Solving Scheduling problems from High-Level Models

Jean-Noël Monette

Thèse présentée en vue de l'obtention du grade
de Docteur en Sciences de l'Ingénieur

May 2010

ICTEAM, pôle d'Ingéniérie Informatique
École Polytechnique de Louvain
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis committee:

Yves Deville (co-director)	ICTEAM, UCLouvain, Belgium
Pascal Van Hentenryck (co-director)	Brown University, RI, USA
Thomas Stützle	Iridia, ULB, Belgium
Philippe Baptiste	CNRS - LIX, France
Peter Van Roy	ICTEAM, UCLouvain, Belgium
Axel van Lamsweerde (president)	ICTEAM, UCLouvain, Belgium

Abstract of the Thesis

Scheduling consists in deciding when a set of activities must be executed under different constraints, in order to optimize a given objective. The two main types of constraints are precedences between activities, and the availability of finite resources. Common objectives are to minimize the total duration, or to minimize the weighted sum of the tardiness of activities with respect to given due-dates.

Scheduling problems are very varied, both in application domains and in featured constraints. They have been a large area of research for decades. A lot of work has been undertaken to express, classify, and solve scheduling problems. Most of these problems are computationally hard to solve (in the sense of being NP-complete) and need complex algorithms (using techniques in the domains of Operations Research and Artificial Intelligence, such as e.g. Constraint Programming, Local and Heuristic Search, Mathematical Programming). But the difficulty lies also in the modeling of the problems, and the mapping between high-level, declarative models and low-level, procedural search techniques.

The problem we are tackling is the gap between the high-level modeling of scheduling problems and their efficient resolution. This gap has several causes. First, most search techniques deal with more or less pure problems, and may not be easily adapted to solve problems with side constraints. Second, it requires a strong background in operations research to cast the problem to the right representation.

Our goal is to facilitate the work of the user, such that no particular expertise is needed to solve a problem of scheduling. Our contributions in this direction are listed next:

- Strong separation between modeling and solving.
- Structural analysis and classification of problems.
- Synthesis techniques to automatically generate search algorithms for a model.
- Simple combination of several search algorithms, to create loosely coupled hybrid algorithms (in particular using Constraint Programming and Local Search).

As a proof of concept, we developed a system, Aeon, supporting the above contributions. It provides a high-level modeling library for scheduling problems and a set of solvers for such problems. A user can model his/her problem in Aeon, and solve it with different search algorithms, without having to write the search procedure. Experimental results show the time spent to analyze a model and to generate an appropriate algorithm is very low. Furthermore, the generated searches give similar results to algorithms written directly in the Comet Solver, on which Aeon is based.

Additionally, our thesis introduces new results on deduction techniques in Constraint Programming. In particular, we present two propagators of global constraints for scheduling problems. The first one is based on the positions of activities to propagate the disjunctive resource constraints. The second one propagates the minimization of the sum over all activities of the earliness and tardiness costs (a so-called Just-In-Time objective function). Using this second propagator, we were able to improve several best known solutions on a hard benchmark of Just-In-Time Scheduling.

Acknowledgments

First of all, I would like to thank Yves Deville for the 5 years he spent helping me and guiding me. I find him a very good advisor. He has a good balance between letting us free, and guiding us to avoid traps and pitfalls. It is also enjoyable to speak with him about subjects outside our research subjects.

I thank also Pascal Van Hentenryck for his many ideas that really pushed me forward, for his reception during my stays at the Brown University, and all the interest he gave to my research.

Many thanks goes to Pierre Schaus, which has been my office mate for so many years. He is a very nice person to work with. He always has plenty of ideas and of motivation.

I'd like also to give thanks to all the members of the BeCool team over the years. There has been always a good atmosphere in the group, as well as interesting interactions.

I give many thanks to the whole (former) INGI department, in particular people of the third floor, and the volley-ball team. They make working at UCL very pleasant. Pierre Dupont has also been of great help for my work, in particular teaching me to write good papers.

Thanks goes as well to the members of my accompaniment committee, Peter Van Roy and Thomas Stützle, for their very interesting feedback and help. I finally thank the whole jury for accepting to be part of it, and to read my thesis.

