edge-finder algorithm (primal version as described in section 3.1.3) achieves the same time-bounds adjustment than the following deduction rules:*

$$\forall \mathcal{K} \subset \{A_1, \cdots, A_n\}, \forall A \notin \mathcal{K},$$

$$\min_{k \in \mathcal{K} \cup \{A\}} r_k + \sum_{k \in \mathcal{K} \cup \{A\}} p_k > \max_{k \in \mathcal{K}} d_k \quad \Rightarrow \mathcal{K} \prec A$$

$$\Rightarrow r_A \geq \min_{k \in \mathcal{K}} r_k + \sum_{k \in \mathcal{K}} p_k$$

Proof: see [Carlier 90].

Let us introduce some **definitions**:

- A preemptive schedule $\mathcal{S}$ of activities $A_1, \cdots, A_n$ is said to be $\overline{\mathbf{A_i}}$**-preemptive** if and only if A_i is not interrupted on this schedule, while the other activities may be interrupted.
- A schedule is said to be acceptable if and only if it meets the release-dates and the due-dates of all activities.
- Given an activity A and a schedule $\mathcal{S}$, let $A_{\mathcal{S}}$ be the set of time point at which A is executed on $\mathcal{S}$.

  $$A(\mathcal{S}) = \{t \backslash\ A \textit{ executed at } t \textit{ on } \mathcal{S}\}$$

  Let $s_{\mathcal{S}}(A)$ and $e_{\mathcal{S}}(A)$ be respectively the start time and end time of A on $\mathcal{S}$.

  $$\begin{cases} s_{\mathcal{S}}(A) = \min(A_{\mathcal{S}}) \\ e_{\mathcal{S}}(A) = \max(A_{\mathcal{S}}) + 1 \end{cases}$$

- Given a set of activities $\mathcal{K}$ let the release-date of $\mathcal{K}$ be the earliest release-date amongst all release-dates of activities in $\mathcal{K}$.
- Let $r_1^{'}, \cdots, r_n^{'}$ be the release-dates deduced by the primal version of the edge-finder $(\forall i, r_i \leq r_i^{'})$.

Proposition 20 $r_i^{'}$ *is the earliest date such that* $\exists \mathcal{S}$*, an acceptable* $\overline{A}_i$*-preemptive schedule on which* $s_{\mathcal{S}}(A_i) = r_i^{'}$.

Let us prove that, $\forall \Omega$ an acceptable $\overline{A}_i$-preemptive schedule, $s_\Omega(A_i) \geq r_i^{'}$ (proposition 21) and that $\exists \mathcal{S}$, an acceptable $\overline{A}_i$-preemptive schedule such that $s_{\mathcal{S}}(A_i) = r_i^{'}$ (proposition 22).

Proposition 21 $\forall \Omega$ *an acceptable* $\overline{A}_i$*-preemptive schedule,* $s_\Omega(A_i) \geq r_i^{'}$.

Proof: Two cases are possible: **either** $r_i^{'} = r_i$ and then 21 obviously holds **or** $r_i^{'} > r_i$. Let us focus on the second case. Since 19 holds, a subset $\mathcal{K}$ of activities has triggered the adjustment achieved by the edge-finder:

$$\exists \mathcal{K} \subset \{A_1, \cdots, A_{i-1}, A_{i+1}, \cdots, A_n\} \,/\, \min_{k \in \mathcal{K} \cup \{A_i\}} r_k + \sum_{k \in \mathcal{K} \cup \{A_i\}} p_k > \max_{k \in \mathcal{K}} d_k$$

Let us suppose that $\exists\Omega$ an acceptable $\overline{A_i}$-preemptive schedule such that, A_i is not scheduled after all activities in $\mathcal{K}$. Since Ω is acceptable, the release-dates and due-dates are met. Consequently, between the release-date of $\mathcal{K} \cup \{A_i\}$ and the due-date of $\mathcal{K}$, all activities in $\mathcal{K}$ and A_i are scheduled; which leads to:

$$\min_{k \in \mathcal{K} \cup \{A_i\}} r_k + \sum_{k \in \mathcal{K} \cup \{A_i\}} p_k \leq \max_{k \in \mathcal{K}} d_k$$

Absurd since 19 holds.

Proposition 22 *$\exists\mathcal{S}$, an acceptable $\overline{A_i}$-preemptive schedule such that $s_{\mathcal{S}}(A_i) = r_i^{'}$.*

Proof: Let us build the schedule $\mathcal{S}$ as follow:

1. Consider the new release date $r_i^{'}$ of A_i and build JPSF (where F stands for "Forward") the primal version of Jackson Preemptive Schedule of $A_1, \cdots, A_n$. JPSF exits otherwise a failure is triggered by the edge-finder.

2. Copy JPSF in $\mathcal{S}$ from time 0 to time $r_i^{'}$.

3. Let $\alpha_1, \cdots, \alpha_n$ be n activities such that.

$$\begin{cases} \forall j, r_{\alpha_j} = r_i^{'} \\ \forall j, p_{\alpha_j} = p_j - \left| A_j(\mathcal{S}) \cap [0\ r_i^{'}) \right| \\ \forall j\ such\ that\ p_{\alpha_j} > 0, d_{\alpha_j} = d_j \\ \forall j\ such\ that\ p_{\alpha_j} = 0, d_{\alpha_j} = +\infty\ \ (fake\ activities) \end{cases}$$

   There exists an acceptable preemptive schedule of activities $\alpha_1, \cdots, \alpha_{i-1}, \alpha_{i+1}, \cdots, \alpha_n$ (since the "right" part of JPSF is an acceptable schedule). We can build JPSFB (the dual version of JPS) of $\alpha_1, \cdots, \alpha_{i-1}, \alpha_{i+1}, cdots, \alpha_n$. Let us copy (JPSFB) on $\mathcal{S}$. $\mathcal{S}$ is then an acceptable preemptive schedule for activities $A_1, \cdots, A_{i-1}, A_{i+1}, \cdots, A_n$.

4. Schedule activity A_i on $\mathcal{S}$ between $r_i^{'}$ and $r_i^{'} + p_i$.

Let us prove that $\mathcal{S}$ is an acceptable $\overline{A_i}$-preemptive schedule. Since $\mathcal{S}$ is an acceptable preemptive schedule for activities $A_1, \cdots, A_{i-1}, A_{i+1}, \cdots, A_n$, we just have to verify that A_i does not overlap with any other activities on $\mathcal{S}$. If A_i overlaps with at least an other activity, let then t_1 be the earliest date at which an overlap occurs. Let then t_2 be the latest date such that $\mathcal{S}$ is full in $[t_1\ t_2)$. Since (1) at time t_2, $\mathcal{S}$ is free, and (2) since (JPSFB) is build such that activities are scheduled as late as possible, t_2 corresponds to a due-date; say d_k. Let us recall our deductions:

1. A_i executes between $r_i^{'}$ and $r_i^{'} + p_i$.

2. The resource is full between t_1 and d_k.

3. A_i and an other activity overlap at t_1.

Unformally, this means that the amount of work on $\mathcal{S}$ over $[r_i^{'}, d_k)$ exceeds the available capacity of the resource. More formally, this can be written:

$$p_i + \sum_{j \neq i} |A_j(\mathcal{S}) \cap [t_1, d_k)| > d_k - r_i^{'}$$

Since there is no overlap between $r_i^{'}$ and t_1, this is equivalent to:

$$r_i^{'} + p_i + \sum_{j \neq i} \left| A_j(\mathcal{S}) \cap [r_i^{'}, d_k) \right| > d_k$$

- Let us consider an activity A_j **such that** $d_j \leq d_k$. We claim that:

$$p_j^* = \left| A_j(\mathcal{S}) \cap [r_i^{'}, d_k) \right|$$

  Indeed, **if** $p_j^* = 0$ then A_j is finished at time $r_i^{'}$ on JPSF. Since JPSF and $\mathcal{S}$ are identical until time $r_i^{'}$, $\left| A_j(\mathcal{S}) \cap [r_i^{'}, d_k) \right| = 0$. **If** $p_j^* > 0$ then, there are p_j^* units of A_j to execute after $r_i^{'}$ on $\mathcal{S}$. Since $d_j \leq d_k$, and since $\mathcal{S}$ is acceptable for activities $A_1, \cdots, A_{i-1}, A_{i+1}, \cdots, A_n$, these p_j^* units are the ones executed on $\mathcal{S}$ before d_k; which leads to:

$$\forall j,\ such\ that\ d_j > d_k,\ \left| A_j(\mathcal{S}) \cap [r_i^{'}, d_k) \right| = p_j^*$$

- Let us consider an activity A_j **such that** $d_j > d_k$. We claim that A_j is not executed in $[r_i^{'}, d_k)$ otherwise, it would mean that **either** $\mathcal{S}$ is full in the interval $[r_i^{'}, d_j)$; which is absurd according to the choice of t_2 **or** that JPSB has scheduled A_j at a time-point $t < t_2$ instead of scheduling it at t_2 where the schedule is free. Absurd. This can be written as follow:

$$\forall j,\ such\ that\ d_j > d_k,\ \left| A_j(\mathcal{S}) \cap [r_i^{'}, d_k) \right| = 0$$

We have proven that

$$\begin{cases} r_i^{'} + p_i + \sum_{j \neq i} \left| A_j(\mathcal{S}) \cap [r_i^{'}, d_k) \right| > d_k \\ \forall j,\ such\ that\ d_j \leq d_k,\ p_j^* = \left| A_j(\mathcal{S}) \cap [r_i^{'}, d_k) \right| \\ \forall j,\ such\ that\ d_j > d_k,\ \left| A_j(\mathcal{S}) \cap [r_i^{'}, d_k) \right| = 0 \end{cases} \tag{5.1}$$

Which leads to:

$$r_i^{'} + p_i + \sum_{j \neq i\ /\ d_j \leq d_k} p_j^* > d_k$$

This test corresponds exactly to the edge-finder test 3.2. It triggers an adjustment of the release-date of A_i to a value strictly greater than $r_i^{'}$; which is absurd since $r_i^{'}$ is already the value of the release-date deduced by the edge-finder.

5.2 A Mixed Edge-Finder

Let $A_1, \cdots, A_n$ be n activities requiring the same unary resource and let us suppose that some of these activities are interruptible. The aim of this section is (1) to describe a set of deduction

rules to update time-bounds of $A_1, \cdots, A_n$, (2) to rewrite these deduction rules thanks to (JPS), and (3) to provide an algorithm based on (JPS) running in $O(n^2)$.

