MSc Thesis

Constraint-Based Scheduling: Two Extensions

MSc of Computer Integrated Manufacturing,
Department of Design, Manufacture and Engineering Management
University of Strathclyde
October 1994

Supervisors:
Professor Carrie (D.M.E.M. - Strathclyde University),
Professor Christie (D.M.E.M. - Strathclyde University),
Professor Portmann (Ecole des Mines de Nancy),
Professor Prosser (Department of Computer Science - Strathclyde University)

Philippe Baptiste
ILOG S.A.
2 Avenue Galliéni
94253 Gentilly Cedex, France

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. 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.

The goal of the work presented here is to extend ILOG SCHEDULE in two directions:

- **The creation of interruptible activities.** 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. In this report, we discuss the feasibility of a direct representation of interruptible activities.
- **The improvement of the constraint propagation process for unary resources.** ILOG SOLVER is flexible enough to enable adjustment of the amount of propagation performed in evaluating the consequences of scheduling decisions. When the amount of propagation is reduced, each decision is made and/or retracted more rapidly, but many decision making and retracting steps (backtracking) are necessary to find a satisfactory solution. When the amount of propagation is extended, the search space is more drastically pruned. In this report, we discuss and compare several algorithms performing extended propagation and show on various examples that extended propagation is often worth it.

Acknowledgments

This MSc thesis is the conclusion of a six-months training period at ILOG. This work has been tremendously 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 of constraint programming. I want to thank Jean-Francois Puget, Younes Alaoui, Michel Leconte, and the all SOLVER and SCHEDULE team. Thank you very much to my supervisors: Professor Carrie and Professor Christie of the DMEM (Department of Design, Manufacture and Engineering Management, Strathclyde University) as well as Patrick Prosser of the Department of Computer Science. A special thank to Mlle Marie-Claude Portmann, the head-master of the Computer Science and Industrial Engineering Department at Ecole des Mines de Nancy, who made this training period possible.

Contents

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. 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. 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.

Some of the reasons why good scheduling systems are very important for a company are the following:

1. Scheduling problems occur at all levels of the company and over all time horizons.
2. With the help of good scheduling solutions, the company's performances increase.
3. Fast decision-making can be achieved. For instance, if a production line problem occurs, a fast re-scheduling decision must be made. The use of a scheduling system enables a "good" decision to be made quickly. Similarly, the production planning may be recomputed every day to take into account modifications of production plan orders and changes in resource capacities.
4. A scheduling system fits very well in the context of Total Quality Management (TQM). Delays are shortened and stocks reduced. These features are fundamental according to the TQM philosophy.

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 (cf. [Le Pape 94b] 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 goal of the work presented here is to extend ILOG SCHEDULE in two directions:

- **The creation of interruptible activities.** 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. However, this representation is costly in terms of CPU time and memory space. In this report, we discuss the feasibility of a direct representation of interruptible activities, which is much less space-consuming than decomposing interruptible activities in a potentially high number of non-interruptible sub-activities.
- **The improvement of the constraint propagation process for unary resources.** ILOG SOLVER is flexible enough to enable adjustment of the amount of propagation performed in evaluating the consequences of scheduling decisions. When the amount of propagation is reduced, each decision is made and/or retracted more rapidly, but many decision making and retracting steps (backtracking) are necessary to find a satisfactory solution. When the amount of propagation is extended, the search space is more drastically pruned. Each decision requires more CPU time, but the number of problem-solving steps decreases. Results of experiments with flexible constraint propagation systems show that the amount of constraint propagation that enables a problem-solver to be the most efficient varies with the problem-solver, with the application (e.g., in job-shop scheduling, the appropriate amount of propagation varies from one shop to another) and with the problem-solving context (e.g., in case of urgency, propagation can be restricted to constraints relating to imminent manufacturing operations) [Collinot 91] [Van Hentenryck 89]. In this report, we discuss ways to extend ILOG SCHEDULE so that the user can require extended propagation of the resource constraints. Our work focuses on unary resources (i.e., resources of capacity 1), although some ideas could possibly generalise to discrete resources.

The remainder of this report is divided in 3 chapters. In chapter 2, we briefly present ILOG, and the two C++ libraries of interest to us, ILOG SOLVER and ILOG SCHEDULE. In chapter 3, we present a prototype implementation of interruptible activities. Chapter 4 is devoted to the extended propagation for unary resources and to the experimental results we have obtained, in particular on job shop scheduling benchmarks which are widely used in the operations research and constraint programming communities.

2

General Presentation

2.1 Presentation of the Company

ILOG was created in 1987 to industrialise the expertise of INRIA, Europe's largest computer research centre 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 centre, the department in which the work corresponding to this report has been done, operates with some fourty 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.

For the good understanding of the following chapters, the concepts described in the following pages have to be fairly well taken in.

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.

An overview of ILOG SOLVER follows.

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:

```
CtEqAdd(x, y, z);                    // relational form
CtEq(x, CtAdd(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 `CtAllNeq` which states that all the variables in a given array are different, or `CtArraySum` 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`.

```
CTGOAL0(f) {
  CtOr(CtAnd(f1(), f2()),
       CtAnd(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.

ILOG SCHEDULE offers an object-oriented scheduling model that predefines the following 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 utilisation constraints which specify how activities can use and share resources.

In this section, we will describe 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** which includes both **resources** with limited capacity and **activities**.

2.4.1 The Schedule

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

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

2.4.2 Resources

ILOG SCHEDULE provides four different types of resources:

- `CtDiscreteResource`: A `CtDiscreteResource` 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. This requirement is represented by resource constraints, and propagation of these constraints entails an update of the earliest and latest start and end times of activities.

- `CtUnaryResource`: A `CtUnaryResource` represents a resource whose capacity is one. There are two methods to take into account the constraints concerning the requirement or the provision of a unary resource. The first method specialises (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 available. In contrast, the second method deals only with requiring (or providing) activities: it consists of updating the earliest and latest start and end times of activities to ensure that the time intervals over which two activities require (or provide) the same unary resource cannot overlap.
- `CtDiscreteEnergy`: A `CtDiscreteEnergy` is similar to a `CtDiscreteResource`, 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.
- `CtStateResource`: A `CtStateResource` 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 `CtIntervalActivity` class. An activity that executes without interruption from its start time to its end time, and uses the same resources from the begining to the end of its execution, can be represented by a single instance of the `CtIntervalActivity` 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 Some Fundamental Technical Points

We will only discuss here two very important points without which the good-understanding of the following pages would not be possible.

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 memorises 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 memorises 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.

These discrete arrays and sequential tables are used in chapter 3 to implement resource requirement constraints for interruptible activities.

Unary Resources

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

- The first method specialises (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 `CtUnaryResource` are arrays and tables of Boolean variables (`CtBoolVar`).
- 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] No time-table needs to be created if this method is used.

The differences between the two representations are highlighted below:

- The disjunctive representation deals 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 two ways of propagating the constraints, knowing that the disjunctive constraint propagates more than the time-table constraint. In chapter 4, we propose extensions of the disjunctive constraint that result in even more constraint propagation being performed. 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 extended formulation presented in chapter 3.

[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`.

3

Implementing Interruptible Activities in ILOG SCHEDULE

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 such activities has to implement such a model. In this chapter, we will discuss the feasibility of a direct representation and describe a prototype implementation of interruptible activities in ILOG SCHEDULE.

In section 3.1, we more thoroughly explain why it would be beneficial to make interruptible activities available in ILOG SCHEDULE and enumerate the functionalities that would be worth offering. Section 3.2 describes the prototype implementation developed at ILOG. Section 3.3 shows how interruptible activities can be used on two examples: (1) to implement Mac Naughton's algorithm [Carlier 88]; and (2) to implement a scheduling algorithm based on the Preemptive Shortest Processing Time rule (PSPT) [Morton 92].

3.1 Functionalities

It is interesting to deal with interruptible activities for two main reasons. On the one hand, they would be very useful for modelling 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 favour 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 example, the problem of sequencing n independent tasks subjected to release dates and due-dates on a unique resource is NP-hard when activities are not interruptible. This means that it is conjectured that no algorithm could optimally solve the problem in an amount of CPU time bounded by a polynomial function of the number of activities. 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. Appendix A summarises the most well-known complexity results concerning preemptive scheduling. A summary of Carlier and Pinson's use of preemptive solutions appears in the next chapter.

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 protoype, 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 and 2 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 would be both more convenient and more efficient.

3.2 Implementation Principle

Our implementation of resource constraints for interruptible activities is based on the general principles used for non-interruptible activities. We describe these principles first, for the non-interruptible activities. Then we explain how the implementation has been extended to interruptible activities. Only the time-table mechanisms have been considered. Also we have purposedly ignored the case of energetic resources, which happens to be quite complex, even in the non-interruptible case.

3.2.1 Resource Constraints in ILOG SCHEDULE

In the case of non-interruptible activities, the basic scheme of propagation is the following one:

- Time-tables are questioned to compute the intervals [StartMin, EndMin) and [StartMax, EndMax) over which the activity can be executed as soon as possible and as late as possible. It is checked that on these intervals, the required "amount of resource" is available. More precisely, the effect of propagation is to increase (resp. decrease) StartMin and EndMin (resp. StartMax and EndMax) until the [StartMin, EndMin) and [StartMax, EndMax) intervals satisfy the constraint.
- Over the interval [StartMax, EndMin) (if $StartMax \leq EndMin$), it is sure that the activity will be executed. Therefore, the corresponding amount of resource over this period is no longer available for other activities.
- In the same way, if the user sets the start (or end) time of the activity, the amount of resource used by the activity is taken into account in the time-table.

This constraint propagation mechanism guarantees a systematic update of activity earliest and latest start and end times when decisions are made: for each activity `A`, it is guaranteed that the earliest and latest start and end times of `A` are consistent with the temporal constraints and with the resource requirements of "scheduled" activities (i.e., activities for which the start and end times have been fixed).[1] This, in turn, guarantees that any attempt to fix the start and end times of all activities will succeed only if the chosen start and end times satisfy all the constraints of the considered problem.

3.2.2 Extension to Interruptible Activities

The management of interruptible activities is based upon the same principles as those described in the last paragraph:

[1] Of course, this does not mean that the earliest and latest start and end times of all activities (scheduled and unscheduled) are globally consistent. In most cases, it is necessary to explore the search space to determine a globally consistent solution.

- The StartMin of each activity is updated so that the required resources are available at StartMin. The EndMin is updated so that between StartMin and EndMin there are at least DurationMin time points at which the resources are available. StartMax and EndMax are computed similarly. Note that if an activity requires several resources, all these resources must be simultaneously taken into account to guarantee that there truly are DurationMin time points between StartMin and EndMin at which the activity can execute.
- If, in the interval [StartMin, EndMax), there are exactly DurationMin time points at which the resources are available, then it is sure that the activity is processed exactly at these instants. Time-tables are updated accordingly. (Of course, a failure occurs if there are not enough points at which the activity can be executed.)
- In the same way, if the user constrains the activity to execute at a given time point (or over a given time interval), the amount of resources used by the activity is taken into account in the time-tables.

This propagation mechanism guarantees as well as the previous one a systematic update of activity earliest and latest start and end times when decisions are made.

3.3 Examples of Use

In this section, we present two small examples of use of our implementation of interruptible activities.

- The first example is an implementation of Mac Naughton's algorithm [Carlier 88]. Mac Naughton's algorithm determines a schedule of minimal makespan for n independent activities (not subjected to release and due times) executing on a unique discrete resource of capacity m. Our implementation is presented in section 3.3.1.
- The second example implements a scheduling algorithm based on the Preemptive Shortest Processing Time rule (PSPT) [Morton 92]. This rule determines a schedule of minimal average completion time for n independent activities, subjected to release time constraints, executing on a unique unary resource. Our implementation is presented in section 3.3.2.