I thank all my friends and family for their presence over the years. I'm especially grateful to my parents that raised me as I am, and are always thoughtful for their children.

Last but certainly not least, I have a very particular though for Marie, that is on my side in the everyday life. She has been a very strong support, especially during the writing of the thesis. Let's go on like this, baby!

TABLE OF CONTENTS

1 THE THESIS

Scheduling has been a large research area for decades. A lot of work has been done to express, classify, and solve scheduling problems. Most of these problems are computationally hard to solve (in the sense of being NP-hard) and need complex algorithms. But the difficulty lies also in the modeling of the problems, and the mapping between high-level, declarative models and low-level, procedural search techniques.

In this thesis, we propose a step to fill the gap between high-level modeling and efficient solving. We propose a system that, given a high-level model of a scheduling problem, analyzes its structure, classifies it, and generates appropriate search algorithms. In the remainder of this chapter, we briefly recall what scheduling is, we present the addressed problem, and we highlight the main contributions of our work.

1.1 Scheduling

Scheduling consists in deciding when a set of activities must be executed under different constraints, in order to optimize a given objective. The two main types of constraints are precedences between activities, and the availability of finite resources. Common objectives are to minimize the total duration, or to minimize the weighted sum of the tardiness of activities with respect to given due-dates.

Scheduling problems are very varied, both in application domains and in featured constraints. Some typical applications are manufacture scheduling, construction scheduling, code optimization in compilers, and pharmaceutical project planning. Variations of the problem features may be concerned with the duration of the activities (fixed or variable), the kinds of precedences, the type and number of resources, or the presence of side constraints. More details are given in Section 2.1.

Solution techniques for scheduling include Constraint Programming, Local Search, Mathematical Programming, Genetic Algorithms and many more. Most of these techniques are, however, targeted at specific problems, and they need a lot of development

to adapt them to different problems. Some techniques are more general (e.g., Constraint Programming), but they still need clever heuristics to deal with larger problems.

1.2 Statement of the Problem

The problem we are tackling is the gap between high-level modeling of scheduling problems and their efficient solution. This gap has several causes. First, most search techniques deal with more or less pure problems, and may not be easily adapted to solve problems with side constraints. Second, it requires a strong background in operations research to cast the problem to the right representation.

We can arbitrarily classify solvers for scheduling into two types. First, those that are tailored to a single class of problems and can be used as a black-box once the user's problem is reduced to the specific class. Second, those that can solve several classes of problems. In such systems, it is necessary to model the problem, and often write the search procedure. To apply the first type of solvers, it is necessary to be able to recognize the class of the problem. For the second type, it is necessary to be clever when modeling the problem and writing the search procedure.

Our goal is to facilitate the work of the user, such that no particular expertise is needed to solve a problem of scheduling. Our contributions in this direction are listed next.

1.3 Summary of the Contributions

Our contributions can be described as follows:

1. Design of a modeling layer independent from the search algorithms. The novelty is not in the proposed modeling abstractions, but in the strong separation between modeling and solving.

2. Structural analysis and classification of problems. We propose basic characteristics that can be used to analyze and classify a problem, in order to choose adequate search algorithms.

3. Development of synthesis techniques to automatically generate search algorithms for a model. The search algorithms may include the state-of-the-art for specific classes of problems. This also allows a model to be solved with different search algorithms.

4. Possibility to combine several search algorithms, and to create loosely coupled hybrid algorithms (in particular using Constraint Programming and Local Search).

5. Development of two propagators of global constraints for scheduling problems. The first one is based on the positions of activities to propagate the disjunctive resource constraints. The second one propagates the minimization of the sum over all activities of the earliness and tardiness costs.

As a proof of concept, we develop a system, AEON, supporting the above contributions. It provides a high-level modeling library for scheduling problems and a set of solvers for such problems. The AEON system has the following additional original characteristics:

- The model is transformed to an internal representation which is simplified and analyzed.
- Several search algorithms (called strategies) are associated with classes of problems inside synthesizers. We provide several synthesizers, each one being associated with an underlying technology (e.g., Constraint Programming, Local Search, Greedy Search...).
- Synthesizers are compositional, making it easy to design hybrid algorithms.
- AEON is extensible. Characteristics and classes of problems can be described in XML files using previously defined characteristics and classes. Including new synthesizers and strategies is also facilitated, in particular by the use of views of the problem, that let the programmer focus on his search algorithm.