Note that in this section, we do not use the modelization described in section 4.2. We focus on the time bounds of activities (release-date, due-date) and we do not consider the "required" or "forbidden" intervals as described in 4.2. However, since for interruptible activities, the equation $start + duration = end$ does not hold, we introduce some more notations: Let $emin_i$ and $smax_i$ denote respectively the minimal end-time and the maximal start-time for each activity A_i. The following equations hold:

$$\begin{cases} r_i + p_i \leq emin_i \\ \\ smax_i + p_i \leq d_i \end{cases}$$

5.2.1 Deduction Rules

We propose two sets of deduction rules. The first one corresponds to proposition 19 and holds for a non-interruptible activity A. The second one (24) holds for any interruptible activity.

Proposition 23

$$\forall \mathcal{K} \subset \{A_1, \cdots, A_n\}, \forall A \notin \mathcal{K}\ A\ non-interruptible,$$

$$\min_{k \in \mathcal{K} \cup \{A\}} r_k + \sum_{k \in \mathcal{K} \cup \{A\}} p_k > \max_{k \in \mathcal{K}} d_k \quad \Rightarrow \mathcal{K} \prec A$$

$$\Rightarrow r_A \geq \min_{k \in \mathcal{K}} r_k + \sum_{k \in \mathcal{K}} p_k$$

Proof: since the proof described in [Carlier 90] does not rely on the fact that the other activities (except A) are non-interruptible activities, this proposition holds.

Proposition 24

$$\forall \mathcal{K} \subset \{A_1, \cdots, A_n\}, \forall A \notin \mathcal{K}\ A\ \textbf{interruptible},$$

$$\min_{k \in \mathcal{K} \cup \{A\}} r_k + \sum_{k \in \mathcal{K} \cup \{A\}} p_k > \max_{k \in \mathcal{K}} d_k \quad \Rightarrow emin_A \geq \min_{k \in \mathcal{K} \cup \{A\}} r_k + \sum_{k \in \mathcal{K} \cup \{A\}} p_k$$

Proof: This proposition states that A cannot be fully scheduled before any activity of $\mathcal{K}$. Over the interval $\left[\min_{k \in \mathcal{K} \cup \{A\}} r_k \ \max_{k \in \mathcal{K}} d_k\right)$, the maximal number of units of A that may be executed is:

$$\Delta = \max_{k \in \mathcal{K}} d_k - \min_{k \in \mathcal{K} \cup \{A\}} r_k - \sum_{k \in \mathcal{K}} p_k$$

The minimal end-time of activity A is then greater than $\max_{k \in \mathcal{K}} d_k + p_A - \Delta$. Which leads to:

$$emin_A \geq \min_{k \in \mathcal{K} \cup \{A\}} r_k + \sum_{k \in \mathcal{K} \cup \{A\}} p_k$$

5.2.2 JPS-Based Deduction Rules

In this section, we propose some deduction rules based on JPS and we prove that they are equivalent to 5.2.1.From now on, $r_i^{'}$ and $emin_i^{'}$ will denote the values of the release-date and of the maximal end-time of activity A_i deduced by rules 23 and 24.

Proposition 25 *Let A_i be a non-interruptible activity then,*

$$\left[r_i + p_i + \sum_{d_j \leq d_k} p_j^* > d_k \right] \Rightarrow \left[r_i \geq \max_{A_j \ / \ d_j \leq d_k} e_{JPS}(A_j) \right]$$

Proof: This proposition corresponds to the characterization of the edge-finder proposed in [Carlier 90].

Definition: *Given an interruptible activity A_i, let $emin_i^{''}$ be:*

$$emin_i^{''} = \max \left(emin_i, \max_{\substack{k \ / \\ r_i + p_i + \sum_{d_j \leq d_k} p_j^* > d_k}} \left(r_i + p_i + \sum_{d_j \leq d_k} p_j^* \right) \right)$$

Proposition 26 *For any activity A_i, $emin_i^{'} = emin_i^{''}$.*

Proof: [Carlier 90] provides a demonstration of this proposition when A_i is a **non-interruptible** activity. We focus here on activities A_i which are **interruptible**. Let us prove that $emin_i^{''} \geq emin_i^{'}$ and that $emin_i^{'} \geq emin_i^{''}$.

- If $\mathbf{emin_i^{'} < emin_i^{''}}$ let us find a contradiction: Since $emin_i^{'} \geq emin_i$, $emin_i^{''}$ is strictly greater that $emin_i$. Let then A_k be the activity such that:

  $$emin_i^{''} = r_i + p_i + \sum_{d_j \leq d_k} p_j^*$$

  Let us consider the latest date t before r_i such that:

  1. JPS is full from t to r_i;
  2. either JPS is empty at date $t-1$ or an activity A_l such that $d_l > d_k$ is scheduled at $t-1$

  Let us then consider the set $\mathcal{K}$ defined as follow:

  $$\mathcal{K} = \left\{ A_j \mid \left\{ \begin{array}{l} r_j \geq t \\ d_j \leq d_k \end{array} \right. \right\}$$

Let us now rewrite the value of $emin_i^{''}$:

$$\begin{aligned} emin_i^{''} &= r_i + p_i + \sum_{d_j \leq d_k} p_j^* \\ &= t + (r_i - t) + \sum_{d_j \leq d_k} p_j^* \\ &= t + (r_i - t) + \sum_{d_j \leq d_k \ and \ r_j \geq t} p_j^* + \sum_{d_j \leq d_k \ and \ r_j < t} p_j^* \end{aligned} \tag{5.2}$$

1. Let us prove first that $\forall A_j$ such that $d_j \leq d_k$ and such that $r_j < t$, $p_j^* = 0$. If A_j is not finished on JPS at time r_i, it should have been executed at time $t-1$ since either JPS is empty at date $t-1$ or an activity A_l such that $d_l > d_k$ is scheduled at $t-1$. Absurd. Equation 5.2 then becomes:

$$\begin{aligned} emin_i^{''} &= t + (r_i - t) + \sum_{d_j \leq d_k \ and \ r_j \geq t} p_j^* \\ &= t + (r_i - t) + \sum_{A_j \in \mathcal{K}} p_j^* \end{aligned} \tag{5.3}$$

2. Let us prove that over the interval $[t \ r_i)$, JPS is full and the activities scheduled in this interval belong to $\mathcal{K}$: If one unit of an activity $A \notin \mathcal{K}$ is scheduled in $[t \ r_i)$, since $d_A \leq d_k$ (definition of t), we know that $r_A < t$. This is absurd since, either JPS is empty at date $t-1$ or an activity A_l such that $d_l > d_k$ is scheduled at $t-1$; which means that A could have been scheduled at $t-1$. This means that $r_i - t$ corresponds to the sum of the lengths of the pieces of activities in $\mathcal{K}$ scheduled before r_i. Consequently, the following equation holds:

$$(r_i - t) + \sum_{A_j \in \mathcal{K}} p_j^* = \sum_{A_j \in \mathcal{K}} p_j$$

   Equation 5.3 then becomes:

$$emin_i^{''} = t + \sum_{A_j \in \mathcal{K}} p_j \tag{5.4}$$

3. Let us now prove that $t = \min_{j \in \mathcal{K}} r_j$. It is obvious that $t \geq \min_{j \in \mathcal{K}} r_j$ since at time t on JPS one unit of an activity of $\mathcal{K}$ is scheduled (cf. item 2). According to the definition of $\mathcal{K}$, it is also true that $t \leq \min_{j \in \mathcal{K}} r_j$. Equation 5.4 then becomes:

$$emin_i^{''} = r_i + \sum_{A_j \in \mathcal{K}} p_j$$

According to the definition of $emin_i^{'}$, $emin_i^{'} \geq r_i + \sum_{A_j \in \mathcal{K}} p_j$. This in turn leads to $emin_i^{'} \geq emin_i^{''}$. Absurd

- If $\mathbf{emin_i^{''} < emin_i^{'}}$, let us find a contradiction: Since $emin_i^{'} > emin_i^{''} \geq emin_i$, an adjustment has been done. Let $\mathcal{K}$ be the set of activities that triggered the deduction and let A_k be an activity in $\mathcal{K}$ such that $d_k = \max_{j \in \mathcal{K}} d_j$.

– If $\mathbf{r_i + p_i + \sum_{j \neq i \text{ and } d_j \leq d_k} p_j^* \leq d_k}$, then, there is enough space between r_i and d_k to schedule (a) the pieces of all activities not finished at time r_i on JPS whose due-dates are lower than d_k and (b) the activity A_i. Since JPS is an acceptable preemptive schedule, and since all activities in $\mathcal{K}$ have a due-date lower than d_k, A_i and all activities of $\mathcal{K}$ can be scheduled before d_k. This is absurd since $\mathcal{K}$ is set of activities that triggered the deduction.

– We know then, that $\mathbf{r_i + p_i + \sum_{j \neq i \text{ and } d_j \leq d_k} p_j^* > d_k}$. According to the definition of $emin_i''$, $emin_i'' \geq \delta$ where,

$$\delta = r_i + p_i + \sum_{d_j \leq d_k} p_j^*$$

According to the building of JPS, δ is the minimal makespan amongst all acceptable preemptive schedules of the set of activities $\Omega = \{A_j \ / \ d_j \leq d_k\}$. Let us recall the value of $emin_i'$:

$$emin_i' = \min_{k \in \mathcal{K} \cup \{A\}} r_k + \sum_{k \in \mathcal{K} \cup \{A\}} p_k$$

This value is a lower bound of the minimal makespan amongs all acceptable preemptive schedules of the set of activities $\mathcal{K}$. Since $\mathcal{K} \subset \Omega$ (see definition of d_k), $emin_i' \leq \delta$. This leads in turn to $emin_i' \leq emin_i''$. Absurd.

5.2.3 An Algorithm for the Mixed Edge-Finder

The following algorithm uses the deduction rules based on (JPS) to update-time bounds of activities:

1. Sort activities in increasing order of due-dates (we suppose in the remaining part of this algorithm that the activities $A_1, A_2, \cdots, A_n$ are sorted in increasing order of due-dates).
2. Build the JPS of $A_1, A_2, \cdots, A_n$.
3. Iterate on activity A_i for $i = 1$ to $i = n$:
   (a) Compute the values of p_j^* (for $j = 1$ to n).
   (b) Compute $\Psi = \sum_{k \neq i} p_k^*$
   (c) Iterate on activity A_j for $j = 1$ to n:
      i. Update the value of Ψ to $\Psi - p_j^*$.
      ii. Perform the basic edge-finding test (3.2):

$$r_i + p_i + \Psi > d_j \tag{5.5}$$

If it succeeds, consider the following alternative:

- Either A_i is a non-interruptible activity and then update $emin(A_i)$ according to 25
- Or A_i is an interruptible activity and then update $emin(A_i)$ according to 26

End If

End Iterate

End Iterate

5.3 A Characterization of the Mixed Edge-Finder

We have proposed in section 5.1 a characterization of the edge-finder. The aim of this section is to prove that this characterization also suits to the mixed edge-finder. Let $r_1^{'}, \cdots, r_n^{'}$ and $emin_1^{'}, \cdots, emin_n^{'}$ be the release-dates and the minimal end-times deduced by the primal version of the mixed edge-finder.