3.3.1 Mac Naughton's Algorithm

The problem is is known as the C_{max} preemptive scheduling problem with no due-dates and with the same release-dates for all activities on a discrete resource. The goal is actually to find a schedule which minimises the makespan.

The value of the minimal makespan is:

$$minMakespan = \max(\max_i(p_i), (\Sigma_i(p_i) + m - 1)/m) \tag{3.1}$$

where p_i denotes the processing time of activity i and m the capacity of the discrete resource. (For a demonstration, look in Appendix A. 6.2.1)

Mac Naughton's algorithm consists in queueing all the activities one after an other and to cut slices of length minMakespan. It is then possible to superpose these slices on the different resources.

This algorithm is very simple to code with the help of ILOG SCHEDULE:

- Definition of the schedule, the resources and the activities:

```
CtInterruptibleActivity** DefineProblem(CtInt m,
                                        CtInt n,
                                        CtInt* durations,
                                        CtInt minMakespan) {
  // Initialization of the CtSchedule object.
  CtSchedule* schedule = new (CtHeap()) CtSchedule(0, minMakespan);
  // Creation of a discrete resource of capacity m.
  CtDiscreteResource* resource =
    new (CtHeap()) CtDiscreteResource(schedule,
                                      CtRequiredResource,
                                      m);
  // Creation of the n activities and resource requirements.
  CtInterruptibleActivity** activities =
    new (CtHeap()) CtInterruptibleActivity*[n];
  for (CtInt i = 0; i < NumberOfActivities; i++) {
    activities[i] =
      new (CtHeap()) CtInterruptibleActivity(schedule,
                                             durations[i]);
    activities[i]->requires(resource, 1);
  }
  return activities;
}
```

- Search for solution:

```
void SolveProblem(CtInt n,
                  CtInt minMakespan,
                  CtInterruptibleActivity** activities) {
  CtInt start = 0;
  for (CtInt i = 0; i < n; i++) {
    CtInt end = start + activities[i]->getDurationMin();
    if (end <= minMakespan) {
      activities[i]->addRequiredInterval(start, end);
      start = end;
    }
    else {
      activities[i]->addRequiredInterval(start, minMakespan);
      activities[i]->addRequiredInterval(0, end - minMakespan);
      start = end - minMakespan;
    }
  }
}
```

If we use the example given in [Carlier 88] (five activities of durations 10, 8, 4, 14, 1, with a discrete resource of capacity 3), we get the following result:

```
A1 executed over [0 10]
A2 executed over [0 4] [10 14]
A3 executed over [4 8]
A4 executed over [0 14]
A5 executed over [8 9]
```

Note that for such a simple problem the constraint-based method is sub-optimal. Indeed, constraints are propagated after each decision, which is useless for merely solving MacNaughton's problem. Also, the prototype implementation has not been highly optimised. This currently leads to a CPU time of about 0.5 seconds for problems

including 50 activities. ILOG developers expect to drastically improve on this in the near future.

3.3.2 The PSPT Algorithm

The Preemptive Shortest Processing Time rule (PSPT) [Morton 92] determines a schedule of minimal average completion time for n independent activities subjected to release time constraints and executing on a unique unary resource.

Morton and Pentico [Morton 92] describe the algorithm as follows:

1. When an activity finishes, start the currently available activity with the smallest remaining processing time (residual duration).
2. If an activity i becomes available while activity j is in process, stop activity j and start activity i if the residual duration of i is strictly smaller than the residual duration of j; otherwise continue activity j.

The following algorithm is equivalent to the algorithm above:

1. Select the activity with the smallest earliest end time.
2. Set the end time of the chosen activity to its earliest end time. (This will automatically set the start time of the activity to its earliest start time, as well as determine all intervals over which the activity executes).
3. Iterate until all activities are scheduled.

Proof: We prove the equivalence of the two algorithms by induction on the sum of the durations of the activities. The result is obviously true if the sum of the durations is 1. Suppose it is true when the sum of the durations is $n-1$. Suppose the sum of the durations is n. We first show that the activity A that executes at the first time point t_0 is the same according to both algorithms. Indeed, this activity must be the one with the smallest duration among those that can start at the first time point.[2] This time point being reserved, we create a new problem by updating durations and release times as follows:

- The duration of A is decreased of one unit. (A disappears if A was already of duration 1.)
- The release dates of the activities that could have executed at t_0 are increased of one unit.

Then the induction hypothesis applies to the new problem and the schedules generated by the two algorithms are equivalent.

With the help of the ILOG SCHEDULE library, the algorithm is written as follows:

- The first step is to select the activity to be scheduled. The following code defines a function `ChooseActivity` which, given a `CtSchedule` object, returns the activity whose earliest end time is minimal among those for which the end time is not fixed.

[2] If there are several activities with the same release date and duration, they are fully interchangeable, so the result is true up to a renaming of the identical activities.

```
CtActivitySelector1(ChooseActivity,
                    !activity->getEndVariable()->isBound(),
                    activity->getEndMin(),
                    CtInterruptibleActivity)
```

- The following function computes the optimal solution:

```
void SolveProblem(CtSchedule* schedule) {
  CtInterruptibleActivity*
    chosenActivity = ChooseActivity(schedule);
  while (chosenActivity) {
    chosenActivity->setEndTime(chosenActivity->getEndMin());
    chosenActivity = ChooseActivity(schedule);
  }
}
```

As an example, consider six activities with respective durations 10, 8, 3, 2, 1, 8, and release times 0, 5, 6, 8, 5, 1. The following result is obtained:

```
A1 executed over [0 1] [23 32]
A2 executed over [15 23]
A3 executed over [6 9]
A4 executed over [9 11]
A5 executed over [5 6]
A6 executed over [1 5] [11 15]
```

Let us note that the same algorithm, applied to a discrete resource of capacity c strictly greater than 1, does not necessarily provide the optimal solution. Indeed, a counter-example is provided in Appendix A.

4

Extended Propagation of Disjunctive Constraints

As mentioned in the general presentation of ILOG SCHEDULE, two different propagation methods are currently available for unary resources. Let's briefly recall them:

- The first method specialises (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 `CtUnaryResource` are arrays and tables of Boolean variables (`CtBoolVar`).
- 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. For instance, if a resource is required (or provided) by two activities throughout two time intervals `[ti1 ti2)` and `[tj1 tj2)`, then 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.

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.

In this chapter, we propose an extension of the second method. The extended method results in even more precise time-bounds. To design this new method, we first reviewed the works of a number of researchers who designed "propagation algorithms" that deduce more precise information than the disjunctive constraint of ILOG SCHEDULE version 1.0. This includes both work performed from a traditional operations research perspective and work on specific constraint programming techniques for scheduling. Section 4.1 describes these techniques and the theoretical results on which they are built. Section 4.2 presents the techniques we implemented as a prototype for a future version of ILOG SCHEDULE. Finally, section 4.3 provides computational results.

4.1 Bibliographical Study

In the literature, there are roughly two types of methods that provide more precise time bounds than the disjunctive constraint of ILOG SCHEDULE. The first type of method, described in section 4.1.1, 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 as, amongst other impressive results, their algorithm was the first to exactly solve a benchmark 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].

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. 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].

This form of propagation is discussed in section 4.1.2.

4.1.1 Automatic Sequencing of Activities

4.1.1.1. 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). A schedule has to verify the following equation:

$$\max_{i \in K}(d_i) - \min_{i \in K}(r_i) \geq \sum_{i \in K}(p_i) \tag{4.1}$$

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

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.

4.1.1.2. 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.

$$\forall k \in K, \left[\max_{i \in K-\{k\}}(d_i) - r_k < \sum_{i \in K}(p_i) \right] \Rightarrow [k \textit{ is not the input of } K] \tag{4.2}$$

$$\forall k \in K, \left[d_k - \min_{i \in K-\{k\}}(r_i) < \sum_{i \in K}(p_i) \right] \Rightarrow [k \textit{ is not the output of } K] \tag{4.3}$$

$$\forall k \in K, \left[\max_{i \in K-\{k\}}(d_i) - \min_{i \in K-\{k\}}(r_i) < \sum_{i \in K}(p_i) \right] \Rightarrow [k \textit{ is not a middle of } K] \tag{4.4}$$

These propositions are based upon the same principle as the "fundamental" one. 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:

$$\forall k \in K, \begin{cases} \max_{i \in K-\{k\}}(d_i) - \min_{i \in K-\{k\}}(r_i) < \sum_{i \in K}(p_i) \\ d_k - \min_{i \in K-\{k\}}(r_i) < \sum_{i \in K}(p_i) \end{cases} \Rightarrow [k \textit{ is the input}] \tag{4.5}$$

$$\forall k \in K, \begin{cases} \max_{i \in K-\{k\}}(d_i) - r_k < \sum_{i \in K}(p_i) \\ \max_{i \in K-\{k\}}(d_i) - \min_{i \in K-\{k\}}(r_i) < \sum_{i \in K}(p_i) \end{cases} \Rightarrow [k \textit{ is the output}] \tag{4.6}$$

$$\forall k \in K, \begin{cases} \max_{i \in K-\{k\}}(d_i) - r_k < \sum_{i \in K}(p_i) \\ d_k - \min_{i \in K-\{k\}}(r_i) < \sum_{i \in K}(p_i) \end{cases} \Rightarrow [k \textit{ is a middle}] \tag{4.7}$$

These propositions are obvious. They set the fact (for the first one for instance) 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 ouptput) 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 summarises the main results.

4.1.1.3. Jackson's Preemptive Schedule

Jackson's preemptive schedule is the preemptive schedule obtained by applying the rule referred to as "Preemptive Earliest Due-Date rule" in [Morton 92]. This schedule minimises the maximal lateness (and maximal tardiness) of n interruptible activities, subjected to release and due time 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 date.

2. If an activity i becomes available while activity j is in process, stop activity j and start activity i if the due date of i is strictly smaller than the due date 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:

$$r_i + p_i + \sum_{A_k \in K / d_k \leq d_j} p_k^* > d_j \tag{4.8}$$

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

$$\forall A_k \in K / d_k \leq d_j, A_k \textit{ is scheduled before } A_i \tag{4.9}$$

$$r_i \geq \max_{A_k \in K / d_k \leq d_j} C_{A_k}^{JPS} \tag{4.10}$$

[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.

where $C_{A_k}^{JPS}$ is the completion time of the activity A_k on Jackson Preemptive schedule.

For proof of this algorithm, see [Pinson 88] [Carlier 90].

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...

4.1.1.4. Computational Experiments: Carlier and Pinson

In [Carlier 89] and [Carlier 90] the previous deductive propositions have been associated with a branch and bound method to solve the **Job Shop Scheduling Problem** which consists in scheduling n jobs on m machines with a minimal makespan (for a more precise definition see 4.3.2). The authors solved optimally the **MT10**, a specially hard instance of the Job Shop Scheduling Problem introduced in [Muth 63]. They proved the optimality of their solution with a small number of branching nodes (4300) and in less than 80 minutes on a PRIME 2655.

4.1.1.5. Computational Experiments: Applegate and Cook

Applegate and Cook [Applegate 91] have implemented a combinatorial branch and bound algorithm that incorporates the key ideas of Carlier and Pinson associated with some other techniques. Computational results are provided for several techniques and combinations of techniques. The optimisation procedure on which they eventually focus is a combination of three algorithms: (1) a variant of the bottleneck shifting procedure of Adams, Balas and Zawack [Adams 88]; followed by (2) a "shuffle" algorithm which mixes ideas from [Adams 88] and [Carlier 89] (i.e., fixing the schedule of a few machines and using a variant of Carlier and Pinson's edge-finding techniques to schedule the other machines); followed by (3) a variant of Carlier and Pinson's algorithm to guarantee optimality of the final solution. This optimisation procedure solves the MT10 problem in under 7 minutes of computation on a Sun Sparcstation 1.