Our work also includes an experimental evaluation of our prototype. This evaluation shows that the time spent to simplify, analyze, and classify a problem is very low, and increases following a low polynomial curve. For problems with 600 activities, this time is around 6 seconds. We observed also that the current search algorithms under AEON are not able to match state-of-the-art algorithms on heavily studied benchmarks (such as the Job-Shop Problem). However this may be overcome by including those algorithms in AEON. Furthermore, our system is well able to deal with problems with side constraints, which is not possible with those dedicated algorithms.

Global Constraints Here is a short description of the two propagators of global constraints developed in our thesis. The two propagators are integrated in AEON.

- A propagator based on the positions for disjunctive resources. In disjunctive resources, activities must be totally ordered and may be assigned a position. This propagator works by trying all the possible positions for an activity and computing what are its earliest and latest starting time in these positions.
- A global constraint for the Just-In-Time objective, that is the minimization of the sum of earliness and tardiness costs for all the activities. This global constraint uses a relaxation of the problem where resource constraints are removed to get a lower bound to the objective value. The solution of the relaxation is then perturbed to study how the lower bound increases, which permits to reduce the domain of the activities. For disjunctive problems, it is also possible to detect precedence constraints that must hold.

```
1  range jobs; //set of jobs
2  range machines; //set of machines
3  range tasks; //set of activities, one per pair of job and machine
4  int proc[tasks]; //duration of activities
5  int mach[tasks]; //machine required by activities
6  int job[jobs,machines]; //sequence of activities of each job
7
8  Schedule<Mod> s();
9  Activity<Mod> A[i in tasks](s, proc[i], IntToString(i));
10 Job<Mod> J[i in jobs](s, IntToString(i));
11 Machine<Mod> M[i in machines](s, IntToString(i));
12 forall(i in tasks) A[i].requires(M[mach[i]]);
13 forall(i in jobs) J[i].containsInSequence(all(j in machines) A[job[i,j]]);
14 s.minimizeObj(makespanOf(s));
15
16 ScheduleSynthesizer<LS> synth();
17 Solution<Mod> sol = synth.resolve(s);
```

Figure 1.1: Modeling and solving the Job-Shop Problem in AEON (data initialization not shown).

1.4 An example of AEON model

The Figure 1.1 on the next page shows a model (in our prototype, AEON) of a Job-Shop Problem (JSP). The used modeling abstractions are explained in Section 5.3 and Appendix A, but it is straightforward to recognize the usual constraints (machines requirement on line 12, jobs ordering on line 13) and the objective (makespan on line 14) of the JSP. The last two lines create a synthesizer and ask it to solve the modeled problem using a Local Search approach. To use another approach, line 16 must be replaced by another synthesizer. This is all that is needed from the user point of view to solve the problem.

Behind the scene, there is, however, a complex machinery to solve this problem. In this particular example, the synthesizer starts by analyzing the model to discover that (among others) the problem is disjunctive, composed of chains of activities, has no deadlines, and finally that it is an instance of a JSP. It then calls the tabu search procedure of [DT93] for JSPs, and initializes it with the data of the problem. When the search procedure returns, the synthesizer collects the solution and passes it to the user. All this process is described in Chapters 3 and 4.

1.5 Another Perspective

Our work has been carried out with an operational research point of view: problems must be solved as good as possible. It is, however, enlightening to look at it from the perspective of software engineering, and more particularly, languages implementation. Our work can be seen as a semantic-aware interpreter/compiler. The modeling library of AEON is a kind of Domain-Specific Language (DSL), and the synthesizers are in charge of interpreting this language. For an introductory book on languages implementation and DSL, see e.g. [Par09].

In this vision, we can cast the various parts of our work to parts of a compiler or interpreter. Those systems often have an intermediate representation (IR), that represents the input but that can be more easily queried or transformed. In AEON, the internal form of the model plays this role, together with the result of the classification. The parsing of our DSL is made almost nonexistent as our modeling library is regular COMET code, the implementation language. The analysis and classification steps correspond to the semantic analysis, which is far more advanced than in classical language implementations. This is why we call it semantic-aware, as we make strong use of the knowledge of the domain to use appropriate search algorithms.