Proposition 27 *If A_i is a non-interruptible activity, $r_i^{'}$ is the earliest date such that $\exists \mathcal{S}$, an acceptable $\overline{A}_i$-preemptive schedule on which $s_{\mathcal{S}}(A_i) = r_i^{'}$.*

Proof: Same proof than for proposition 28.

Proposition 28 *If A_i is an interruptible activity, $emin_i^{'}$ is the earliest date such that $\exists \mathcal{S}$, an acceptable preemptive schedule on which $e_{\mathcal{S}}(A_i) = emin_i^{'}$.*

Let us prove first (proposition 29) that, $\forall \Omega$ an acceptable preemptive schedule, $e_{\Omega}(A_i) \geq emin_i^{'}$ we will then (proposition 30)prove that $\exists \mathcal{S}$, an acceptable preemptive schedule such that $e_{\mathcal{S}}(A_i) = emin_i^{'}$.

Proposition 29 *$\forall \Omega$ an acceptable preemptive schedule, $e_{\Omega}(A_i) \geq emin_i^{'}$.*

Proof: Two cases are possible: **either** $emin_i^{'} = emin_i$ and then 29 obviously holds **or** $emin_i^{'} > emin_i$. Let us focus on the second case. Since 19 holds, a subset $\mathcal{K}$ of activities has triggered the adjustment achieved by the edge-finder:

$$\exists \mathcal{K} \subset \{A_1, \cdots, A_{i-1}, A_{i+1}, \cdots, A_n\} \; / \; \min_{k \in \mathcal{K} \cup \{A_i\}} r_k + \sum_{k \in \mathcal{K} \cup \{A_i\}} p_k > \max_{k \in \mathcal{K}} d_k$$

Let us suppose that $\exists \Omega$ an acceptable preemptive schedule such that, A_i does not end after all activities in $\mathcal{K}$. Since Ω is acceptable, the release-dates and due-dates are met and then, between the release-date of $\mathcal{K} \cup \{A_i\}$ and the due-date of $\mathcal{K}$, all activities in $\mathcal{K}$ and A_i are scheduled; which leads to:

$$\min_{k \in \mathcal{K} \cup \{A_i\}} r_k + \sum_{k \in \mathcal{K} \cup \{A_i\}} p_k \leq \max_{k \in \mathcal{K}} d_k$$

Absurd.

Proposition 30 *$\exists \mathcal{S}$, an acceptable preemptive schedule such that $e_{\mathcal{S}}(A_i) = emin_i^{'}$.*

Sketch of the proof: Since the proof is very similar to the demonstration of proposition 22, we only present the basic idea on which it relies. Let us consider the set of activities $\{A_1, \cdots, A_{i-1}, A_{i+1}, \cdots, A_n\}$ and let us build the JPSB (backward) of these activities up to the date $emin_i^{'}$. The remaining pieces of activities are then scheduled according to JPSF (forward). We then try to schedule A_i before $emin_i^{'}$. If this is not possible, we are then able to find an activity A_k such that:

$$\begin{cases} r_i + p_i + \sum_{j \, neq \, i \; and \; d_j \leq d_k} p_j^* > d_k \\ r_i + p_i + \sum_{j \, neq \, i \; and \; d_j \leq d_k} p_j^* > emin_i^{'} \end{cases}$$

which leads to a contradiction.

Chapter 6

An Approximation Algorithm for Job-Shop Scheduling

6.1 Introduction

[1] We focus here on one of the best known scheduling problems, viz., the **Job Shop Scheduling Problem** (JSSP). In the JSSP one is given a set of jobs and a set of resources. Each job consists of a set of activities that must be processed in a given order. Furthermore, each activity is given an integer processing time and a resource on which it has to be processed. Once an activity is started, it is processed without interruption and a resource can process at most one activity at a time. A schedule specifies a start time for each activity. In the optimization variant of the JSSP one is asked to find a schedule that minimizes the **makespan**, i.e., the maximum completion time of the activities. For a more formal definition of the JSSP see for instance [Garey 79].

Optimization algorithms for the JSSP proceed by branch-and-bound. Most lower bound calculations on the makespan are based on the relaxation of the capacity constraints of all resources except one, resulting in a single-machine scheduling problem with release times and deadlines. This single-machine bound has been strengthened by a number of people among which [Carlier 89], [Brucker 92], [Nuijten 94], [Caseau 94b] and [Carlier 94]. In the last decade, many variants of local search have been proposed for the solution of the JSSP. Iterative improvement, simulated annealing, and threshold accepting were used by [Aarts 94]. Tabu search was used by [Taillard 89], [Barnes 91], and [Nowicki 93]. Genetic local search was used by [Aarts 94]. [Adams 88] developed an approach that is a combination of a constructive approach and iterative improvement. Variants of this approach are the **bottle-t** and **shuffle** algorithms proposed by [Applegate 91].

In constraint-based scheduling the usual way of solving problems is by using systematic tree search. In this way constraint-based scheduling stays close to optimization algorithms. In

[1] A large part of this chapter is cited from [Baptiste 95d]

this paper we present both constraint-based optimization algorithms and a constraint-based **approximation** algorithm. This approximation algorithm uses two important ideas from [Applegate 91] and [Nuijten 94]. The crucial component of ILOG SCHEDULE for this approach is its extensive propagation for disjunctive constraints as discussed by [Baptiste 95b].

6.2 Some Optimization Algorithms

In ILOG SCHEDULE, the propagation of temporal constraints, i.e., precedence relations with or without minimal and maximal delays between activities, is **complete** [Baptiste 95a]. This means that propagating the constraints is sufficient to determine whether a set of temporal constraints is consistent. The propagation also determines earliest and latest start and end times for activities that are globally consistent with all the temporal constraints.

As a result, solutions to the JSSP can be obtained by sequencing all the activities that require a common resource. Given an upper bound for the makespan, the following algorithm either finds a solution to the JSSP or proves that no solution exists:

1. Select a resource among the resources required by unordered activities.
2. Select the activity to execute first (or last) among the unordered activities that require the chosen resource. Post and propagate the corresponding precedence constraints. If an inconsistency is detected, a backtrack occurs. Keep the other activities as alternatives to be tried upon backtracking.
3. Iterate step 2 until all the activities that require the chosen resource are ordered.
4. Iterate steps 1 to 3 until all the activities that require a common resource are ordered.

To minimize makespan, this search algorithm merely needs to be encapsulated in a `IlcMinimize` SOLVER statement. `IlcMinimize` is a SOLVER function which receives as its arguments a search algorithm and a criterion to minimize. As long as the search algorithm succeeds in generating a solution to the problem, `IlcMinimize` adds a new constraint stating that the value of the criterion must be strictly smaller than the value of the best solution found so far. When the search algorithm fails, i.e., reports that there is no better solution, it is known that the best solution found is optimal.

An alternative to `IlcMinimize` consists in performing a binary search that dichotomizes the interval where the possible values of the makespan occur. Given a lower bound lb and an upper bound ub for the optimal makespan, we try to solve the problem with a new constraint stating that the makespan must be at most $(lb + ub)/2$. If this succeeds, ub becomes the makespan of the solution that was found; if this fails, lb becomes $((lb + ub)/2) + 1$. When lb and ub are equal, the optimal makespan has been found.

Coming back to the algorithm described above; heuristics are used to select the resource (step 1) and the activity to execute either first or last among the unordered activities that require the resource (step 2). We have implemented the following heuristics.

Resource Selection. The **slack** of each resource is computed. The slack is defined as the minimal difference between **supply** and **demand** over each time interval $[r_i, d_i)$, where r_i and d_i denote the minimal earliest start time and the maximal latest end time of any of the unordered activities A_i. The resource supply over $[r_i, d_i)$ is $d_i - r_i$, which reflects the fact that the resource can perform only one activity at a time. The demand over the interval $[r_i, d_i)$ is the sum of the durations of the activities that must execute between r_i and d_i. The resource with the smallest slack is selected.

Activity Selection. There are two things to consider in this step. First, we need to decide whether we select an activity to execute first or an activity to execute last among the unordered activities, and second which activity is indeed selected. Several alternatives may be considered: (a) always selecting an activity to execute first; (b) always selecting an activity to execute last; (c) deciding between first and last with respect to a dynamic criterion like the number of activities that can be scheduled first and the number of activities that can be scheduled last. Indeed, the earliest and latest start and end times computed by constraint propagation can be used to determine that some activities cannot be first and that some activities cannot be last [Carlier 89] [Baptiste 95b]. When the number of activities that can be first is smaller than the number of activities that can be last, we may rather select one of the possible firsts to avoid branching on a high number of possibilities. Similarly, when the number of activities that can be last is smaller than the number of activities that can be first, we shall rather select one of the possible lasts. This may be particularly important when there are very few solutions: it is less likely to make a mistake when a choice is made between few alternatives. However, this argument may be balanced by the fact that we noticed that scheduling always from the same end may allow better exploitation of the constraint propagation that takes place. As the choice was unclear, we tried all three alternatives (a), (b), and (c).

The activity to schedule first is selected according to the following EST-LST rule: the activity with the smallest Earliest Start Time is chosen (EST); in case of ties, the activity with the smallest Latest Start Time (LST) is chosen. A symmetric heuristic, LET-EET (Latest End Time - Earliest End Time) applies when selecting an activity to execute last. When the number of possible firsts equals the number of possible lasts and strategy when (c) is used, the tie is broken by looking at the difference between the best activity F_1 and the second best activity F_2 according to the EST-LST rule, and at the difference between the best activity L_1 and the second best activity L_2 according to the LET-EET rule. If the difference between F_1 and F_2 (according to EST-LST) is greater than the difference between L_1 and L_2, F_1 is selected to execute first; otherwise, L_1 is selected to execute last.

Computational results. Tables 6.1, 6.2, and 6.3 give the results obtained on the ten 10x10 JSSP instances used in the computational study of [Applegate 91]. Table 6.1 gives the results obtained by always selecting an activity to execute first. Table 6.2 gives the results obtained by always selecting an activity to execute last. Table 6.3 gives the results obtained by dynamically deciding between first and last. In both cases, a binary search that dichotomizes on the possible

values of the makespan is used. The columns "BT" and "CPU" give the total number of backtracks and CPU time needed to find an optimal solution and prove its optimality. Columns "BT(pr)" and "CPU(pr)" give the number of backtracks and CPU time needed for the proof of optimality. Column "CPU(1)" gives the CPU time needed to find the first solution, including the time needed for stating the problem and performing initial constraint propagation. All CPU times in this paper are given in seconds on a RS6000 workstation. We remark that none of the three strategies dominates the others. Over the ten instances, the total number of backtracks is 579711 for strategy (a), 416971 for strategy (b) and 501174 for strategy (c). We remark that for each of the three strategies, the difference between different instances is very important.