As far as constraint propagation is concerned, the optimisation procedure finally adopted by Applegate and Cook "does not propagate more" than Carlier and Pinson's procedure (the one described in [Carlier 90]). When Applegate and Cook obtain good results in comparison to Carlier and Pinson, it is due to the finding of a heuristic combination that happens to work well on the tested problems.

4.1.1.6. Task Intervals

In [Caseau 94b], Caseau and Laburthe present a constraint-based technique that is shown to be very efficient as far as the computational speed is concerned. The special instance of the JSSP known as the MT10 is solved optimally in less than 1300 seconds. This brings their constraint programming approach in the same range of efficiency as the best operations research algorithms (those presented above). To achieve such results, Caseau and Laburthe have developed two sets of rules based on the concept of "task intervals":

- **edge-finding within a task interval:** The *edge-finding* technique is very powerful but, as we have said, it cannot be naively applied to all the subsets S of activities requiring the same resource (since for n activities, this would lead to 2^n sets). To avoid this, [Caseau 94b] associates to all couples of activities (A, B) scheduled on the same machine the set of activities (clique) $K_{(A,B)}$ which will surely be scheduled between the earliest start time of A (r_A) and the latest end time of B (d_B):

$$(A, B) \rightarrow K_{(A,B)} = \{\alpha / r_A \leq r_\alpha \; and \; d_\alpha \leq d_A\} \tag{4.11}$$

  $K_{(A,B)}$ is called a task interval and is considered of interest only when A and B belong to $K_{(A,B)}$. Deductive edge-finding propositions are applied to task intervals: for each activity C in $K_{(A,B)}$, the rules determine whether C is necessarily the input (or the output) of $K_{(A,B)}$.

- **edge-finding between a task interval and another task:** this second set of rules is used to determine whether a task α can be performed before a task interval $K_{(A,B)}$ to which it does not belong. This is done by testing if α may be before, after or inside $K_{(A,B)}$:

$$\left[d_B - r_\alpha - p_\alpha - \sum_{i \in K_{(A,B)}} p_i < 0 \right] \Rightarrow \left[\alpha \; not \; before \; K_{(A,B)} \right] \tag{4.12}$$

$$\left[d_B - r_A - p_\alpha - \sum_{i \in K_{(A,B)}} p_i < 0 \right] \Rightarrow \left[\alpha \; not \; inside \; K_{(A,B)} \right] \tag{4.13}$$

$$\left[d_\alpha - r_A - p_\alpha - \sum_{i \in K_{(A,B)}} p_i < 0 \right] \Rightarrow [\alpha \; not \; after \; K_{(A,B)}] \tag{4.14}$$

According to these propositions, release-dates and due-dates can be updated. For example, if α may not be before $K_{(A,B)}$ but may be inside, then α is at least after one of the activities of $K_{(A,B)}$. Therefore, the release date of α is updated according to the following equation:

$$r_\alpha = \max(r_\alpha, \min_{i \in K_{(A,B)}} (r_i + p_i)). \tag{4.15}$$

Note that computing the expression

$$\min_{i \in K_{(A,B)}} (r_i + p_i) \tag{4.16}$$

can be costly. To avoid unnecessary computation, Caseau and Laburthe check whether the activity α can start after or before the minimal completion time of the activity A. If

$$r_\alpha < r_A + p_A \tag{4.17}$$

is true, then the edge-finding test is made. Otherwise, the edge-finding test is not made because it is sure that the deduction will not be efficient since:

$$[r_\alpha \geq r_A + p_A] \Rightarrow [r_\alpha \geq \min_{i \in I(C)} r_i + p_i]. \tag{4.18}$$

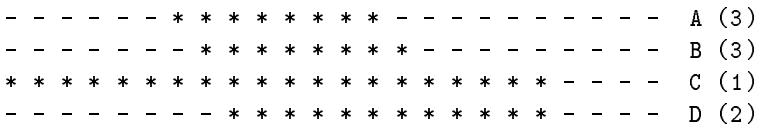
A few remarks can be made about these sets of rules. First, there are at most n^2 task intervals to consider for a resource on which n activities are scheduled. This means that in the worst case, a computational time in $O(n^3)$ is necessary to apply the rules to all the task intervals and all the activities of a given resource. In comparison, the edge-finding procedure of Carlier and Pinson runs in $O(n^2)$. However, the implementation of task intervals is simpler (e.g., no need to compute Jackson's preemptive schedule), so it is difficult to determine for which values of n Carlier and Pinson's procedure will be the most efficient. Also, Caseau and Laburthe suggest that it is possible to reduce the number of task intervals so as to avoid redundant computations. Indeed, the same clique may be associated to two different couples of activities. It is then possible to use a mechanism to "order" the cliques. However, according to [Caseau 94b], the maintenance of a non-redundant set of cliques is too computationaly expensive to gain any benefit.

Another important remark is that the second set of rules can lead constraint propagation to different results depending on the order in which constraints are propagated. 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. The second set of rules presented above "breaks" the "fixpoint semantics" of constraint propagation. This is illustrated below on a simple example.

Let A, B, C, D be four activities to schedule on a unary resource with respect to the following release and due-dates:

A ONE-MACHINE PROBLEM			
Activity	duration	release-date	due-date
A	3	6	14
B	3	7	15
C	1	0	20
D	2	8	20

The following figure displays the intervals $[r_i, d_i)$ of each of the four activities.



Let us try now to update the release and due dates according to the previous rules.

1. $K_{(A,A)} = \{A\}$. No deduction.
2. $K_{(A,B)} = \{A, B\}$. Let us try to deduce something with D:

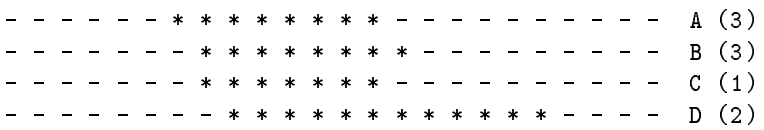
$$[15 - 8 - 2 - (3 + 3) = -1 < 0] \Rightarrow [D \textit{ not before } K_{(A,B)}]$$

$$[15 - 6 - 2 - (3 + 3) = 1 \geq 0] \Rightarrow [D \textit{ may be inside } K_{(A,B)}]$$

$$[20 - 6 - 2 - (3 + 3) = 6 \geq 0] \Rightarrow [D \textit{ may be after } K_{(A,B)}]$$

   This means that D may not be before $K_{(A,B)} = \{A, B\}$ but may be inside $K_{(A,B)}$. Consequently, r_D is updated to $\min(r_A + p_A, r_B + p_B) = 9$.
3. $K_{(A,C)} = \{A, B, D\}$. No deduction.
4. $K_{(A,D)} = \{A, B, D\}$. No deduction.
5. $K_{(B,A)} = \{\}$. No deduction.
6. $K_{(B,B)} = \{B\}$. No deduction.
7. $K_{(B,C)} = \{B, D\}$. No deduction.
8. ...

Let us now restart the example after changing the due-dates and the release-dates of the activity C: $r_C = 7$ and $d_C = 14$. These new values reduce the domain of the possible dates of execution of C (i.e., the problem is more constrained); therefore, if the system respects fixpoint semantics, the propagation should deduce at least as much new information as before.



1. $K_{(A,A)} = \{A\}$. No deduction.
2. $K_{(A,B)} = \{A, B, C\}$. Let us try to deduce something with D:

$$[15 - 8 - 2 - (3 + 3 + 1) = -2 < 0] \Rightarrow [D \textit{ not before } K_{(A,B)}]$$

$$[15 - 6 - 2 - (3 + 3 + 1) = 0 \geq 0] \Rightarrow [D \textit{ may be inside } K_{(A,B)}]$$

$$[20 - 6 - 2 - (3 + 3 + 1) = 5 \geq 0] \Rightarrow [D \textit{ may be after } K_{(A,B)}]$$

   This means that D may not be before $K_{(A,B)} = \{A, B, C\}$ but may be inside. Consequently r_D is updated to $\min(r_A + p_A, r_B + p_B, r_C + p_C) = \min(6 + 3, 7 + 3, 7 + 1) = 8$. The deduction $r_D = 9$ has not been done!
3. $K_{(A,C)} = \{A, C\}$. No deduction.

4. $K_{(A,D)} = \{A, B, C, D\}$. No deduction.
5. $K_{(B,A)} = \{C\}$. No deduction.
6. $K_{(B,B)} = \{B, C\}$. No deduction.
7. $K_{(B,C)} = \{C\}$. No deduction.
8. ...

The counter-example shows that the second set of rules does not respect fixpoint semantics. This is not a real problem when developing a specific algorithm, but may be annoying in the context of a constraint programming library like ILOG SCHEDULE. Indeed, this means that a user of the library could face different execution behaviors depending on the order in which constraints are posted. Such variability would make it less easy to use ILOG SCHEDULE as a tool for the development of complex scheduling applications.

4.1.1.7. The algorithm of Nuijten, Aarts, van Erp Taalman Kip and van Hee

In [Nuijten 93], a variant of the algorithm of Carlier and Pinson is developed. This new algorithm runs in $O(n^2)$ (as Carlier and Pinson algorihtm) but it is simpler in the sense that it does not require the computation of Jackson's preemptive schedule.[2]

4.1.2 Energy-Based 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, from now on, we will work on the one-machine problem.

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.

$$W_A^{[t_1,t_2)} = \max\left(0, \min(p_A, t_2 - t_1, r_A + p_A - t_1, t_2 - d_A + p_A)\right) \tag{4.19}$$

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.

$$\left[d_B - r_A < \sum_{\alpha \notin \{A,B\}} W_\alpha^{[r_A,d_B)} + p_A + p_B \right] \Rightarrow \; B \; before \; A \tag{4.20}$$

[2] Since our writing of this review, we found an article by Carlier and Pinson [Carlier 94], in which they describe an algorithm running in $O(n * \log n)$. We have not studied this algorithm in detail yet.

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 ealiest 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:

$$[x - r_A < TotalW] \Rightarrow \left[r_A = r_A + \sum_{\alpha \neq A} W_\alpha^{[r_A, x)} \right] \tag{4.21}$$

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).

These rules are quite efficient. In a simple experiment, we combined the first rule with the edge-finding technique of Carlier and Pinson restricted to the overall set of yet unordered operations. With the resulting algorithm, MT10 was solved in about eight hours of CPU time, on a HP-700 workstation. This is significantly more than [Carlier 90], [Applegate 91] and [Caseau 94b]; yet the result striked us given that our algorithm was extremely simple.

While using this algorithm, we noticed that the number of backtracks required to prove the optimality of the solution varied with the order in which constraints were posted. This suggests that the energetic rule embedded in the algorithm violates fixpoint semantics. Yet we have been unable to put together a simple counter-example showing that fixpoint semantics are violated.

Energetic reasoning is also used in the TOSCA system of Beck [Beck 92] and in the energetic resources of ILOG SCHEDULE (cf. section 2.4.2). Beck uses a special structure called Habograph to compute the **required energy consumption** and the **maximal energy available** over each time interval $[s\ e)$ up to a given discretization of time. When the required energy consumption exceeds the maximal energy available, a failure occurs. When the required energy consumption is close to the maximal energy available, a constraint threat is indicated. Potentially, the habograph could be used to determine that an activity, say A, cannot fully execute during a given time interval $[s\ e)$, in a way similar to applying rule 4.21 above (but with predetermined time intervals, rather than intervals depending on the earliest and latest start and end times of activities).

In ILOG SCHEDULE, an energetic resource represents the fact that the amount of energy that can be used over given intervals of time is limited. Roughly, an instance of the `CtDiscreteEnergy` class does not represent a resource per se, but the amount of work performed by the resource over regular intervals of time. The capacity of a `CtDiscreteEnergy` is not measured as a number of machines or as a number of men and

women, but as a number of machining hours or as a number of human-hours, human-weeks or human-months, spent for the performance of the scheduled activities (or made available through this performance).