The strategies correspond in turn to the back-end of an interpreter, where the input is effectively executed. In our prototype, strategies execute a search algorithm, making AEON an interpreter, but it would be simple to transform it to a compiler. Strategies should then produce code that, when executed, runs a search algorithm. This might be also seen as a source-to-source translation from the declarative model to an executable piece of code.

1.6 Outline of the Thesis

The remainder of the thesis is organized as follows. Chapter 2 presents the necessary background and the related work. Notions of scheduling are introduced, followed by an overview of solution techniques, in particular Constraint Programming and Local Search. We then present the programming language COMET, and systems related to ours: Modeling systems, scheduling systems and generic algorithms.

The analysis and classification of problems are then presented in Chapter 3. In this chapter, we present what are the features of scheduling problems we are interested in, how a problem is internally represented, how it is analyzed, and how the classification is carried out.

In Chapter 4, we highlight the synthesis of search algorithms. There, we detail what are the synthesizers, strategies and views. We also present extension mechanisms, and how hybrid algorithms can be composed.

The description of our prototype is done in Chapter 5. It shows how AEON supports our contributions and how it is organized. Then it describes the modeling and solution of problems, and the existing synthesizers.

This is followed by an experimental validation of the system in Chapter 6. We show how our prototype behaves on well-known benchmark problems (Job-Shop Prob-

lems, Resources-Constrained Project Scheduling Problems, Job-Shop Problem with Total Tardiness, Group-Shop Problems, Just-In-Time Job-Shop Problems). We also show that the time spent by the system to analyze and classify a problem is very low.

At last, we present our work in Constraint Programming. The propagator based on positions for disjunctive resources is presented in Chapter 7. The global constraint for the Just-In-Time objective is introduced in Chapter 8. Each propagator is presented with experimental results.

A general conclusion of our work is given in Chapter 9, with perspectives of future research.

This work is accompanied by several appendices. Appendix A presents the documentation of the modeling classes. It is followed by several complete examples of models in Appendix B. Finally, Appendix C describes the different features used for classification.

1.7 Publications

A preliminary version of AEON has been presented at the 11th Informs Computing Society Conference [MDV09a]. The global constraints were presented respectively at CP-AI-OR'07 [MDD07] and ICAPS-2009 [MDV09b]. Here is the list of our major publications:

[MDD07] Jean-Noël Monette, Yves Deville, and Pierre Dupont. A position-based propagator for the open-shop problem. In *CPAIOR07: Proceedings of the 4th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 186–199, Berlin, Heidelberg, 2007. Springer-Verlag.

[MDV09a] Jean-Noël Monette, Yves Deville, and Pascal Van Hentenryck. Aeon: Synthesizing scheduling algorithms from high-level models. *Operations Research and Cyber-Infrastructure, Proceedings of the 11th Informs Computing Society Conference*, pages 43–59, 2009.

[MDV09b] Jean-Noël Monette, Yves Deville, and Pascal Van Hentenryck. Just-in-time scheduling with constraint programming. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *ICAPS*, pages 241-248. AAAI, 2009.

2

BACKGROUND

Scheduling covers a very wide family of problems. Most of these problems are hard to solve and many different techniques have been developed for tackling them. In this chapter, we will review the basic notions of scheduling and some of the solution technologies. The chapter starts by presenting the scheduling notions and problems that are considered in our work. Next, Constraint Programming and Local Search, two well known technologies, are covered in some details; this presentation is followed by a short overview of other optimization techniques. Finally, we present the related work, that is modeling and scheduling systems, and search algorithms.

2.1 Scheduling

Scheduling problems are optimization problems. Optimization problems may be defined in a constraint-oriented way. In this setting, a problem is defined by decision variables, constraints on the decision variables and an objective function defined on the decision variables. The decision variables are the unknowns of the problem that must be fixed. They have a domain that defines the set of values they can take. The constraints define which assignments of the variables are possible and which ones are not. The optimization function decides which assignments are better than others. Decision variables will be denoted by upper case letters, while data of the problems are given with lower case letters.