Instance	CPU(1)	BT	CPU	BT(pr)	CPU(pr)
MT10	.4	69758	1076.4	7792	126.8
ABZ5	.3	17636	218.1	5145	62.6
ABZ6	.3	898	15.2	291	4.7
LA19	.3	21910	293.3	5618	75.8
LA20	.3	74452	845.9	22567	249.2
ORB1	.4	13944	222.0	5382	84.7
ORB2	.3	114715	1917.2	30519	500.9
ORB3	.4	190117	3193.8	25809	449.0
ORB4	.3	64652	1131.2	22443	395.2
ORB5	.3	11629	172.8	3755	55.0

Table 6.1: Results on ten 10x10 instances of the JSSP with strategy (a)

Instance	CPU(1)	BT	CPU	BT(pr)	CPU(pr)
MT10	.4	14076	199.1	4181	58.6
ABZ5	.3	14747	205.5	3562	51.2
ABZ6	.3	2186	28.7	822	10.4
LA19	.3	25906	350.9	7935	109.9
LA20	.3	53773	612.3	18044	193.5
ORB1	.3	95802	1569.6	23107	385.8
ORB2	.3	33557	525.7	12385	194.2
ORB3	.4	151136	2451.8	35945	571.5
ORB4	.3	14524	174.7	1841	22.4
ORB5	.3	11264	152.5	2522	34.8

Table 6.2: Results on ten 10x10 instances of the JSSP with strategy (b)

6.3 An Approximation Algorithm

The results obtained by the optimization algorithms are quite good. However, the overall CPU time varies a lot from one instance to the other. By examining a number of runs more closely, we found two explanations for that. First, the dichotomizing strategy may lead to proving several times the quasi-optimality of a solution. Second, and more surprisingly, we found that a significant portion of the CPU time is spent finding solutions relatively far from the optimal

Instance	CPU(1)	BT	CPU	BT(pr)	CPU(pr)
MT10	.3	12844	188.1	4735	66.9
ABZ5	.3	17992	247.5	4519	62.2
ABZ6	.3	1116	16.4	312	4.7
LA19	.3	22235	312.7	6561	92.0
LA20	.3	95611	1126.1	20626	231.1
ORB1	.3	24749	423.9	6261	109.6
ORB2	.3	37982	647.7	14123	234.4
ORB3	.3	269697	4160.9	22138	351.6
ORB4	.3	9770	125.5	1916	24.1
ORB5	.3	9178	130.1	2658	37.4

Table 6.3: Results on ten 10x10 instances of the JSSP with strategy (c)

value. For example, on the MT10 instance, strategy (a) spends nearly half of the total CPU time finding a schedule of makespan 968, while the optimal value is 930. Our interpretation of this phenomenon is that when the upper bound is much higher than the optimal makespan, there are many solutions, making it easy to find one. Furthermore, when the upper bound is very close to the optimal makespan, constraint propagation becomes very effective in pruning the search space. Then, although it is not **easy** to find a solution, constraint propagation provides reliable guidance. However, for intermediate values of the upper bound, it is fairly difficult to find a solution and constraint propagation does not provide much guidance. This increases the probability of taking a wrong decision and as a systematic search strategy is used, it may take long to recover from such a mistake.

An alternative to an optimization algorithm is an approximation algorithm. As discussed in Section 6.1, many types of approximation algorithms have been applied to the JSSP. We used two important ideas from [Applegate 91] and [Nuijten 94] to develop our own approximation algorithm.

[Applegate 91] use the **shuffle** procedure to improve solutions to the JSSP. The basic idea is to fix a number of decisions, based on the best solution found so far, and search for an optimal schedule among those that respect those decisions. This procedure gives very good results, especially when effective constraint propagation techniques are used. Indeed, the propagation of the imposed decisions results in a drastic reduction of the search space and thus enables a fast resolution of the remaining subproblems.

[Nuijten 94] uses a randomized procedure to solve a wide range of scheduling problems. Of this procedure we use the way to escape from dead ends. The employed approach consists of two parts. First, chronological backtracking is used in order to solve the instance at hand. However, if this does not lead to a solution after a reasonable number of backtracks, the search is restarted. A problem then is to direct the search along a path different from the ones followed previously. Therefore, the restarting of the search is combined with randomized selection strategies. In this way, the probability of following the same search path more than once is very small since the number of possible paths is usually very large.

Our algorithm combines the two ideas as follows. At each step, a number of ordering

decisions are kept. As in [Applegate 91], we search for an optimal schedule among those that respect those decisions. However, the decisions that are kept are randomly selected and the search is stopped after a given number of backtracks. More precisely, our algorithm works as follows.

1. Generate an initial solution by solving the constraint satisfaction problem in which the makespan is not constrained. The propagation of temporal constraints is complete and thus a first solution is obtained quickly and without backtracking.

2. For at most N iterations without improvement, the following subroutine is applied:

   - Constrain the makespan to be lower than the best known makespan.
   - Keep randomly some ordering decisions from the previous schedule. For each pair of activities (A B) consecutively scheduled on the same resource, the ordering decision "A before B" is kept with a probability p.
   - Search for a solution using a limited number of backtracks NB: a `IlcMinimize` is launched on a problem-solving algorithm equivalent to the one described in Section 6.2. However, the `IlcMinimize` is bound to return after NB backtracks. If a new solution is found then it is recorded as the new best solution, and the algorithm restarts step 2. Otherwise, a new iteration is attempted.

3. Step 2 terminates when the solution has not been improved for N iterations. Then the probability p is decreased. If the probability p is greater than a given threshold p_0, proceed to step 2. Otherwise return the best solution found so far.

We further refined the algorithm by introducing four different activity selection heuristics. Each of these heuristics is tried in turn in step 2 of the algorithm. Step 2 terminates when N iterations of each of the heuristics have failed to produce a solution better than the best available solution. The four activity selection heuristics are the following.

- The first one is the strategy referred as (c) in Section 6.2. This strategy dynamically determines whether to select an activity to schedule first or last. The activity to schedule first is chosen according to the EST-LST rule. The activity to schedule last is chosen according to the LET-EET rule. Ties are broken as explained in Section 6.2.

- The second strategy also dynamically determines whether to select an activity to schedule first or last. However, the activity to schedule first is chosen according to another rule being the EDD-EST rule. With this rule the activity with the smallest latest end time is chosen, and in case of ties, preference is given to the activity with the smallest earliest start time. The symmetric rule, LRD-LET, is used for determining the activity to schedule last.

- The third strategy always selects an activity to execute first. This activity is selected randomly among the activities that can be scheduled first.

- The fourth strategy always selects an activity to execute last. This activity is selected randomly among the activities that can be scheduled last.

Computational results We first applied our approximation algorithm to the ten instances of [Applegate 91]. For each instance we did five runs using different seeds for the random number generator. We choose $N = 10$, $NB = 100$, $p = 0.6$, and have p decreased by steps of 0.6 down to $p_0 = 1/(2 \cdot m)$, where m is the number of resources. The first run we did found an optimal solution for seven out of the ten instances. The second run found an optimal solution for two of the three remaining instances. The average mean relative error (MRE) over three runs was 0.1%. This means that on average, the algorithm provided a solution within 0.1% of the optimal solution. The CPU time for executing the complete algorithm varied between 90 and 180 seconds. On average, the best solution was found after 72.5 seconds of CPU time. This is of course much smaller than the CPU times obtained with the optimization algorithm.

Instance	LB/UB	AVG(5)	BEST(5)	CPU
MT10	930	930	930	187.6
LA02	655	655	655	4.1
LA19	842	843	842	173.8
LA21	1046	1061	1046	610.9
LA24	935	941	940	430.8
LA25	977	980	977	376.8
LA27	1235	1269	1254	1012.1
LA29	1130/1157	1214	1196	1320.7
LA36	1268	1269	1268	633.8
LA37	1397	1397	1397	233.8
LA38	1196	1216	1207	1071.9
LA39	1233	1235	1233	805.6
LA40	1222	1234	1229	947.2

Table 6.4: Results on thirteen instances of the JSSP

We also applied the algorithm to the thirteen instances used by [Vaessens 94]. The parameters were unchanged except for p_0 that was set to $1/m$ to acomodate instances with a high number of jobs in a reasonable amount of time. Table 6.4 gives the results obtained. In this table, the column "LB/UB" gives the best known lower and upper bound on the minimum makespan. If only a single number is given, both bounds coincide resulting in an optimal value of the makespan. Column "AVG(5)" provides the average makespan obtained over five runs of the algorithm. Column "BEST(5)" provides the best makespan obtained after the same five runs. Column "CPU" provides the average CPU time.

These results significantly improve on those reported in [Nuijten 94] for a similar approach. The average MRE over five runs is 1.20% and the MRE for the best of five runs is 0.72%. Even if we use the lower bounds used in [Vaessens 94], we get an MRE of 0.92% for the best of five runs. Only one of the algorithms used in [Vaessens 94] achieves an MRE smaller than 1 % percent, namely the tailored taboo search algorithm of [Nowicki 93] for which the MRE is 0.54%.

6.4 Combining Approximation and Optimization

We finally combined the approximation and optimization algorithms as follows. When the approximation algorithm terminates, the optimization algorithm is launched using `IlcMinimize` and the strategy referred to as (c) in Section 6.2. Table 6.5 gives the average results obtained over three runs of the ten 10x10 instances of [Applegate 91]. The total number of backtracks over the ten instances is 215256, which significantly improves on the results reported in Section 6.2.

Instance	CPU(1)	BT	CPU	BT(pr)	CPU(pr)
MT10	.3	13684	235.8	4735	67.3
ABZ5	.3	19303	282.1	4519	61.3
ABZ6	.3	6227	100.6	312	4.7
LA19	.4	18102	269.5	6561	91.0
LA20	.3	40597	496.7	20626	227.2
ORB1	.3	22725	407.3	6261	108.0
ORB2	.3	31490	507.1	14123	228.7
ORB3	.4	36729	606.1	22138	342.6
ORB4	.3	13751	213.7	1916	23.7
ORB5	.3	12648	210.9	2658	36.5

Table 6.5: Results on ten 10x10 instances of the JSSP

6.5 Conclusion

We showed that the algorithms we presented perform well. In particular, on a well studied set of instances one single approach known to date performs slightly better than our constraint-based approximation algorithm. We, furthermore, showed that by using this approximation algorithm, we can improve the performance of the optimization algorithms. We remark that this combination allowed us to solve the LA21 instance to optimality. To the best of our knowledge, the optimal makespan for this instance was not yet published anywhere, although we know that various people succeeded to obtain the same result. However, the number of backtracks that the proof of optimality took, being 3.955.036, has inspired us to do research in which we focus on improving the propagation algorithms that we use. We expect that the theoretical results we obtained in that respect, will lead to even better computational results fairly soon.