The following code, for example, creates a `CtDiscreteEnergy` to represent the fact that at most 50 human-days of work can be spent each week. The time unit is supposed to be the day and date 0 corresponds to the beginning of the first week.

```
CtDiscreteEnergy* energy =
  new (CtHeap()) CtDiscreteEnergy(schedule, CtRequiredResource);
energy->makeTimeTable(0, 7 * numberOfWeeks, 7, 50);
```

Energetic reasoning is consequently performed for each period of the form `[7*n, 7*(n+1))`. In a sense, an energetic resource of ILOG SCHEDULE corresponds to a "piece of habograph" that is used to update earliest and latest start and end times of activities. Energetic resources are occasionally used in ILOG SCHEDULE applications to implement redundant constraints that result in more constraint propagation being performed.

4.1.3 What About Memory Consumption?

The previously described mechanisms have been implemented in several systems and shown to give interesting results as far as computational time is concerned. One issue however to be dealt with is the management of the additional data structures used to implement these techniques. Indeed these data structures may be more or less costly to maintain both in terms of CPU time and in terms of memory consumption. For example, the task intervals of [Caseau 94b] require the maintenance of a data structure in $O(n^2)$ where n is the number of activities requiring a given resource. Note that, in principle, this memory space consumption could be avoided, but at some expense in terms of CPU time. Similarly, the habograph of [Beck 92] uses a memory space in $O((h/g)^2)$ where h is the scheduling horizon and g the grain of the habograph.[3] Using implementation cues similar to those used in the sequential time-tables of ILOG SCHEDULE, the habograph could be reduced to time-intervals $[r_A\ d_B)$ such that r_A is the earliest start time of an activity A and d_B the latest end time of another (or the same) activity B. The space complexity would then be $O(n^2)$ as in [Caseau 94b].

Erschler, Lopez and Thuriot [Erschler 91] do not say much about their implementation choices in this respect. However, the use of the energetic rules for all pairs $[r_A\ d_B)$ obviously requires either $O(n^2)$ in space, or $O(n^3)$ in CPU time.

The less consuming techniques consequently seem to be those of [Carlier 90] and [Nuijten 93]. In the following, we describe an implementation of these techniques in ILOG SCHEDULE and the experimental results obtained.

4.2 An Extension of ILOG SCHEDULE Disjunctive Constraints

Three extensions of the disjunctive constraint of ILOG SCHEDULE have been implemented.

[3] An energetic resource in ILOG SCHEDULE uses a memory space in $O(h/g)$ only since the energetic analysis is restricted to time intervals of size g.

4.2.1 Energetic Reasoning and Restricted Edge-Finding

In a preliminary experiment, we combined the first rule of Erschler, Lopez and Thuriot (cf. section 4.1.2) with the edge-finding technique of Carlier and Pinson restricted to the overall set of yet unordered operations. With the resulting algorithm, MT10 was solved in about eight hours of CPU time, on a HP-700 workstation. While using this algorithm, we noticed that the number of backtracks required to prove the optimality of the solution varied with the order in which constraints were posted. This suggested that the energetic rule embedded in the algorithm violated fixpoint semantics. We consequently changed the rule as follows: rather than computing the energy required over the time interval $[r_A, d_B)$, we compute the energy W that would be required by other activities between the latest start time of A and the earliest end time of B if A were scheduled before B. In the case of the unary resource, this allows us to deduce that if A is before B, then at least W units of time must elapse between the end of A and the beginning of B. We can then use this W to update, still under the hypothesis that A is before B, the latest start time of A and the earliest end time of B. As long as W increases and is not large enough to prove that A cannot be before B, we iterate.

We decided to implement energetic reasoning using data structures in $O(n)$ rather than $O(n^2)$. A consequence of that is that, even without iterating the energy calculation, the implemented energetic reasoning algorithm runs in $O(n^3)$.[4] Using the same $O(n^3)$ loop, we were also able to directly apply the edge-finding technique to all the 3-tuples and $(n-1)$-tuples of activities (where n denotes the number of unscheduled activities requiring a given resource), without computing Jackson's preemptive schedule. The resulting algorithm was tested on three types of problems: a famous "bridge" scheduling problem that is often used as a simple benchmark in the constraint programming community [Van Hentenryck 89]; a number of job-shop scheduling problems, including the famous MT10; an industrial project scheduling problem submitted by a customer of ILOG and found to be extremely difficult to solve with propagation reduced to arc-consistency. These problems and the results obtained are described in section 4.3.

4.2.2 Complete Edge-Finding Using Jackson's Preemptive Schedule

The second propagation algorithm we implemented is based in Jackson's preemptive schedule. It is a direct implementation in ILOG SCHEDULE of the technique described by Carlier and Pinson in [Carlier 90]. However, our implementation computes Jackson's preemptive schedule in $O(n^2)$ rather than $O(n * log(n))$. This allowed us to re-use existing data structures rather than implementing new ones: it would be interesting to determine the number of activities n at which the price of additional data structures would be balanced by the resulting CPU time savings.

The algorithm also applies the rules of Caseau and Laburthe to a unique task interval for each resource, i.e., the task interval consisting of all unscheduled activities [Caseau 94b]. Note that this corresponds to some rules already proposed by Carlier and Pinson in [Carlier 89]: if an activity cannot be the input (output) of a 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).

The overall propagation algorithm runs in $O(n^2)$ (as in [Carlier 90]) and uses data structures in $O(n * log(n))$. It is applied iteratively as long as it results in updates of earliest and latest start and end times. Using ILOG SCHEDULE, it has been

[4] We did not try to implement an equivalent algorithm with both space and time complexity in $O(n^2)$. It was felt by the ILOG team that a memory consumption in $O(n^2)$ is rarely compatible with problem sizes and memory restrictions of industrial applications.

implemented in two days. It has been tested on the previously described problems. In most cases, it resulted in much less backtracking and smaller CPU times than the energetic reasoning algorithm of the previous section.

4.2.3 Complete Edge-Finding Using the Algorithm of Nuijten et al

The third propagation algorithm is similar to the one above but uses the edge-finding technique described in [Nuijten 93] rather than computing Jackson's preemptive schedule.

Normally, the two edge-finding algorithms result in the same deductions. However, in the case of the algorithm of Nuijten et al, we were incidentally able to deduce a bit more than the pure edge-finding technique. Indeed, each time the earliest start time of an activity is updated, we can note that the activity cannot be the first to execute among the yet unscheduled activities of the considered resource. This information can then be used to prune the search tree.

Such deductions are also possible in the case of the algorithm of Carlier and Pinson, but they are less numerous. As we shall see in the next section, the two algorithms provide slightly different results in terms of CPU time and number of backtracks.

4.3 Experimental Results

The three propagation algorithms above, i.e., the energetic reasoning algorithm (ERA), the edge-finding algorithm based on the computation of Jackson's preemptive schedule (EFJ), and the edge-finding algorithm based on [Nuijten 93] (EFN), have been tested on three types of problems:

- a famous "bridge" scheduling problem that is often used as a simple benchmark in the constraint programming community [Van Hentenryck 89];
- a number of job-shop scheduling problems, including the famous MT10;
- an industrial project scheduling problem submitted by a customer of ILOG and found to be extremely difficult to solve with propagation reduced to arc-consistency.

The problems and the results obtained are presented in the three following sections. Whenever possible, we also provide comparisons with the results published in the literature. The corresponding algorithms are referred to as CP for Carlier and Pinson [Carlier 90], AC for Applegate and Cook [Applegate 91], and CL for Caseau and Laburthe [Caseau 94b].

4.3.1 The Bridge Scheduling Problem

Problem Statement

The chosen problem consists of scheduling the construction of a five-segment bridge. This problem is very commonly used as a benchmark in the constraint programming community. In the following, we offer a simple, but yet efficient, program for solving the problem.

The bridge is made of five segments. Five preformed bearers must be placed on top of six pillars: one bearer between any two successive pillars. Following the notations

generally used for this problem, the activity which consists in positioning bearer number `k` on top of pillars number `k` and `k+1` is denoted `Tk`.

The construction of each pillar requires the accomplishment of four activities: excavation (denoted `Ak` for pillar number `k`), formwork (denoted `Sk`), concrete foundation (denoted `Bk`) and masonry (denoted `Mk`). In addition:

- The implementation of foundation piles `P1` and `P2` is required between the excavations of the central pillars `A3` and `A4` and the corresponding formworks `S3` and `S4`. Excavations and foundations precede the corresponding formworks. Also, no more than `3` days can elapse between the end of a particular foundation (or excavation for the pillars with no foundation activity) and the beginning of the corresponding formwork.
- A concrete setting period `ABk` is required between the concrete foundation `Bk` and the corresponding masonry work `Mk`. Hence, for each pillar, `Sk` precedes `Bk`, `Bk` precedes `ABk`, `ABk` precedes `Mk`, and both `Mk` and `M(k+1)` precede `Tk`.
- The time between the completion of a formwork `Sk` and the completion of the corresponding concrete foundation `Bk` is at most `4` days.
- The delivery of the preformed bearers (an activity denoted `L`) occurs exactly `30` days after the beginning of the construction project; it lasts two days. Positioning activities `Tk` cannot start before the end of the delivery activity `L`.
- On both sides of the bridge, filling activities `V1` and `V2` are required to allow proper connection of the bridge to the roadway. `V1` and `V2` cannot start before the end of the corresponding bearer positioning activities `T1` and `T5`.
- Temporary housing is needed for the construction workers. This entails both building (an activity denoted `UE`) and later disassembling (an activity denoted `UA`) temporary housing. The erection of the temporary housing `UE` must begin at least six days before the formworks `Sk`. The removal of the temporary housing `UA` can start at most two days before the end of the last masonry work `Mk`.
- The project end `PE` is an activity of null duration, which executes when the positioning activities `Tk`, the filling activities `Vk`, and the disassembling activity `UA` are completed.

The goal is to minimise the makespan of the overall project, i.e., the time at which the last activity `PE` ends.

The following table lists the activities to be scheduled and specifies, for each activity, its duration in days and the unary resoure to be used, if any is necessary.

Name	Activity description	Duration	Resource
A1	Excavation	4	Excavator
A2	Excavation	2	Excavator
A3	Excavation	2	Excavator
A4	Excavation	2	Excavator
A5	Excavation	2	Excavator
A6	Excavation	5	Excavator
P1	Foundation piles	20	Pile driver
P2	Foundation piles	13	Pile driver
S1	Formwork	8	Carpentry
S2	Formwork	4	Carpentry
S3	Formwork	4	Carpentry
S4	Formwork	4	Carpentry
S5	Formwork	4	Carpentry
S6	Formwork	10	Carpentry
B1	Concrete foundation	1	Concrete mixer
B2	Concrete foundation	1	Concrete mixer
B3	Concrete foundation	1	Concrete mixer
B4	Concrete foundation	1	Concrete mixer
B5	Concrete foundation	1	Concrete mixer
B6	Concrete foundation	1	Concrete mixer
AB1	Concrete setting time	1	
AB2	Concrete setting time	1	
AB3	Concrete setting time	1	
AB4	Concrete setting time	1	
AB5	Concrete setting time	1	
AB6	Concrete setting time	1	
M1	Masonry work	16	Bricklaying
M2	Masonry work	8	Bricklaying
M3	Masonry work	8	Bricklaying
M4	Masonry work	8	Bricklaying
M5	Masonry work	8	Bricklaying
M6	Masonry work	20	Bricklaying
T1	Positioning	12	Crane
T2	Positioning	12	Crane
T3	Positioning	12	Crane
T4	Positioning	12	Crane
T5	Positioning	12	Crane
V1	Filling	15	Caterpillar
V2	Filling	10	Caterpillar
L	Delivery of the preformed bearers	2	
UE	Erection of the temporary housing	10	
UA	Removal of the temporary housing	10	
PE	End of project	0	

Algorithm

The algorithm we have used to solve the scheduling problem works as follows:

1. Select a resource among the resources required by unordered activities.
2. Select the activity to execute first among the unordered activities that require the chosen resource. Post the corresponding precedence constraints. 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.

Each time a solution is found (with makespan C), we restart the process with an additional constraint, requiring the makespan to be strictly smaller than C. When the algorithm fails, we know that the previous C is the optimal makespan.

Scheduling problems are generally such that resources are not equally loaded. Some resources are, over some periods of time, more relied upon than others. These resources are often called "critical": their limited availability is a factor which prevents the reduction of a project cost or duration. It is, in general, very important to schedule critical resources first, in order to optimise the use of these resources without being bound by the schedule of other resources. A standard way to measure the "criticality" of a resource consists of comparing the "demand" for the resource to its availability ("supply") over a specific period of time. Here, for example, the time period under consideration may run from the soonest of the earliest start times of the activities to order, to the last of the latest end times of these activities. As the capacity of the resource is `1` at all times, the "supply" over this time interval is the length of the interval. The "demand" over the interval is the sum of the durations of the activities to execute. Criticality is then defined as the difference between "demand" and "supply". The main interest of ordering all the activities of the critical resource first is that when all these activities are ordered, there is very little work left to do because constraint propagation has automatically ordered activities on the other resources.

The next step requires the selection of the activity to schedule first. We choose the activity with the soonest earliest start time; in addition, the activity with the soonest latest start time is chosen when two or several activities share the same earliest start time.

Results

The following table compares the results obtained with a standard ILOG SCHEDULE program ARC (with propagation restricted to arc-consistency on the bounds of variables domains) to the results obtained using ERA, EFJ and EFN.

The columns of the tables represent:

- **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 HP715 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 (i.e., for the last iteration).
- **PT:** the amount of time required for the optimality proof, in seconds on a HP715 workstation.
- **TM:** the total amount of memory used for solving the problem, in kilobytes.

Results obtained by Caseau and Laburthe (CL) are also given, with computational times in seconds on a SPARC1 workstation.

COMPUTATIONAL RESULTS						
Algo.	TF	TT	IT	PF	PT	TM
ARC	176	.3	5	149	.1	76
ERA	15	.6	5	15	.2	72
EFJ	14	.6	5	12	.1	72
EFN	14	.5	5	12	.1	72
CL	N.A.	.4	N.A.	0	.0	N.A.

On this experiment, we see that the number of backtracks significantly decreases when more propagation is being performed. Also, we see that on such a problem, the algorithm based on arc-consistency performs better than the others in terms of CPU time: the cost of additional constraint propagation is not balanced by the reduction of the search effort.

The following section describes experiments made on more difficult problems.

4.3.2 Job-Shop Scheduling Problems

Problem Statement

The Job-Shop Scheduling Problem (JSSP) consists of n jobs to be performed using m machines. The benchmark problems used in our computational study are such that each job consists of m activities (of given processing times) to be executed in a specified order. Each activity requires a specified machine and each machine is required by a unique activity of each job. Job-shop scheduling is an NP-complete problem [Garey 79], known to be among the worst in this class. Many different methods have been tested. Most of them are branch and bound methods based upon a disjunctive graph, but other approaches have been used with some success as well: linear integer programming, genetic algorithms, simulated annealing ...

Algorithm

A variant of the algorithm used for the bridge construction scheduling problem has been used to solve job-shop problems. The basic structure of the algorithm is exactly the same as the one used for the bridge problem:

1. Select a resource among the resources required by unordered activities.
2. Select the activity to execute first among the unordered activities that require the chosen resource. Post the corresponding precedence constraints. 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.

However, we made two important modifications:

- Rather than just using a criticality criterion to select the resource to schedule (step 1 of the algorithm), we rely on two criteria. For each resource, we compute (1) the number of backtracks required to solve the problem restricted to the activities of that resource and (2) the "criticality" as defined in section 4.3.1. The second criterion is used to distinguish between resources that are equal with respect to the first criterion. Note that the computation of (1) requires search for solutions to the m one-machine problems: this search induces backtracks and may also fail. If it fails (for a given maximal makespan), it proves that the m-machines problem (for the same maximal makespan) has no solution.

- The second modification concerns the minimisation of the makespan value. Rather than decreasing step-by-step the makespan maximal value, we dichotomise 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.

Note that in some cases, the propagation of a new upper bound for the makespan variable results in a new lower bound for the same variable. Indeed, it is determined that to satisfy the upper bound, some activities have to be scheduled in a given order. This, in turn, results in an update of the minimal makespan.

As far as optimality proofs are concerned, two cases can consequently occur:

- either imposing an upper bound UB greater than or equal to the optimal makespan M is sufficient to prove that M is a lower bound; the proof results from constraint propagation alone.
- or imposing M as an upper bound is not sufficient to prove that M is a lower bound; the proof consequently requires some amount of backtracking search.

Results

The three variants of the propagation algorithm, i.e., ERA, EFJ and EFN, have been applied to a number of problems including:

- 8 problems from [Carlier 84] (chapter 4 section 5). These problems (CAR1 to CAR8) are in fact 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.
- 10 problems (ORB1 to ORB10) generated by a group of people in Bonn with the challenge to make the problems difficult (cf. [Applegate 91]). Using ERA, we were not able to solve ORB1 and ORB2 in three hours of CPU time. Subsequently, only the EFJ and EFN algorithms were applied to this set of problems.
- 4 problems mentioned in [Adams 88] including the famous MT10 problem. These problems are known as MT06, MT10, ABZ5 and ABZ6.
- 60 problems mentioned in [Sadeh 91], which have been widely used in the constraint-based scheduling community. These problems are much smaller than the others (50 activities for each problem), but are interesting because they have been constructed to reflect six different types of scheduling situations. The problems are numbered ExDDRy-z. "x" refers to additional constraints concerning job release and due-dates: x=0 means that all the release-dates are the same and that all the due-dates are the same; x=n (narrow) means that the release-dates (due-dates) are close one to the other; x=w (wide) means the release-dates (due-dates) vary a lot from one job to the other. "y" is the number of bottleneck resources of the considered problem: y=1 means there is a unique bottleneck; y=2 means there are two bottlenecks, which makes the problem potentially harder. "z" is an experiment index running from 1 to 10 for each possible value of the pair (x y).

A table of results is provided for each problem set. When possible, we also provide the results published in [Carlier 90] (CP), [Applegate 91] (AC), and [Caseau 94b] (CL). For

the [Sadeh 91] problems, we provide detailed results only for those problems which were reasonably hard to solve. Other results are averaged for each (x y) series.

The columns of the tables represent:

1. **Problem:** the reference to the JSSP 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.
2. **Algo.:** the algorithm used.
3. **Mak.:** the optimal makespan.
4. **TF:** the total number of backtracks required to solve the problem (solution + optimality proof).
5. **TT:** the total amount of time required to solve the problem (solution + optimality proof), in seconds on a HP715 workstation.
6. **IT:** the number of iterations (restartings of the algorithm with a new makespan upper bound) necessary to solve the problem.
7. **PF:** the number of backtracks required for the optimality proof. In this column, "IMM" means that the proof was immediate.
8. **PT:** the amount of time required for the optimality proof, in seconds on a HP715 workstation. In this column, "IMM" means that the proof was immediate.
9. **TM:** the total amount of memory used for solving the problem, in kilobytes.

For CP, the computational times are in seconds on a PRIME2655. For AC and CL, the computational times are in seconds on a SPARC1 workstation. In the case of AC, we use the number of nodes given in [Applegate 91] as number of backtracks. In reality, the number of backtracks is necessarily smaller than the number of nodes.

CAR* PROBLEMS: COMPUTATIONAL RESULTS								
Problem	**Algo.**	**Mak.**	TF	TT	IT	PF	PT	TM
CAR1(11x5)	ERA	7038	76	11	12	IMM	IMM	84
	EFJ		50	3	12	IMM	IMM	84
	EFN		50	2	12	IMM	IMM	88
CAR2(13x4)	ERA	7166	532	39	11	4	0	84
	EFJ		716	11	12	IMM	IMM	84
	EFN		710	11	12	IMM	IMM	84
CAR3(12x5)	ERA	7312	982	84	11	IMM	IMM	96
	EFJ		957	23	9	IMM	IMM	96
	EFN		957	21	9	IMM	IMM	96
CAR4(14x4)	ERA	8003	57	10	13	IMM	IMM	84
	EFJ		7437	121	13	IMM	IMM	88
	EFN		7247	115	13	IMM	IMM	92
CAR5(10x6)	ERA	7702	63697	3189	12	6903	307	92
	EFJ		24054	521	12	1662	34	92
	EFN		24052	522	12	1661	34	96
CAR6(8x9)	ERA	8313	11756	705	12	951	54	104
	EFJ		12886	326	12	1165	27	104
	EFN		12876	320	12	1165	25	108
CAR7(7x7)	ERA	6558	4717	155	12	787	24	76
	EFJ		3295	64	12	510	9	76
	EFN		3273	56	12	505	9	76
CAR8(8x8)	ERA	8264	10828	555	11	1983	100	96
	EFJ		5361	120	12	1173	29	96
	EFN		5360	107	12	1173	26	96

The table above provides the results for the 8 problems from [Carlier 84]. From these results, it is clear that in terms of CPU time, the EFJ/EFN algorithms dominate the ERA algorithm. An exception to that rule is the CAR4 problem for which ERA performs much better than EFJ/EFN. However, by tracing the resolution of that problem in more details, we have seen that ERA makes a "lucky" decision in a case in which two activities are equivalent in terms of both their earliest and latest start times. By modifying slightly the algorithm to prevent that "lucky" decision from being made, we recovered results in the same order of magnitude as the EFJ/EFN results.

An interesting remark is that the advantage of EFJ/EFN is less obvious in terms of numbers of backtracks than in terms of CPU time. This suggests that the energetic reasoning of ERA is powerful as far as pruning the search space is concerned. However, it costs a lot of CPU time compared to EFJ/EFN. This is not a surprise given the algorithmic complexity of the two algorithms. On some problems, we tried to combine energetic reasoning with full edge-finding. The results were disappointing as the combination did not prune the search space much more than each of ERA and EFJ taken separately.

Another remark is that EFJ and EFN are very close to each other in terms of CPU time. EFN seems to perform a little better. However, as we shall see in the following tables, this is not always the case.

ORB* PROBLEMS: COMPUTATIONAL RESULTS								
Problem	**Algo.**	**Mak.**	TF	TT	IT	PF	PT	TM
ORB1(10x10)	EFJ	1059	87915	2816	9	5322	215	140
	EFN		87644	2940	9	5322	224	140
	AC		71812	1483	N.A.	N.A.	N.A.	N.A.
ORB2(10x10)	EFJ	888	147196	5061	10	39061	1362	140
	EFN		146702	5446	10	38952	1458	140
	AC		153578	2484	N.A.	N.A.	N.A.	N.A.
	CL		N.A.	N.A.	N.A.	16400	1200	N.A.
ORB3(10x10)	EFJ	1005	245349	8993	9	26501	1048	140
	EFN		245126	9597	9	26501	1118	140
	AC		130181	2297	N.A.	N.A.	N.A.	N.A.
ORB4(10x10)	EFJ	1005	80205	3576	8	27938	1214	140
	EFN		80171	3742	8	27938	1272	140
	AC		44547	1013	N.A.	N.A.	N.A.	N.A.
ORB5(10x10)	EFJ	887	77567	2128	9	5418	173	140
	EFN		77384	2236	9	5360	177	140
	AC		23113	526	N.A.	N.A.	N.A.	N.A.
ORB6(10x10)	EFJ	1010	286595	11737	9	148039	6054	140
	EFN		285053	12431	9	147254	6417	140
ORB7(10x10)	EFJ	397	44614	1474	9	8215	266	140
	EFN		44482	1514	9	8169	272	140
ORB8(10x10)	EFJ	899	659	26	10	60	2	140
	EFN		659	27	10	60	2	140
ORB9(10x10)	EFJ	934	45215	1688	10	14572	527	140
	EFN		45215	1767	10	14572	552	140
ORB10(10x10)	EFJ	944	6832	291	9	2387	107	140
	EFN		6824	309	9	2385	115	140

The table above provides the results for ORB problems [Applegate 91]. The results reported in [Applegate 91] and [Caseau 94b] are significantly better than ours. It is worth noting that rather complex heuristics have been used in both cases, in [Applegate 91] to get a very good initial solution, in [Caseau 94b] to shorten the proof. On the contrary, we have been using very simple heuristics, both for finding solutions and for proving the inexistence of solutions at a given makespan. Obviously, the use of elaborate heuristics is paying off in terms of CPU time.