Chapter 7

The Preemptive Job-Shop Scheduling Problem

In this chapter, we study the Preemptive Job-Shop Scheduling Problem (PJSSP), an extension of the well-known Job-Shop Scheduling Problem (JSSP). Informally speaking, the only difference between the JSSP and the PJSSP is that "preemptive" schedules are allowed: Each activity can be interrupted to let a more urgent activity be executed (section 7.1 provides a formal definition of the PJSSP). Few work has been carried on the PJSSP in comparison with the JSSP. We are not actually aware of any previous attempt to solve significant instances of the PJSSP. Basically, we present here two algorithms to solve some instances of this problem: Section 7.3 presents an algorithm relying on the resource constraints described in chapter 4. Section 7.4 describes a specific constraint propagation algorithm for the PJSSP.

7.1 The PJSSP

In the PJSSP, one is given a set of jobs and a set of machines. Each job consists of a set of operations that must be processed in a given order. Furthermore, each operation is given an integer processing time and a machine on which it has to be processed. An operation can be interrupted. There is no restriction on the number of interruptions or on the duration of an interruption or on the duration of a chunk. A machine can process at most one operation at a time. Finally, a fixed scheduling horizon is given. A schedule specifies a start time for each operation. The problem is to find a schedule, if it exists, in which no machine processes more than one operation at a time and which meets the scheduling horizon and the order in which the operations must be processed.

Proposition 31 *The decision variant of the PJSSP is NP-complete in the strong sense.*

Proof: see [Garey 79].

7.2 A Dominance Criterion

One of the problems encountered when solving a preemptive scheduling problem is that to schedule an interruptible activity A, the basic type of decisions are:

- schedule A in an interval of time;
- do not schedule A in another interval of time;

Such a decision-making process may represent a very large amount of decisions (in the worst case the number of decisions may be proportional to the duration of the activity). It is then very interesting to reduce the search space thanks to a dominance criterion.

Definitions: Let us introduce some more notations:

- Given an activity A, a schedule $\mathcal{S}$, let $A_{\mathcal{S}}$ be the set of time point at which A is executed on $\mathcal{S}$.

$$A(\mathcal{S}) = \{t \backslash\ A \textit{ executed at } t \textit{ on } \mathcal{S}\}$$

- Let $s_{\mathcal{S}}(A)$ and $e_{\mathcal{S}}(A)$ be respectively the start time and end time of A on $\mathcal{S}$.

$$s_{\mathcal{S}}(A) = \min(A_{\mathcal{S}})$$
$$e_{\mathcal{S}}(A) = \max(A_{\mathcal{S}}) + 1$$

Let M be the optimal makespan of a given instance of the PJSSP. The following proposition holds:

Proposition 32 *There exists $\mathcal{S}$, an optimal schedule (i.e., whose makespan is equal to M) such that, for all pair of activities (A, B) requiring the same resource,* **either** *A preempts B* **or** *B preempts A* **or** *A and B do not preempt each other.*

Proof: Let ω be an optimal schedule of all resources. Let $A_1^i, A_2^i, \cdots, A_n^i$ be the n activities requiring the machine i and let us denote by $\alpha_1^i, \alpha_2^i, \cdots, \alpha_n^i$, n activities such that:

- release date of $\alpha_k^i = s_\omega(A_i^k)$
- due date of $\alpha_k^i = e_\omega(A_i^k)$
- duration of α_k^i = duration of A_i^k.

Since there exists, by construction, an acceptable schedule of the activities $\alpha_1^i, \alpha_2^i, \cdots, \alpha_n^i$, the Jackson Preemptive Schedule of these activities can be built. Let $\mathcal{S}^i$ be this schedule. Let then $\mathcal{S}$ be the schedule made out of the m one-machine schedules $\mathcal{S}^1$, $\mathcal{S}^2, \cdots, \mathcal{S}^m$. $\mathcal{S}$ is acceptable and optimal for the following reasons:

1. **Precedence constraints are satisfied.** Let A and B be two activities of the same job such that $A \prec B$. Since ω is an acceptable schedule, $e_\omega(A) \leq s_\omega(B)$. According to the definition of $\mathcal{S}$, $e_{\mathcal{S}}(A) \leq e_\omega(A)$ and $s_{\mathcal{S}}(B) \geq s_\omega(B)$; which leads to $e_{\mathcal{S}}(A) \leq s_{\mathcal{S}}(B)$.
2. **Resource constraints are satisfied.** Obvious by construction.

3. **Optimality.** Same proof than for item (1).

Recall that JPS "schedules" as soon as possible the unscheduled activity of minimal due-date. Hence, an activity A can be interrupted on $\mathcal{S}$ by B if and only if the due-date of B is lower (actually strictly lower because of the definition of the due-dates of activities α_k^i) than the due-date of A. Which means also that for all pair of activities (A, B) requiring the same resource, **either** A preempts B on $\mathcal{S}$ **or** B preempts A on $\mathcal{S}$ **or** A and B do not preempt each other on $\mathcal{S}$.

Definitions:

- The previous proposition allows to reduce the search space to prioritized schedules. Let us denote by the integer $\pi_{\mathcal{S}}(A)$ the priority level of each activity A on a schedule $\mathcal{S}$. Given two activities A and B (requiring the same resource), if A has a greater priority than B, then B cannot interrupt A:

$$[\pi_{\mathcal{S}}(A) > \pi_{\mathcal{S}}(B)] \Rightarrow [B \textit{ cannot preempt } A]$$

  Note that this priority rule can be stated like a **constraint**: B is not allowed to execute in the interval [maximal start time of A, minimal end time of B).

- An activity is said to be available at time t if and only if its (job) predecessors are scheduled and ends before t.

Proposition 33 *There is an optimal schedule $\mathcal{S}$ such that $\forall A, B$ scheduled on the same machine,*

$$[B \textit{ preempts } A] \Rightarrow [B \textit{ not available before } s_{\mathcal{S}}(B)]$$

Proof: Consider the optimal schedule $\mathcal{S}$ build in the proof of proposition 32 and recall that each of the one-machine schedule $\mathcal{S}_i$ is a Jackson Preemptive Schedule. Since B preempts A, B has a due-date smaller than the one of A; which means that B could not be scheduled at the time points where A is executed. More formally, this can be written as follow:

$$[B \textit{ preempts } A] \Rightarrow [B \textit{ not available before } s_{\mathcal{S}}(B)]$$

7.3 A Search Algorithm Relying On Generic Preemptive Resource Constraints

We present in this section an algorithm to solve the PJSSP and we provide some computational results. An experimental comparison between the constraint propagation algorithms described in chapter 4 is carried out.

7.3.1 An Algorithm to Solve PJSSP

Given an instance of the PJSSP, the "priority constraint" described in section 7.2 allows us to build an algorithm searching for an acceptable optimal schedule:

- Compute an upper bound M of the optimal makespan (for instance the sum of the durations of all activities).
- Constrain the end-time of the last activity of each job to be strictly lower than M.
- Build chronologically the schedule:
  1. Let t be the earliest date such that there is an activity available at t.
  2. Select an activity A amongst the activities available at t.
  3. Compute $\mathcal{K}$, the set of unscheduled activities available at t on the same machine than A.
  4. Compute $t^{'}$, the earliest date strictly greater than t such that the sets of activities, requiring the same machine than A, respectively available at t and at $t^{'}$ differ ($t^{'}$ may for instance correspond to a release-date or to the end of a forbidden interval $\cdots$).
  5. Schedule A in the time interval $[t, t^{'})$ and post the constraint: "$\forall \alpha \in \mathcal{K} - \{A\}$, α is not available until A is fully scheduled" (this constraint is the priority constraint as described in section 7.2).
  6. Keep the other activities of $\mathcal{K}$ as alternatives to be tried upon backtracking.
  7. Iterate to step (1) until all activities are scheduled.
- If a solution is found, set M to the current value of the makespan and iterate; otherwise the optimal makespan is M, exit.

Proof of the Algorithm: Let us suppose that the algorithm looks for an acceptable schedule ending before the date M. We have to prove that (1) if the algorithm ends with a solution Ω, this solution is acceptable and (2) if the algorithm claims there is no solution, there is indeed no acceptable schedule ending before M.

1. **If the algorithm ends with a solution Ω, this solution is acceptable**. Obvious according to the construction of the algorithm.
2. **If the algorithm claims there is no solution, there is indeed no acceptable schedule ending before** M. We just have to prove that the optimal schedule $\mathcal{S}$ (cf. proposition 32) can be built by the algorithm. Let us consider the strategy which consists in selecting amongst the unscheduled activities available at the current date, the activity whose end-time on $\mathcal{S}$ is minimum. Let $\mathcal{S}^{'}$ be such a schedule. If $\mathcal{S} \neq \mathcal{S}^{'}$ let t be the earliest date at which $\mathcal{S}$ and $\mathcal{S}^{'}$ differ on a machine i. Let A be the activity scheduled at t on $\mathcal{S}$ and $A^{'}$ the activity scheduled at t on $\mathcal{S}^{'}$ (see figure 7.1). We can then deduce the two following results:

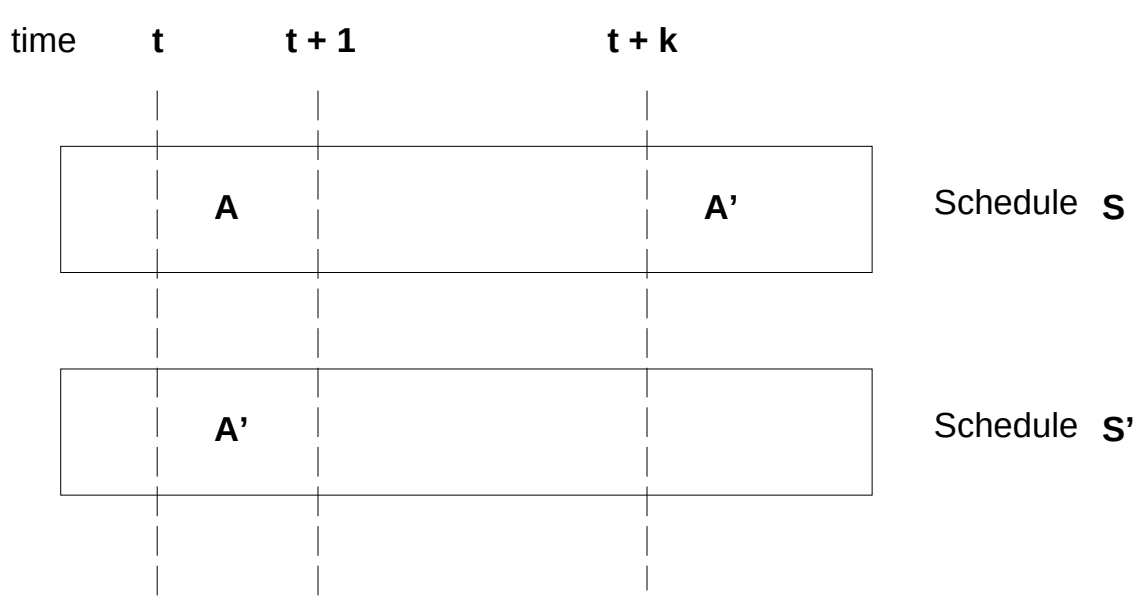

Figure 7.1: Schedules S and S'

- $A^{'}$ is scheduled at t on $\mathcal{S}^{'}$, which means that $\pi_{\mathcal{S}}(A^{'}) > \pi_{\mathcal{S}}(A)$;
- Since $\mathcal{S}$ and $\mathcal{S}^{'}$ are the same before t, activity $A^{'}$ must execute on $\mathcal{S}$ after the date t.

This means that, on schedule $\mathcal{S}$, there is an activity $A^{'}$, more urgent than A which could be scheduled *earlier* than it is (by using time-points used by A). Absurd, since proposition 33 holds.

This algorithm does not make any assumption about the **enumeration strategy**. After having tried several heuristics, it seems that the most promising consists in selecting the activity *Act* of minimal due-date. Note that for a "one-machine" problem, this strategy is optimal (see proposition 6).

In a seek of clarity the algorithm reaches the optimal value of a solution by restarting the search, each time a solution is found (with makespan M), with an additional constraint, requiring the makespan to be strictly smaller than M. When the algorithm fails, we know that the previous value of the makespan M is optimal. Actually, our algorithm, rather than decreasing step-by-step the makespan maximal value, dichotomizes on the domain of the SOLVER variable representing the makespan: when the domain of the makespan variable is [min max], we try to solve the problem with a new constraint stating that the makespan must be smaller than (or equal to) $(\min + \max)/2$. If this succeeds, max becomes the makespan of the solution that was found; if this fails, min becomes $1 + ((\min + \max)/2)$. When min and max are equal, the optimal solution has been found.

7.3.2 Computational Results

The search algorithm presented in the previous section has been tested with the four resource constraints presented in chapter 4, i.e.,

DISJ: the **DISJ**unctive algorithm,

SCF: the algorithm based on the **S**earch for a **C**ompatible **F**low in G,

AEC: the algorithm **A**djusting **E**dges **C**apacities,

GUTB: the algorithm performing a **G**lobal **U**pdating of **T**ime-**B**ounds.

The following pages provide tables related to different computational results. The columns of these tables represent:

Inst: the reference to the JSSP (or PJSSP) instance being solved. In parentheses, we provide the number of jobs n and the number of machines m. For example, CAR1(11x5) means that the CAR1 problem consists of 11 jobs and 5 machines.

N-P Mak: the optimal makespan in the non-preemptive case (JSSP).

P Mak: the optimal makespan in the preemptive case (PJSSP). When we have been unable to find the optimal makespan, we give a lower bound and an upper bound of this value. Note that, since the algorithm halts when it either failed to find a solution or to make a proof after a given number of backtracks, the gap between the upper-bounds and the lower-bounds is often large. In fact, these bounds could be improved by the use of a minimizing procedure instead of a dichotomizing one.

TF: the total number of backtracks required to solve the problem (solution + optimality proof).

TT: the total amount of time required to solve the problem (solution + optimality proof), in seconds on a RS6000 workstation.

IT: the number of iterations (restartings of the algorithm with a new makespan upper bound) necessary to solve the problem.

PF: the number of backtracks required for the optimality proof.

PT: the amount of time required for the optimality proof, in seconds on a RS6000 workstation.

TM: the total amount of memory used for solving the problem, in bytes.

Table 7.1 presents the optimal solution of one 6*6 instance of the PJSSP taken from [Muth 63] known as "ft06". The algorithm based on DISJ has been found unable to solve larger instances; which illustrates the poor efficiency of this very simple propagation algorithm.

Inst	N-P Mak	P Mak	TF	TT	IT	PF	PT	TM
ft06	55	54	6353	3.51	4	4775	2.6	60428

Table 7.1: ft06

Figure 7.2 displays the Gantt-chart of an optimal schedule of the 6*6 instance known as ft06. Each resource corresponds to a line of the Gantt-chart. The activities are represented by rectangles. Two numbers are written on the center of each piece of activity. The first one corresponds to the identifier of the job and the second, to the identifier of the activity inside the job. For instance the activity $\frac{3}{2}$ is the second activity of the job number 3. The dates at which each chunk of activity is scheduled are written on top of the schedule of each machine.

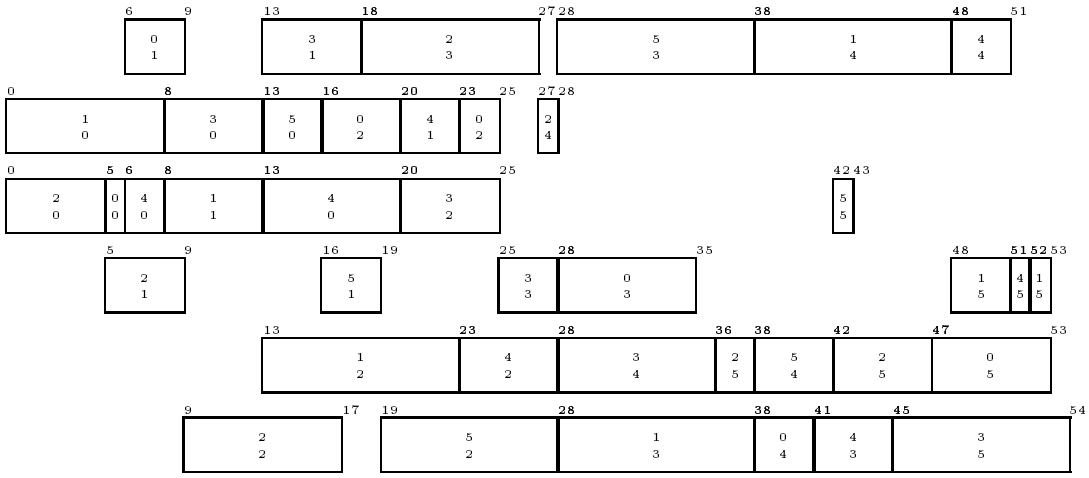

Figure 7.2: an optimal schedule of ft06nn

The remaining algorithms (SCF, AEC, GUTB) have been tested on 19 instances of the PJSSP:

- ft06 from [Muth 63].
- 8 problems from [Carlier 84] (chapter 4 section 5). These problems (CAR1 to CAR8) are flow-shop problems: the order of the machines is the same for each job. Note that we have not attempted to use the fact that these problems are flow-shop problems. We just treated them as job-shop problems.
- The ten first smaller problems (la01-la40) from [Lawrence 84].

Table 7.2 provides the computational results of the SCF algorithm on 19 instances of the PJSSP. It is clear that this algorithm is far more efficient than DISJ. The total cpu-time required to solve ft06 is divided by 14 and the total number of backtracks comes down from 4775 to 21! However, the average time spent per backtrack (which is a way to measure the experimental complexity of an algorithm) increases from 1300 backtracks per second to less than 90. Note that the search has been stopped each time the current number of backtracks to find a solution, with a makespan lower than or equal to a given value, has exceeded 10000.

The algorithm SCF has shown to be unable to solve 4 instances of the PJSSP: car3, car5, car6, car8. Table 7.3 reports computational experiments for AEC. It seems that the AEC algorithm performs better than SCF: Indeed, one single instance (car5) remains unsolved. It is

Inst	N-P Mak	P Mak	TF	TT	IT	PF	PT	TM
ft06	55	54	24	0.25	4	21	0.1	60428
la01	666	666	7	0.21	8	1	0	80528
la02	655	655	2697	15.77	8	1	0	88568
la03	597	597	266	1.88	9	3	0	88568
la04	590	567	8014	42.94	9	1	0	84548
la05	593	593	7	0.22	8	1	0	80528
la06	926	926	9	1.2	10	1	0	124748
la07	890	890	933	22.82	10	1	0	132788
la08	863	863	10	3.93	9	1	0	128768
la09	951	951	9	1.53	10	1	0	124748
la10	958	958	9	1.24	10	1	0	124748
car1	7038	6931	99	3.26	12	20	0.6	96608
car2	7166	7166	1910	52.12	12	194	5.3	96608
car3	7312	6963/7345	-	-	-	-	-	-
car4	8003	8003	104	4.5	12	1	0	104648
car5	7702	6430/8824	-	-	-	-	-	-
car6	8313	7852/8381	-	-	-	-	-	-
car7	6558	6218	55269	451.78	12	8496	67.2	84548
car8	8264	7934/8371	-	-	-	-	-	-

Table 7.2: Performance of the algorithm SCF on 19 instances of the PJSSP

interesting to note that [Baptiste 94] reports that in the non-preemptive case, car5 is the "most difficult" instance to solve amongst the other instances taken from [Carlier 84]. Let us also not note that in term of number of backtracks, AEC performs always better than SCF but, this is not the case in term of cpu-time excepted, for the most hard instances (car7 and of course for the instances unsolved by SCF).Note that the search has been stopped each time the current number of backtracks to find a solution, with a makespan lower than or equal to a given value, has exceeded 5000.

Table 7.4 presents the performance of the GUTB algorithm. This algorithm clearly outperforms AEC in term of backtracks and in term of cpu-time. It allows to solve the 19 instances of the PJSSP. The comparison between LUTB and SCF is close from the one between AEC and SCF. The number of backtracks is always much smaller with LUTB than with SCF but, in term of cpu time, many instances are solved faster with SCF. The cost of additional constraint propagation is not always balanced by the reduction of the search effort. However, it appears that on some "hard" instances, the extended propagation methods drasticly reduce the search space and allows to solve instances which could not be solved by other algorithms.