ABZ*-MT* PROBLEMS: COMPUTATIONAL RESULTS								
Problem	**Algo.**	**Mak.**	TF	TT	IT	PF	PT	TM
ABZ5(10x10)	EFJ	1234	45187	1369	10	19574	590	136
	EFN		44552	1491	10	19246	649	136
	AC		57848	952	N.A.	N.A.	N.A.	N.A.
ABZ6(10x10)	ERA	943	1792	155	9	439	40	136
	EFJ		1076	49	9	305	13	136
	EFN		1068	52	9	302	14	136
	AC		1269	91	N.A.	1269	N.A.	N.A.
MT06(6x6)	ERA	55	25	1	5	3	0	64
	EFJ		10	0	5	3	0	64
	EFN		10	0	5	3	0	64
	CP		1	1	N.A.	0	0	N.A.
MT10(10x10)	ERA	930	99189	9651	10	19243	1854	140
	EFJ		52174	2052	10	8148	333	140
	EFN		51980	2194	10	8137	355	140
	CP		5036	5943	N.A.	4203	4400	N.A.
	AC		16055	372	N.A.	16055	315	N.A.
	CL		N.A.	1270	N.A.	6728	730	N.A.

The table above provides the results for ABZ and MT problems [Adams 88]. A quick look at the colums providing the number of backtracks and the CPU time for the optimality proofs shows that the algorithms based on EFJ/EFN are in the same range of efficiency as the algorithms reported in [Carlier 90] [Applegate 91] [Caseau 94b]. This is not true for the algorithm based on ERA, which remains significantly slower than the other algorithms.

With respect to the total number of backtracks and CPU time, the figures provided by [Carlier 90], [Applegate 91] and [Caseau 94b] are significantly smaller than those obtained by our algorithms. However:

- Carlier and Pinson [Carlier 90] start their search from a heuristically determined value as an upper bound for the makespan.[5] This heuristic provides a very good upper bound in the case of MT10, thereby reducing the time wasted in the search of solutions.
- Applegate and Cook [Applegate 91] use distinct heuristic methods prior to the branch-and-bound algorithm. These heuristic methods, which provide good upper bounds for the ABZ5 and ABZ6 problems, also happen to provide a solution of makespan 930 in the case of MT10. Then, the proof of optimality is the only thing that remains to be done.

[5] From what we understood, this value is not previously proven to be an upper bound.

The three following tables provide results obtained on the ExDDRy-z PROBLEMS generated by Norman Sadeh [Sadeh 91]. Only the average results are provided for the one-bottleneck problems. Detailed results are provided for the instances with two bottlenecks. In most cases, EFJ/EFN perform better than ERA on these instances. Note that the first solution to these problems is always found without backtracking (even when there are release-dates and due-dates), in an average CPU time of 0.16 seconds.

E0DDRy-z PROBLEMS: COMPUTATIONAL RESULTS								
Problem	**Algo.**	**Mak.**	TF	TT	IT	PF	PT	TM
E0DDR1 (average)	ERA	N.A.	7	4	8	IMM	IMM	76
	EFJ		5	1	6	IMM	IMM	76
	EFN		5	1	6	IMM	IMM	80
E0DDR2-1	ERA	157	354	26	8	IMM	IMM	76
	EFJ		91	2	8	IMM	IMM	76
	EFN		91	3	8	IMM	IMM	80
E0DDR2-2	ERA	162	74	9	8	IMM	IMM	80
	EFJ		74	3	8	IMM	IMM	76
	EFN		74	2	8	IMM	IMM	80
E0DDR2-3	ERA	150	47	6	8	IMM	IMM	76
	EFJ		49	2	8	IMM	IMM	76
	EFN		49	2	8	IMM	IMM	80
E0DDR2-4	ERA	141	219	14	7	IMM	IMM	76
	EFJ		165	4	8	IMM	IMM	76
	EFN		165	4	8	IMM	IMM	80
E0DDR2-5	ERA	141	320	15	8	IMM	IMM	76
	EFJ		21	2	8	IMM	IMM	76
	EFN		21	1	8	IMM	IMM	80
E0DDR2-6	ERA	145	914	30	8	IMM	IMM	76
	EFJ		19	1	7	IMM	IMM	76
	EFN		19	1	7	IMM	IMM	80
E0DDR2-7	ERA	151	10	7	8	IMM	IMM	76
	EFJ		11	2	8	IMM	IMM	76
	EFN		11	2	8	IMM	IMM	80
E0DDR2-8	ERA	145	4230	79	8	21	1	76
	EFJ		90	2	7	21	0	76
	EFN		90	2	7	21	0	80
E0DDR2-9	ERA	162	3467	319	8	IMM	IMM	80
	EFJ		2982	58	8	IMM	IMM	76
	EFN		2982	58	8	IMM	IMM	80
E0DDR2-10	ERA	138	829	41	8	IMM	IMM	76
	EFJ		6342	194	5	IMM	IMM	76
	EFN		6342	197	5	IMM	IMM	80

ENDDRy-z PROBLEMS: COMPUTATIONAL RESULTS								
Problem	**Algo.**	**Mak.**	TF	TT	IT	PF	PT	TM
ENDDR1 (average)	ERA	N.A.	5	3	7	IMM	IMM	76
	EFJ		4	1	6	IMM	IMM	76
	EFN		4	1	6	IMM	IMM	80
ENDDR2-1	ERA	158	37	8	8	IMM	IMM	80
	EFJ		25	2	8	IMM	IMM	76
	EFN		25	1	8	IMM	IMM	80
ENDDR2-2	ERA	167	52	6	8	IMM	IMM	76
	EFJ		52	2	8	IMM	IMM	76
	EFN		52	2	8	IMM	IMM	80
ENDDR2-3	ERA	151	33	8	7	IMM	IMM	76
	EFJ		68	3	8	IMM	IMM	76
	EFN		68	3	8	IMM	IMM	80
ENDDR2-4	ERA	142	59	7	8	IMM	IMM	76
	EFJ		148	3	7	IMM	IMM	76
	EFN		148	3	7	IMM	IMM	80
ENDDR2-5	ERA	153	28	8	8	IMM	IMM	76
	EFJ		20	2	8	IMM	IMM	76
	EFN		20	2	8	IMM	IMM	80
ENDDR2-6	ERA	155	601	24	7	IMM	IMM	76
	EFJ		3830	29	7	IMM	IMM	76
	EFN		3437	26	7	IMM	IMM	80
ENDDR2-7	ERA	154	19	6	8	IMM	IMM	76
	EFJ		17	2	7	IMM	IMM	76
	EFN		17	1	7	IMM	IMM	80
ENDDR2-8	ERA	152	19913	823	8	IMM	IMM	76
	EFJ		6822	115	8	IMM	IMM	76
	EFN		6822	122	8	IMM	IMM	80
ENDDR2-9	ERA	165	5	2	7	IMM	IMM	76
	EFJ		5	1	7	IMM	IMM	76
	EFN		5	1	7	IMM	IMM	80
ENDDR2-10	ERA	145	140	10	7	3	0	76
	EFJ		87	3	7	3	0	76
	EFN		87	3	7	3	0	80

EWDDRy-z PROBLEMS: COMPUTATIONAL RESULTS								
Problem	**Algo.**	**Mak.**	TF	TT	IT	PF	PT	TM
EWDDR1 (average)	ERA	N.A.	5	3	7	IMM	IMM	76
	EFJ		4	1	6	IMM	IMM	76
	EFN		4	1	6	IMM	IMM	80
EWDDR2-1	ERA	161	22	7	8	IMM	IMM	76
	EFJ		17	1	7	IMM	IMM	76
	EFN		17	1	7	IMM	IMM	80
EWDDR2-2	ERA	168	49	7	8	IMM	IMM	76
	EFJ		50	2	8	IMM	IMM	76
	EFN		50	2	8	IMM	IMM	80
EWDDR2-3	ERA	152	44	8	8	IMM	IMM	76
	EFJ		33	2	7	IMM	IMM	76
	EFN		33	1	7	IMM	IMM	80
EWDDR2-4	ERA	148	24	4	7	IMM	IMM	76
	EFJ		15	1	6	IMM	IMM	76
	EFN		15	1	6	IMM	IMM	80
EWDDR2-5	ERA	160	8	2	7	IMM	IMM	76
	EFJ		7	1	7	IMM	IMM	76
	EFN		7	1	7	IMM	IMM	80
EWDDR2-6	ERA	164	3855	138	8	IMM	IMM	76
	EFJ		340	6	5	IMM	IMM	76
	EFN		340	6	5	IMM	IMM	80
EWDDR2-7	ERA	157	13	8	8	IMM	IMM	76
	EFJ		10	2	7	IMM	IMM	76
	EFN		10	2	7	IMM	IMM	80
EWDDR2-8	ERA	158	114	12	8	IMM	IMM	76
	EFJ		269	6	8	IMM	IMM	76
	EFN		269	5	8	IMM	IMM	80
EWDDR2-9	ERA	166	4	4	8	IMM	IMM	76
	EFJ		5	1	7	IMM	IMM	76
	EFN		5	1	7	IMM	IMM	80
EWDDR2-10	ERA	148	65	8	7	IMM	IMM	76
	EFJ		89	3	7	IMM	IMM	76
	EFN		89	3	7	IMM	IMM	80

4.3.3 An Industrial Project Scheduling Problem

Problem Statement

This section reports results concerning an industrial project scheduling problem submitted by a customer of ILOG and found to be extremely difficult to solve with propagation reduced to arc-consistency. The problem consists of scheduling two projects that require common resources. There are fourty-five activities and five resources to consider. Each activity requires up to four resources. Within each project, activities are subjected to precedence constraints. Three of the resources are unary resources. The number of activities that require each unary resource is close to thirty. The two other resources are discrete resources with capacity greater than one.

There are two optimisation criteria: the goal is to minimise the scheduled end times of two activities, one for each project. As the projects rely on common resources, these two optimisation criteria are conflicting. As a result, the final user wants to impose upper bounds on the two criteria, and wants the system to tell whether there exists a solution satisfying these upper bounds.

Algorithm

Given that discrete resources are involved, it is not possible to use the same algorithm as for the bridge and the job-shop scheduling problems. The algorithm we have used is the one described in [Le Pape 94c].

1. Initialise the set of selectable activities to the complete set of activities to schedule.
2. If all the activities have fixed start and end times, set the makespan variable to the makespan of the obtained solution and exit. Otherwise, remove from the set of selectable activities those activities which have fixed start and end times.
3. If the set of selectable activities is not empty, select an activity from the set, create a choice point for the selected activity (to allow backtracking) and schedule the selected activity from its earliest start time to its earliest end time. Then goto step 2.
4. If the set of selectable activities is empty, backtrack to the most recent choice point. (If there is no such choice point, report that there is no problem solution and exit.)
5. Upon backtracking, mark the activity that was scheduled at the considered choice point as not selectable as long as its earliest start and end times have not changed. Then goto step 2.