The success of the algorithms AEC and GUTB led us to build a new algorithm incorporating both of these techniques. However, the resulting resource constraint propagation algorithm has shown to perform poorly:

Inst	N-P Mak	P Mak	TF	TT	IT	PF	PT	TM
ft06	55	54	5	0.53	4	2	0.1	60428
la01	666	666	5	3.11	6	1	0.1	80528
la02	655	655	707	45.8	8	1	0.1	88568
la03	597	597	11	9.72	8	2	0.2	88568
la04	590	567	164	27.34	6	0	0	84548
la05	593	593	7	3.21	8	1	0.1	80528
la06	926	926	9	15.7	10	1	0.1	124748
la07	890	890	193	79.83	6	0	0	128768
la08	863	863	7	38.57	9	1	0	128768
la09	951	951	9	18.93	10	1	0.6	124748
la10	958	958	9	16.74	10	1	0	124748
car1	7038	6931	13	6.79	13	1	0	96608
car2	7166	7166	211	56.3	11	40	7.9	96608
car3	7312	7226	1591	157.38	8	1	0	104648
car4	8003	8003	29	25.28	13	1	0	104648
car5	7702	6430/8824	-	-	-	-	-	-
car6	8313	8151	11784	3329.38	11	2580	758.6	116708
car7	6558	6218	2103	280.08	12	311	41.1	84548
car8	8264	8171	6955	1325.34	12	1318	254.1	96608

Table 7.3: Performance of the algorithm (SCF + AEC) on 19 instances of the PJSSP

Inst	N-P Mak	P Mak	TF	TT	IT	PF	PT	TM
ft06	55	54	6	0.4	4	2	0	60428
la01	666	666	5	2.18	6	1	0	80528
la02	655	655	26	8.89	9	1	0	84548
la03	597	597	7	5.78	8	1	0	88568
la04	590	567	3	4.86	4	0	0	80528
la05	593	593	7	2.15	8	1	0	80528
la06	926	926	9	9.94	10	1	0.1	124748
la07	890	890	11	31.91	6	0	0	128768
la08	863	863	7	21.05	9	1	0	128768
la09	951	951	9	12.16	10	1	0.2	124748
la10	958	958	9	10.48	10	1	0	124748
car1	7038	6931	13	4.87	13	1	0	96608
car2	7166	7166	25	15.19	11	1	0.2	96608
car3	7312	7226	53	18.02	10	1	0	104648
car4	8003	8003	17	14	13	1	0	104648
car5	7702	7667	73135	10295.8	12	19265	2673.7	100628
car6	8313	8151	6499	1342.43	11	1648	318.4	112688
car7	6558	6218	820	102.62	12	122	14.7	84548
car8	8264	8171	3459	545.83	11	688	106.4	96608

Table 7.4: 19 instances of the PJSSP solved with the (SCF + GUTB) constraint propagation algorithm

- The total **cpu-time** required to solve the 19 instances increased from 13076 seconds for GUTB to 44045 seconds for GUTB + AEC.

- More surprisingly, the total number of backtracks performed to solve the 19 instances also increased! Less than 85000 backtracks were required by GUTB and more than 130000 for GUTB + AEC. In fact, for all instances except car5, the two algorithms perform similarly in term of backtracks but, concerning the instance car5 the number of backtracks is almost twice greater when combining GUTB and AEC. It seems that GUTB makes a "lucky" decision while GUTB+AEC fails to make this decision (since the combination of the two algorithm deduces different time-bounds and thus may alter the strategy).

Inst	N-P Mak	P Mak	TF	TT	IT	PF	PT	TM
ft10	930	890/915	-	-	-	-	-	-
abz5	1234	1159/1219	-	-	-	-	-	-
abz6	943	924	17578	3955.48	8	10879	2268.3	144848
la19	842	812	39286	7150.1	9	14184	2482.4	140828
la20	902	871	5494	1483.58	9	1627	463.8	144848
orb1	1059	991/1054	-	-	-	-	-	-
orb2	888	864	56863	11199.2	9	20203	3835.3	148868
orb3	1005	951/1254	-	-	-	-	-	-
orb4	1005	977/980	-	-	-	-	-	-
orb5	887	849	16457	4721.25	10	4496	1296.6	144848
orb6	1010	724/1312	-	-	-	-	-	-
orb7	397	389	27823	6217.16	8	13348	2930	148868
orb8	899	894	17773	3468.88	10	1	0.1	156908
orb9	934	917	12619	3322.41	9	3855	1054.6	148868
orb10	944	930	14775	1355.22	9	12144	981.9	148868

Table 7.5: 15 hard instances of the PJSSP including ten 10x10 JSSP instances used in the computational study of [Applegate 91] (SCF + GUTB)

The good behavior of the GUTB algorithm on the 19 small instances of the PJSSP led us to try to solve larger instances (table 7.5). Fifteen 10*10 instances have been studied including the 10 benchmarks from [Applegate 91] and the well-know ft10 instance. 9 of these instances have been solved (optimal solution and proof of optimality). GUTB has been found unable to solve the remaining instances (search has been stopped each time the current number of backtracks to find a solution, with a makespan lower than or equal to a given value, has exceeded 25000).

In a last experiment, we have tried to solve the famous ft10 instance. After more than 24 hours of cpu time (RS6000), we managed to find and prove the optimality of a solution with makespan 900 (see Table 7.6). Figure 7.3 displays an optimal schedule of the instance ft10.

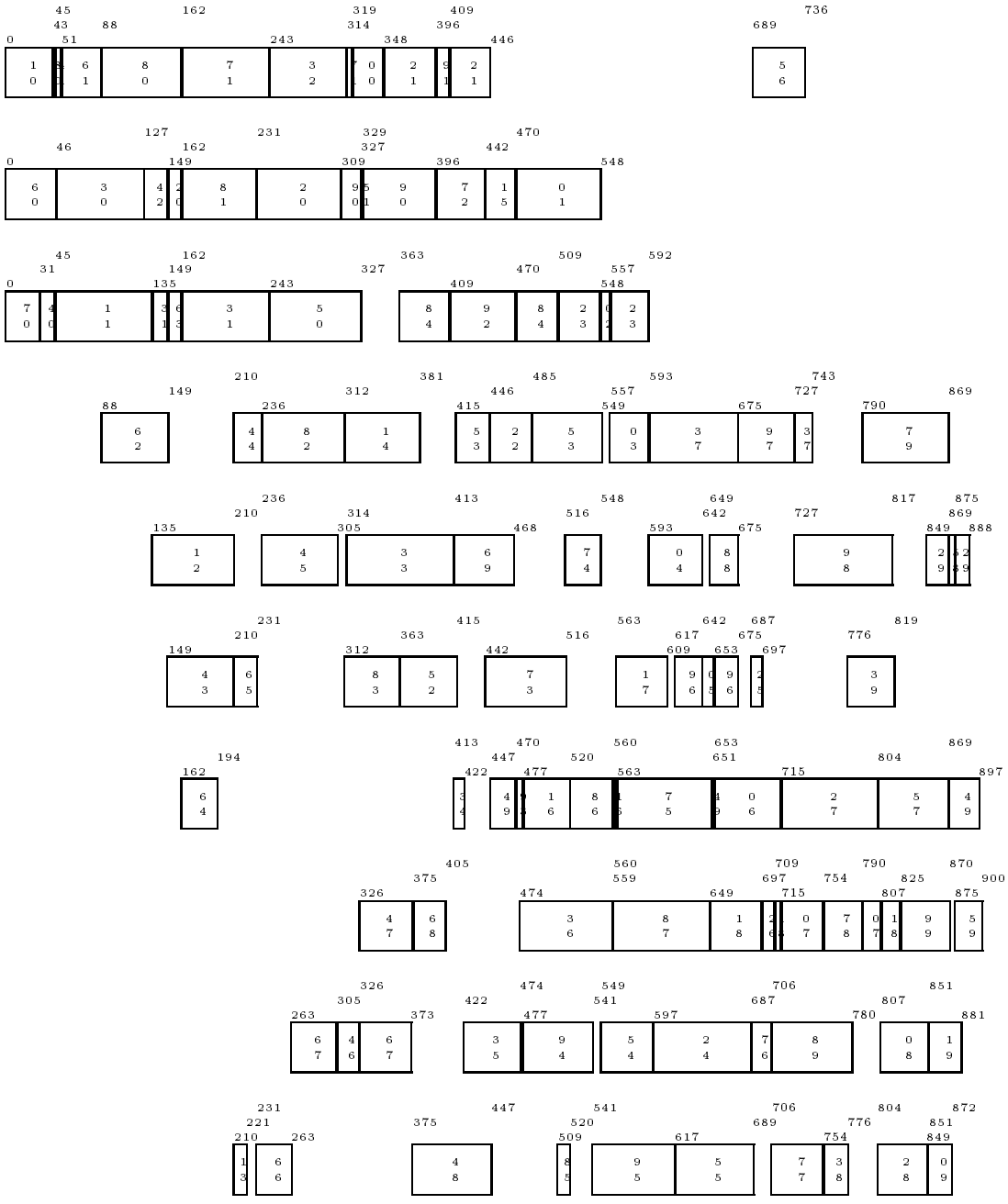

Figure 7.3: an optimal schedule of ft10

Inst	N-P Mak	P Mak	TF	TT	IT	PF	PT	TM
ft10	930	900	254801	97585.7	9	49817	19626.6	156908

Table 7.6: ft10 solved (AEC)

7.4 A Specific Resource Constraint Algorithm for Solving the PJSSP

We propose in this section an other algorithm to solve the PJSSP. It differs from the other algorithms since it does not rely on the generic constraint propagation algorithms described in chapter 4. Exceptionally, we present the search algorithm before describing the resource constraint.

7.4.1 Search Algorithm and Resource Constraint

Let us introduce an extra notation: for each resource $R_1, \cdots, R_m$, let $\mathcal{L}_i$ be a list of activities. At the beginning of the algorithm these lists are empty.

- Compute an upper bound M of the optimal makespan (for instance the sum of the durations of all activities).
- Constrain the end-time of the last activity of each job to be strictly lower than M.
- While there is an unscheduled resource,
  1. select R_i a resource
  2. While all activities are not in the $\mathcal{L}_i$
     (a) Select an activity $A \notin \mathcal{L}_i$, keep the other activities ($\notin \mathcal{L}_i$ as alternatives to be tried upon backtracking
     (b) Add A to the end of the list $\mathcal{L}_i$
  3. end While
- end While
- If a solution is found, set M to the current value of the makespan and iterate; otherwise the optimal makespan is M, exit.

Let us now describe the resource constraint propagation algorithm. Let us suppose that n activities require the resource R and that the the list $\mathcal{L}$ is:

$$\mathcal{L} = (A_1, A_2, \cdots, A_k)$$

In a seek of clarity, we suppose (without any loss of generality) that A_1 is the first activity in the list, that A_2 is the second etc.

1. The first step of the algorithm consists in propagating the two following constraints:
   - $end(A_1) < end(A_2) < \cdots < end(A_k)$
   - $\forall i \notin \{1, \cdots, k\}, end(A_k) < end(A_i)$
2. The second step of the algorithm consists in computing JPS and to state that the activities in the list cannot end before their end-times on JPS:

   $$\forall i \in \{1, \cdots, k\}, emin_i = e_{JPS}(A_i)$$

   Note that this adjustment is obvious since (1) the order of the due-dates of activities $(A_1, A_2, \cdots, A_k)$ will not change and (2) since on JPS the most urgent activities are scheduled as soon as possible.

Proof of the algorithm:

- We claim that if, for a given makespan M, the algorithm answers that there is a solution, there is indeed an acceptable schedule whose makespan is lower than or equal to M:

  Let us build the JPS of each machine and let us prove that the overall schedule is acceptable:

  - On each machine, the JPS can be built otherwise, a failure would have been deduced during the propagation.
  - The precedence constraints are satisfied since
    - each activity ends at its minimal end-time (recall that the constraint propagation ensure that the activities in the lists $\mathcal{L}$ cannot end before their end-times on JPS);
    - the propagation of temporal constraints is complete.

- We claim that if, for a given makespan M, the algorithm answers that there is no solution, there is indeed no acceptable schedule whose makespan is lower than or equal to M. We just have to prove that the algorithm is able to find an optimal solution: Let us consider the optimal schedule such that the schedule of any resource is a JPS (see proof of proposition 32). Since the algorithm only exhaustively change the orders of the end-times of activities, till it finds a solution, it happens that the orders correspond to the ones on JPS.

7.4.2 Experimental Results

The algorithm presented above seems interesting since it allows to "order" all activities of a resource before trying to schedule another resource. This heuristic performs very well for Job-Shop-Scheduling Problems (see for instance [Applegate 91]). However, the performances of this algorithm on the PJSSP were disappointing. We tried several different resource and activity selection heuristics but none of them seem to outperform another.

Chapter 8

Conclusion

We think we have highlighted in this report some interesting points:

- The prototype of interruptible activities seems to be an interesting possible extensions of ILOG SCHEDULE. It is more efficient in terms of "propagation power" than a previous prototype relying on a time-tabling mechanism but it is costly in term of cpu time.
- Approximation algorithms in a constraint-based environment may be very efficient. Moreover we think that, the framework of our approximation algorithm can be applied to many other combinatorial problems.
- We have also provided a characterization of the edge-finder algorithm which allows to understand "intuitively" the behavior of this algorithm. We think it may be an interesting basis for improving the propagation algorithms that we use. We expect that the theoretical results we have obtained in that respect, will lead to even better computational results fairly soon.

We try to carry on our work on these subjects and more generally on constraint-based scheduling a still very open and attractive domain.

Bibliography

[Aarts 94] E.H.L. Aarts and J.K. Lenstra. *Local Search in Combinatorial Optimization.*, Wiley, 1994.

[Adams 88] Joseph Adams, Egon Balas and Daniel Zawack. *The Shifting Bottleneck Procedure for Job-Shop Scheduling.* Management Science, 34(3):391-401, 1988.

[Applegate 91] David Applegate and William Cook. *A Computational Study of the Job-Shop Scheduling Problem.* Operations Research Society of America, Journal on Computing, Vol 3. No 2.

[Baptiste 94] Philippe Baptiste. *Constraint-Based Scheduling: Two Extensions.* MSc Thesis, University Of Strathclyde, Glasgow, 1994.

[Baptiste 95a] Philippe Baptiste and Claude Le Pape. *Disjunctive Constraints for Manufacturing Scheduling: Principles and Extensions.* Proceedings of the 3rd International Conference on Computer Integrated Manufacturing, Singapore, 1995.

[Baptiste 95b] Philippe Baptiste and Claude Le Pape. *A Theoretical and Experimental Comparison of Constraint Propagation Techniques for Disjunctive Scheduling.* Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montréal, Québec, 1995.

[Baptiste 95c] Philippe Baptiste, Claude Le Pape and Wim Nuijten. *Incorporating Efficient Operations Research Algorithms in Constraint-Based Scheduling.* Proceedings of the 1st International Joint Workshop on Artificial Intelligence and Operations Research, Timberline Lodge, Oregon, 1995.

[Baptiste 95d] Philippe Baptiste, Claude Le Pape and Wim Nuijten. *Constraint-Based Optimization and Approximation for Job Shop Scheduling.* Proceedings of the IJCAI '95 Workshop on Intelligent Manufacturing Systems, Montrál, Qébec, 1995.

[Baker 74] Kenneth R. Baker. *Introduction to Sequencing and Scheduling.* John Wiley and Sons, 1974.

[Barnes 91] J.W. Barnes and J.B. Chambers. *Solving the Job Shop Scheduling Problem using Tabu Search.* Technical Report, University of Texas, 1991.

[Beck 92] Howard Beck. *Constraint Monitoring in TOSCA.* Working Papers of the AAAI Spring Symposium on Practical Approaches to Planning and Scheduling, Stanford, California, 1992.

[Brucker 92] P. Brucker and B. Jurisch and B. Sievers. *A Branch & Bound Algorithm for the Job-Shop Scheduling Problem.* Technical Report, University of Osnabrück, 1992.

[Carlier 84] Jacques Carlier. *Problèmes d'Ordonnancement à Contraintes de Ressources : Algorithmes et Complexité.* Thèse de Doctorat d'Etat, Université Paris VI, 1984.

[Carlier 88] Jacques Carlier et Philippe Chrétienne. *Problèmes d'ordonnancement : Modélisation / Complexité / Algorithmes.* Masson, 1988.

[Carlier 89] Jacques Carlier and Eric Pinson. *An Algorithm for Solving the Job-Shop Problem.* Management Science, 35(2):164-176, 1989.

[Carlier 90] Jacques Carlier and Eric Pinson. *A Practical Use of Jackson's Preemptive Schedule for Solving the Job-Shop Problem.* Annals of Operations Research, 26:269-287, 1990.

[Carlier 94] Jacques Carlier and Eric Pinson. *Adjustment of Heads and Tails for the Job-Shop Problem.* European Journal of Operational Research, 78:146-161, 1994.

[Caseau 94a] Yves Caseau and Jean-François Puget. *Constraints on Order-Sorted Domains.* Proceedings of the ECCAI Workshop on Constraint Processing, ECAI, Amsterdam, The Netherlands, 1994.

[Caseau 94b] Yves Caseau and François Laburthe. *Improved CLP Scheduling with Task Intervals.* Proceedings of the Eleventh International Conference on Logic Programming, Santa Margherita Ligure, Italy, 1994.

[Collinot 91] Anne Collinot and Claude Le Pape. *Adapting the Behavior of a Job-Shop Scheduling System.* International Journal for Decision Support Systems, 7(3):341-353, 1991.

[Erschler 76] Jacques Erschler. *Analyse sous contraintes et aide à la décision pour certains problèmes d'ordonnancement.* Thèse de Doctorat d'Etat, Université Paul Sabatier, 1976.

[Erschler 91] Jacques Erschler, Pierre Lopez, Catherine Thuriot. *Raisonnement Temporel sous Contraintes de Ressource et Problèmes d'Ordonnancement.* Revue d'Intelligence Artificielle, 5(3):7-32, 1991.

[Garey 77] Michael R. Garey and David S. Johnson. *Two-processor scheduling with start-time and dead-lines.* SIAM J. Comput. 6.

[Garey 79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[Gondran 84] Michel Gondran and Michel Minoux. *Graphs and Algorithms.* John Wiley and Sons, 1984.

[Gonzalez 77] Gonzalez T. *Optimal Mean Finish Time Preemptive Schedules.* Report No. 220, Computer Science Department, Pennsylvania State University, University Park, PA.

[Lawler 73] E.L.Lawler *Optimal sequencing of a single machine subject to precedence constraints.* Management science 18.

[Lawler 78] E.L.Lawler and J.Labetoulle *Preemptive scheduling of unrelated parallel processors.* J. Assoc. Comput. Mach.

[Lawrence 84] S.Lawrence ASK WIM ??? *Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques* 1984

[Leconte 95] Michel Leconte. *A Bounds-based Reduction Scheme for Constraints of Difference* Technical Report, ILOG S.A., 1995. Submited to Annals of Artificial Intelligence and Mathematics, 1995.

[Le Pape 88] Claude Le Pape. *Des systèmes d'ordonnancement flexibles et opportunistes.* Thèse de l'Université Paris XI, 1988.

[Le Pape 94a] Claude Le Pape. *Implementation of Resource Constraints in* ILOG SCHEDULE*: A Library for the Development of Constraint-Based Scheduling Systems.* Intelligent Systems Engineering, 3(2):55-66, 1994.

[Le Pape 94b] Claude Le Pape. *Using a Constraint-Based Scheduling Library to Solve a Specific Scheduling Problem.* Proceedings of the AAAI-SIGMAN Workshop on AI Approaches to Modelling and Scheduling Manufacturing Processes, TAI, New Orleans, Louisiana, 1994 (to appear).

[Le Pape 94c] Claude Le Pape, Philippe Couronné, Didier Vergamini and Vincent Gosselin. *Time-versus-Capacity Compromises in Project Scheduling.* Proceedings of the Thirteenth Workshop of the UK Planning Special Interest Group, Strathclyde, United Kingdom, 1994.

[Lopez 91] Pierre Lopez. *Approche énergétique pour l'ordonnancement de tâches sous contraintes de temps et de ressources.* Thèse de l'Université Paul Sabatier, 1991.

[Morton 92] Thomas E. Morton and David W. Pentico. *Heuristic Scheduling Systems.* To appear.

[Muth 63] J.F. Muth and G.L.Thompson. *Industrial Scheduling.* Prentice-Hall. Englewood Cliffs, NJ. 1963.

[Nowicki 93] E. Nowicki and C. Smutnicki. *A Fast Taboo Search Algorithm for the Job-Shop Problem.* Preprinty nr. 8/93, Instytut Cybernetyki Technicznej, Politechnicki Wroclawskiej, 1993.

[Nuijten 93] W.P.M. Nuijten, E.H.L. Aarts, D.A.A. van Erp Taalman Kip, K.M. van Hee. *Job Shop Scheduling by Constraint Satisfaction.* Computing Science Note 93/39. Eindhoven University of Technology.

[Nuijten 94] W. P. M. Nuijten. *Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach.* PhD Thesis, Eindhoven University of Technology, 1994.

[Pinson 88] Eric Pinson. *Le problème de job-shop.* Thèse de l'Université Paris VI, 1988.

[Puget 91] Jean-François Puget et Patrick Albert. *PECOS : programmation par contraintes orientée objets.* Génie logiciel et systèmes experts, (23):100-105, 1991.

[Puget 92] Jean-François Puget. *Programmation par contraintes orientée objet.* Douzièmes journées internationales sur les systèmes experts et leurs applications, Avignon, France, 1992.

[Puget 94] Jean-François Puget. *A C++ Implementation of CLP.* Technical Report, ILOG S.A., 1994.

[Regin 94] Jean-Charles Regin. *A filtering algorithm for constraints of difference in CSPs.* Proceedings of AAAI, Seattle, Washington, 1994.

[Sadeh 91] Norman Sadeh. *Look Ahead Techniques for Micro-Opportunistic Job-Shop Scheduling.* PhD Thesis, Carnegie-Mellon University, 1991.

[Taillard 89] E. Taillard. *Parallel Taboo Search Technique for the Job Shop Scheduling Problem.* Internal Report, Département de Mathématique, Ecole Polytechnique Fédérale de Lausanne, 1989.

[Vaessens 94] R. J. M. Vaessens, E. H. L. Aaarts and J. K. Lenstra. *Job-Shop Scheduling by Local Search.* COSOR Memorandum 94-05, Eindhoven University of Technology, 1994.

[Van Hentenryck 89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.