The correctness of this algorithm is easy to demonstrate. Indeed, the activity A chosen at step 3 must either start at its earliest start time (and consequentlty end at its earliest end time) or must be postponed to start later. But starting A later makes sense only if other activities prevent A from starting at its earliest start time, in which case the scheduling of these other activities must eventually result (thanks to constraint propagation!) in the update of the earliest start and end times of A. Hence a backtrack from step 4 signifies that all the activities which do not have fixed start and end times have been postponed. This is absurd as in any solution the activity which will start the earliest among those which have been postponed could be set to start at its earliest start time. This is why it is correct to backtrack from step 4 up to the most recent choice point.

Results

The algorithm was applied with different propagation procedures, i.e., ARC (arc-consistency on the bounds of variables domains), ERA, EFJ and EFN. Each of the tables below provides CPU times obtained for different values of the upper bounds of the two criteria. In these tables, "****" means that the algorithm could not solve the problem in less than one hour of computational time.

CPU TIME FOR SOLVING THE PROBLEM													
	Algo.	117	118	119	120	121	122	123	124	125	126	127	128
117	ARC	1	4	5	5	6	7	9	11	15	19	21	24
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
118	ARC	1	4	5	5	6	9	15	18	21	26	35	42
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
119	ARC	1	4	5	6	6	10	16	27	34	45	47	60
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
120	ARC	2	4	6	6	7	11	17	29	39	52	57	73
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
121	ARC	2	5	11	11	12	49	61	72	78	108	164	187
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
122	ARC	2	8	13	20	22	64	122	142	155	187	276	412
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
123	ARC	4	10	20	31	32	80	144	214	248	300	372	564
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
124	ARC	4	13	22	32	49	97	163	249	323	396	528	689
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
125	ARC	5	12	29	40	59	133	205	294	387	632	832	1076
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
126	ARC	7	13	28	41	66	147	273	388	495	790	1332	1665
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
127	ARC	6	20	28	40	73	167	357	468	628	963	1580	2220
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
128	ARC	8	22	41	53	81	202	396	604	751	1175	1875	2573
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1

CPU TIME FOR SOLVING THE PROBLEM													
	Algo.	117	118	119	120	121	122	123	124	125	126	127	128
129	ARC	11	27	52	77	127	234	518	708	919	1484	2373	3182
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
130	ARC	13	40	65	100	184	281	606	990	1172	1934	3258	****
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
131	ARC	15	44	85	126	234	423	729	1146	1752	2580	****	****
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
132	ARC	18	49	105	181	282	496	1237	1502	2110	****	****	****
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
133	ARC	26	61	122	232	377	548	1398	2524	3057	****	****	****
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
134	ARC	27	75	148	276	514	752	1469	2933	****	****	****	****
	ERA	4	4	4	4	4	4	4	4	4	4	4	4
	EFJ	0	0	0	0	0	0	0	0	0	0	0	0
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
135	ARC	28	88	187	356	635	1025	1903	3233	****	****	****	****
	ERA	9	15	25	26	27	113	142	143	143	143	455	630
	EFJ	1	1	6	10	14	52	60	62	62	59	152	140
	EFN	1	1	5	10	13	49	60	58	58	60	149	146
136	ARC	30	101	138	247	498	884	1891	3021	****	****	****	****
	ERA	12	30	12	12	12	12	12	12	12	12	12	12
	EFJ	1	2	1	1	1	1	1	1	1	1	1	1
	EFN	1	2	1	1	1	1	1	1	1	1	1	1
137	ARC	41	108	188	217	354	681	1522	3454	****	****	****	****
	ERA	12	53	12	12	12	12	12	12	12	12	12	12
	EFJ	1	3	1	1	1	1	1	1	1	1	1	1
	EFN	1	3	1	1	1	1	2	1	2	1	2	2
138	ARC	45	88	187	307	477	858	1934	****	****	****	****	****
	ERA	12	12	12	12	12	12	12	12	12	12	12	12
	EFJ	1	1	1	1	1	1	1	1	1	1	1	1
	EFN	1	1	1	1	1	1	1	1	2	1	1	1
139	ARC	1	0	1	1	1	0	0	0	1	0	0	0
	ERA	12	12	12	12	12	12	12	12	12	12	12	12
	EFJ	1	1	1	1	1	1	1	1	1	1	1	1
	EFN	1	1	1	1	1	1	1	1	1	1	1	1
140	ARC	1	0	1	1	1	1	0	0	1	0	0	1
	ERA	12	12	12	12	12	12	12	12	12	12	12	12
	EFJ	1	1	1	1	1	1	1	1	1	1	1	1
	EFN	1	1	1	1	1	1	1	1	1	1	2	1

CPU TIME FOR SOLVING THE PROBLEM													
	Algo.	129	130	131	132	133	134	135	136	137	138	139	140
117	ARC	36	55	79	92	119	164	219	227	171	87	1	1
	ERA	4	4	4	4	4	4	35	129	44	73	12	12
	EFJ	0	0	0	0	0	0	4	13	2	4	1	1
	EFN	1	1	1	1	1	1	5	16	2	6	1	1
118	ARC	56	93	130	183	232	316	433	245	323	143	1	1
	ERA	4	4	4	4	4	4	37	14	44	73	12	12
	EFJ	0	0	0	0	0	0	9	1	2	5	1	1
	EFN	1	1	1	1	1	1	10	1	2	6	1	1
119	ARC	79	113	167	283	445	564	828	395	162	264	1	1
	ERA	4	4	4	4	4	4	37	14	45	75	12	12
	EFJ	0	0	0	0	0	0	28	1	2	5	1	1
	EFN	1	1	1	1	1	1	25	1	2	6	1	1
120	ARC	90	131	189	373	564	805	1188	621	205	383	1	1
	ERA	4	4	4	4	4	4	39	15	46	75	12	12
	EFJ	0	0	0	0	0	0	30	1	2	4	1	1
	EFN	1	1	1	1	1	1	31	1	2	6	1	1
121	ARC	229	293	442	555	831	1181	2009	1058	320	525	1	1
	ERA	4	4	4	4	4	4	92	14	48	75	12	12
	EFJ	0	0	0	0	0	0	44	1	2	5	1	1
	EFN	1	1	1	1	1	1	46	1	2	6	1	1
122	ARC	490	600	763	1294	1647	2051	3055	2022	613	874	1	1
	ERA	4	4	4	4	4	4	117	17	48	75	12	12
	EFJ	0	0	0	0	0	0	50	1	2	5	1	1
	EFN	1	1	1	1	1	1	49	1	2	6	1	1
123	ARC	741	886	1140	1805	3046	****	****	3299	1305	1974	1	1
	ERA	4	4	4	4	4	4	117	17	49	75	12	12
	EFJ	0	0	0	0	0	0	50	1	2	5	1	1
	EFN	1	1	1	1	1	1	49	1	2	6	1	2
124	ARC	910	1191	1672	2335	****	****	****	****	1724	3025	1	1
	ERA	4	4	4	4	4	4	121	17	50	75	12	12
	EFJ	0	0	0	0	0	0	50	1	2	5	1	1
	EFN	1	1	1	1	1	1	49	1	2	6	1	2
125	ARC	1229	1771	2604	****	****	****	****	****	2218	****	1	1
	ERA	4	4	4	4	4	4	118	17	48	75	12	12
	EFJ	0	0	0	0	0	0	50	1	2	5	1	1
	EFN	1	1	1	1	1	1	50	1	2	6	1	2
126	ARC	1959	2481	****	****	****	****	****	****	3459	****	1	1
	ERA	4	4	4	4	4	4	324	18	48	74	12	12
	EFJ	0	0	0	0	0	0	98	1	2	5	1	1
	EFN	1	1	1	1	1	1	98	1	3	6	1	2
127	ARC	2645	3427	****	****	****	****	****	****	****	****	1	1
	ERA	4	4	4	4	4	4	373	18	53	73	12	12
	EFJ	0	0	0	0	0	0	100	1	2	5	1	1
	EFN	1	1	1	1	1	1	101	1	3	6	1	2
128	ARC	3257	****	****	****	****	****	****	****	****	****	0	1
	ERA	4	4	4	4	4	4	560	18	52	74	12	12
	EFJ	0	0	0	0	0	0	89	1	2	6	1	1
	EFN	1	1	1	1	1	1	93	2	3	6	1	2

CPU TIME FOR SOLVING THE PROBLEM													
	Algo.	129	130	131	132	133	134	135	136	137	138	139	140
129	ARC	****	****	****	****	****	****	****	****	****	****	0	1
	ERA	4	4	4	4	4	4	950	14	52	80	12	12
	EFJ	0	0	0	0	0	0	110	1	2	6	1	1
	EFN	1	1	1	1	1	1	116	2	3	6	1	2
130	ARC	****	****	****	****	****	****	****	****	256	****	1	1
	ERA	4	4	4	4	4	4	938	16	12	69	12	12
	EFJ	0	0	0	0	0	0	109	1	1	5	1	1
	EFN	1	1	1	1	1	1	117	2	2	6	1	2
131	ARC	****	****	****	****	****	****	****	****	****	1	1	1
	ERA	4	4	4	4	4	4	1334	14	12	12	12	12
	EFJ	0	0	0	0	0	0	156	1	1	1	1	1
	EFN	1	1	1	1	1	1	171	2	2	1	1	2
132	ARC	****	****	****	****	****	****	****	****	****	****	1	1
	ERA	4	4	4	4	4	4	1701	16	12	12	12	12
	EFJ	0	0	0	0	0	0	175	1	1	1	1	1
	EFN	1	1	1	1	1	1	197	2	2	1	2	2
133	ARC	****	****	****	****	****	****	****	****	****	****	****	1
	ERA	4	4	4	4	4	4	2062	17	12	12	12	12
	EFJ	0	0	0	0	0	0	185	1	1	1	1	1
	EFN	1	1	1	1	1	1	216	2	1	1	2	2
134	ARC	****	****	****	****	****	****	****	****	****	****	1	****
	ERA	4	4	4	4	4	4	2055	20	12	12	12	12
	EFJ	0	0	0	0	0	0	186	1	1	1	1	1
	EFN	1	1	1	1	1	1	215	2	2	1	1	2
135	ARC	****	****	****	****	****	****	****	****	****	****	1	1
	ERA	933	934	1241	1829	2105	2109	2718	1002	12	12	12	12
	EFJ	156	157	191	227	225	229	271	103	1	1	1	1
	EFN	168	171	203	253	260	265	299	113	2	1	1	2
136	ARC	****	****	****	****	****	****	****	****	****	****	1	1
	ERA	12	12	12	14	16	16	16	20	20	12	12	12
	EFJ	1	1	1	1	1	2	2	2	2	1	1	1
	EFN	2	2	2	2	2	2	2	2	2	1	1	2
137	ARC	****	****	****	****	****	****	****	****	****	****	1	1
	ERA	12	12	12	12	12	12	12	12	12	12	12	12
	EFJ	1	1	1	1	1	1	1	1	1	1	1	1
	EFN	2	2	2	2	2	2	2	2	2	1	2	2
138	ARC	****	****	****	****	****	****	****	****	****	****	1	1
	ERA	12	12	12	12	12	12	12	12	12	12	12	12
	EFJ	1	1	1	1	1	1	1	1	1	1	1	1
	EFN	2	1	2	2	2	1	2	2	2	1	2	2
139	ARC	0	0	1	1	1	1	1	1	1	1	1	0
	ERA	12	12	12	12	12	12	12	12	12	12	12	12
	EFJ	1	1	1	1	1	1	1	1	1	1	1	1
	EFN	1	1	1	1	1	1	1	2	2	2	2	2
140	ARC	0	1	1	1	0	1	1	1	1	1	1	1
	ERA	12	12	12	12	12	12	12	12	12	12	12	12
	EFJ	1	1	1	1	1	1	1	1	1	1	1	1
	EFN	2	1	2	2	2	2	2	2	2	2	2	2

It appears that the algorithm based on ERA is much more robust than the one based on ARC, but is an order of magnitude slower for the easiest problems. The algorithms based on EFJ/EFN "win" as they enjoy the robustness of the algorithm based on ERA without paying the $O(n^3)$ price in CPU time.

5

Conclusion

The prototypes that have been implemented seem to be interesting possible extensions of the ILOG SCHEDULE software. Actually, they allow (1) to model interruptions (interruptible activities) and (2) to use an extended propagation for unary resources. According to the computational experiments, a large part of the problems that we have tested have been solved far more easily than before.

It is interesting to notice that research on this subject is still carried on. As a matter of conclusion, we will highlight a recent result of Carlier and Pinson. In [Carlier 94], an algorithm to adjust the bounds of activities to be scheduled on a unary reource in $O(n \log n)$ instead of $O(n^2)$ is detailed. Unfortunately, we did not have enough time to implement this algorithm and to compare it to the others. The numerous papers published on this subject, and more generally on constraint-based scheduling show that this domain is still very open.

6

Annex A: Preemptive Scheduling

Preemptive scheduling problems are sometimes easier than non-preemptive ones. It occurs that NP-hard scheduling problems have polynomial preemptive solutions. Although algorithms as well as completeness results are often unknown we give an overview of some of the most common results concerning preemptive problems. In this annex, we describe the most well-known algorithms minimizing one criterion among the following ones: the makespan (i.e., the completion time of the latest activity), the average lateness (equivalent to average completion time), the maximal lateness. After having recalled the notations, we will focus on the one-machine problem. In a later section, we will describe some results for the n machines problem and discuss a special instance (the minimisation of the average lateness) of this problem.

Notations

Let n be the number of activities and m the number of processors.

An activity A_i is caracterised by:

- its total duration p_i
- its earliest start time (release date) r_i
- its latest end time (due date) d_i

The criterions are:

- the Makespan M
- the maximum lateness $L_{max} = max(L_i) = \max(endTime(A_i) - d_i)$. Note that a solution which minimises maximum lateness also minimises maximal tardiness.
- the average lateness $|L| = \frac{1}{n} * \sum L_i$

6.1 The One-Machine Problem

6.1.1 Scheduling with Release-Dates and Due-Dates

The problem is to find a schedule that satisfies the release time constraints and meets all the deadlines. In the non-preemptive case, this problem is NP-hard [Garey 77]. In the preemptive case, [Lawler 73] proved that it could be solved in a polynomial time.

6.1.2 Makespan Minimisation

This problem may be solved polynomially with the following algorithm:

Let t be the current date. One unit of one of an activity schedulable at t (i.e., which can start at t) is scheduled at t. The residual duration of this activity is decreased of 1 and its release date is increased of 1.

Proof (rough sketch)

Let H be the previously described schedule. We are then brought to distinguish two cases:

- if there is a date $t \leq M$ at which no activity is executed then, according to the algorithm, this means that all the activities A_i such that $r_i \leq t$ are scheduled before t. Then, we can then consider that the restriction of the schedule to the time interval $[0, t)$ is optimal. We can then iterate the proof on a smaller problem (i.e., the makespan minimisation of the activities which are not finished at the time t).
- else, $\forall t \leq M$ the resource is full. The makespan is then obviously optimal.

6.1.3 Minimisation of the Average Lateness

This problem may be solved polynomially with the following algorithm:

Let t be the current date. While one activity is schedulable at t and the resource is free at t do:

1. Select A, the activity with the shortest residual duration.
2. Schedule at t one unit of A
3. Set StartMin of A to $t + 1$
4. Decrease duration of A of 1

Proof

Let's prove that H, the schedule built according to the previous algorithm is optimal for the criterion $|L|$.

Assume H is not optimal. Let O be the optimal schedule for the criterion $|L|$ for which the first date t at which O and H do not coincid, is maximal.

Before t, the same activities are scheduled at the same time on O and H. Let o and h be the activities scheduled at t respectively on O and on H[1]. Let's build then an optimal schedule O' on which the earliest date for which O and H do not coincid is greater than or equal to $t + 1$. The general idea is to intervert the activities o and h on O. Some cases have to be distinguished.

- If the residual duration of o is strictly greater than the residual duration of h, let's then intervert o and h (a new preemption on o may be introduced). Let O' be this new schedule. This is absurd because O' is better than O according to the criterion $|L|$:

$$|L| - |L'| = \textit{ residual duration of } o - \textit{ residual duration of } h < 0$$

[1] If h does not exist and o does, then the algorithm did not schedule an activity which could have been scheduled. Absurd.

If o does not exist and h does, then the hypothesis of maximality on t is refuted because one can obviously build up an optimal schedule on which the earliest date at which it does not coincid with H is greater than or equal to $t + 1$. Absurd.

- The residual duration of h can not be strictly greater than the one of o because of the algorithm. (o would have been chosen instead of h)
- Consequently, the residual duration of o is equal to the residual duration of h. It is then possible to intervert o and h. This is absurd because of the maximality hypothesis on t.

Conclusion: H is optimal.

6.1.4 Minimisation of the Maximal Lateness

The following list algorithm builds up polynomially the minimal schedule H. Let t be the current date. While one activity is schedulable at t and the resource is free at t do:

1. Select A, the activity with the smallest due-date.
2. Schedule at t one unit of A
3. Set StartMin of A to $t+1$
4. Decrease duration of A of 1

Proof

Assume H is not optimal. Let O be the optimal schedule for the criterion L_{max} for which the first date t at which O and H do not coincid is maximal.

Before t, the same activities are scheduled at the same time on O and H. Let o and h be the activities scheduled at t respectively on O and on H[2].

We are now going to build-up O', an optimal schedule on which the earliest date at which O and H do not coincid is $t+1$. Let's intervert o and h on O. Many cases have to be distinguished:

- If $d_h < d_o$.

  Let's build up O' on which h is scheduled at its earliest possible start on the places which were occuped by o and h. o is scheduled on the remaining places.

  On O', the lateness of h is strictly lower than its lateness on O. Moreover, o finishes on O' at the same date than h on O. Consequently, the lateness of o on O' is strictly lower than the lateness of h on O. O' is then at least as good as O. O' is optimal. This can not fit with maximality hypothesis on t. Absurd.

- If $d_h > d_o$.

  Absurd (because of the algorithm).

- If $d_h = d_o$.

  It is then possible to intervert h and o. Then O' is optimal. This can not fit with maximality hypothesis on t. Absurd.

Conclusion: H is optimal.

[2] If h does not exist and o does, then the algorithm did not schedule an activity which could have been scheduled. Absurd.
If o does not exist and h does, then the hypothesis of maximality on t is refuted because one can obviously build up an optimal schedule on which the earliest date at which it does not coincid with H is greater than or equal to $t+1$. Absurd.

6.2 The n Machines Problem

6.2.1 Makespan Minimisation

A Special Instance: no Release-Dates, no Due-Dates

Mac Naughton's algorithm solves polynomially this problem. It is based upon the following proposition:

The quantity $B = \max\left(\frac{\sum_{i=1}^{n} p_i}{m}, \max_i(p_i)\right)$ is a lower-bound of the optimal duration.

Actually, the duration of a schedule is higher than or equal to the duration of one task as well as the sum of the duration of the tasks divided by the number of processors.

The following algorithm is described in [Carlier 88]:

- Sequence all the activities;
- Cut the sequence into m slices of duration B;
- Allocate each slice to one processor;

Let us notice that an activity is interrupted at most once and that in such a case, the two pieces of activities are scheduled at the begining and at the end of the schedule. Hence they never overlap. The makespan of such a schedule is B. It is obviously optimal.

A Special Instance: no Due-Dates

The following list algorithm builds up polynomially the minimal schedule H. Let t be the current date. While one activity is schedulable at t and one resource is free at t do:

1. Select A, the activity with the longuest residual duration.
2. Schedule at t one unit of A
3. Set StartMin of A to $t+1$
4. Decrease duration of A of 1

Proof

Assume H is not optimal. Let O be one of the schedule amongst the optimal schedules for which the first date t at which it does not coincid with H[3] is maximal and for which the number of activities scheduled at t on both H and O is maximal.

Let's consider the both following cases:

- If at t the number of activities which execute on O is strictly lower than the number of activities which execute on H, let then h be an activity executing on H and not on O. It is then possible to execute h on O at t. Let O' be this schedule. O' is optimal and the number of activities scheduled at t on both H and O is strictly greater than the one on O. Absurd.

[3] Two schedules $O1$ and $O2$ coincid at time t if and only if the same activities are scheduled at t on both $O1$ and $O2$.

- If at t the number of activities which execute on H is strictly lower than the number of activities which execute on O: Absurd, because of the construction of H.

Consequently, the same number of activities are scheduled at t on O and H. Let then o and h be two activities scheduled at t respectively on O (and not on H) and on H (and not on O).

Because of the building-up of H the residual duration of o at t is lower than or equal to the residual duration of h. then, there is a date $t' > t$ at which, one unit of h is scheduled and no unit of o is.

Let us then build up O' steming from O on which the unit of o scheduled at t has been interverted with the unit of h scheduled at t' on O[4].

O' is optimal and the number of its activities which coincids at t with H is strictly greater than the one on O. Absurd.

Therefore, H is optimal.

6.2.2 Minimisation of the Average Lateness

We have been unable to find a known polynomial algorithm for that problem. In the following paragraph, we will give some counter example of a very "intuitive" algorithm.

- An extension of the previous one-machine case is the following algorithm: At time t, the shortest unscheduled activity is scheduled.

  example 1

  Let the resource be of capacity 2 and 5 activities A, B, C, D, E:

	A	B	C	D	E
r_i	0	0	0	4	4
p_i	1	3	4	1	1

```
0 1 2 3 4 5 6 7 8
A
B B B
C C C C
        D
        E
```

  Assume $d_i = r_i$, the algorithm produces the following preemptive schedule:

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

  The cost of this schedule is $1 + 3 + 6 + 1 + 1 = 12$

  We are going to see that this schedule is not optimal:

  Let's consider the following schedule:

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

  The cost of this schedule is $1 + 4 + 4 + 1 + 1 = 11 < 12$

  the preemptive schedule is not optimal!

[4] t' is chosen like this because, when interverting the two units of the activities, it is sure that there is no superposition (two units of the same activity scheduled at the same time) at t' on o.

- According to the latest example, let us build a preemptive schedule in which superpositions are allowed.

  Definition: An activity A may be superposed at the time t (i.e., more than one unit of A are executed at t) if A is late (i.e., $r_A < t$) and if no other activity can be scheduled at t.

  In a more formal way, one can say that we create a new class of schedules and that we try to find a dominant set. We can then adapt to this new class of schedule the previously described algorithm. It produces the following preemptive schedule:

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

  The cost of this schedule is $1+3+4+1+1=10$

  Let's consider now the following example:

  example 2

  Let's take a resource of capacity 2 and 9 activities A, B, C, D, E, F, G, H, I

	A	B	C	D	E	F	G	H	I
r_i	0	0	0	3	4	4	5	6	7
p_i	1	3	4	3	1	1	1	1	1

```
0 1 2 3 4 5 6 7 8

A
B B B
C C C C
      D D D
        E
        F
          G
            H
              I
```

  According to our algorithm, the preemptive schedule is:

```
A C C C C G H I
B B B D E F D D
```

  The cost of this schedule is $1+3+5+5+1+2+1+1+1=20$

  Let's consider now the following schedule:

```
C C C C F G H I
A B B B E D D D
```

  The cost of this schedule is $1+4+4+5+1+1+1+1+1=19$

  the preemptive, interruptible schedule is not optimal!

  One shall have accepted to superpose the activity C instead of scheduling D at $t=3$

Starting from the last example, some other (more complex) classes of schedules have been studied, but all the results have been disappointing.

Note that the restricted problem in which all the release-dates are the same, is described in [Garey 79]: [Gonzalez 77] details a polynomial algorithm which solves this special instance.

Bibliography

[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.

[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.

[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.* Submitted to: 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 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.

[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.

[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.

[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.

[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.

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

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