Scheduling may be defined as the problem of deciding *when* to execute a given set of activities, subject to temporal constraints and resources capacities, in order to optimize some function. The problem is extended to *how* to do it when several alternatives are possible. The main concepts of scheduling are concerned with the activities, the temporal constraints, the resources and the objective. We will review these concepts before presenting some well-known problems and presenting some classifications of the problems.

2.1.1 Notions

Scheduling deals with time, which is a continuous domain. However, in most scheduling applications, the time is discretized, meaning that it is composed of small steps that are supposed to be indivisible. Depending on the application, those steps may be days, hours, seconds, or fractions of a second (among others). In the remainder of this work, we make the assumption that time is discretized. This implies that all quantities related to time are integers and that events (e.g. the start of an activity) may only happen at integer times. In addition, we suppose that the time line starts at zero and is interrupted at some point in the future known as the horizon. Nothing may happen outside those bounds.

Activities The central objects of a scheduling problem are the *activities*, also called tasks. An activity is "something" that must be executed at some point in time. Activities will be most often denoted by the two first letters of the alphabet, upper case and possibly indexed: A, B, A_1, A_2. An activity has a duration (or processing time). This duration may be fixed ($p(A)$) or variable (defined by the bounds $\underline{p}(A)$ and $\overline{p}(A)$). An activity may be preemptive or not. Preemption means that the activity may be interrupted during its processing and restarted later. An activity may also be given a release date before which it cannot start ($r(A)$), and a deadline after which it cannot end ($d(A)$). An activity may be executed following different modes. We call them multi-mode activities. If an activity has several modes, each mode may have a different duration (possibly variable) and define different resource requirements (see below). Finally, an activity may also be optional, meaning that its execution is not mandatory (but this may change the value of the solution).

Several decision variables are associated with each activity. The main decision variable is the time point at which an activity starts ($S(A)$, whose domain is the time interval between the origin and the horizon). In addition, and depending on the variant, additional decision variables may be necessary. If the duration is not fixed, the completion time ($C(A)$, whose domain is the time interval between the origin and the horizon) and/or the duration ($P(A)$, whose domain ranges over the positive integers) are used. If the task is preemptive, it is also necessary to know the set of time steps where the activity is effectively executed (a set variable, whose domain is the set of all the subsets of the interval from the origin to the horizon). Finally, indicative variables are used when the activity has several possible modes ($M(A)$, domain is composed of values denoting the modes) or is optional ($E(A)$, a boolean variable whose domain is composed of the truth values, true and false).

Graphically, activities may be represented by rectangles in a 2D space. The horizontal dimension is the time and the vertical dimension may be used for different purposes that will be detailed later. Figure 2.1 shows a non-preemptive activity with fixed processing time and the associated variables. The square brackets delimit the window of possible execution of the activity.

Precedences Activities are related to other activities using precedence constraints. Those constraints restrict the relative positions of the different activities. In their sim-

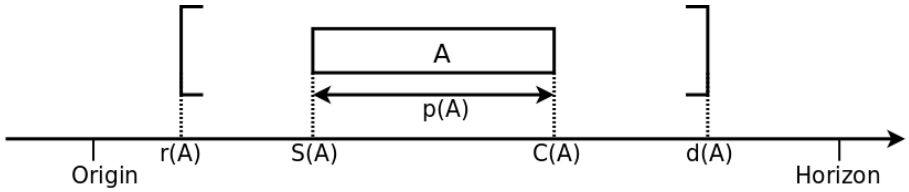


Figure 2.1: Example of an activity A with the associated data.

plest form, they just tell that an activity may only start after another one is finished. That is $C(A) \leq S(B)$. In general, the precedence constraints may also introduce delays between the activities, and relate not only the start of an activity to the end of the other but also the start of the two activities, or the end of the two activities. It is possible also to define a maximum distance between two tasks.

The notion of job appears also in many problems. We define a job as a set of related activities. This general definition covers a large number of cases. In some problems, a job is a sequence of activities that must be executed in order. In other ones, a job is a set of activities that cannot be executed at the same time. Yet in other ones, it is a set of optional activities that must be all executed or all not executed. In some cases, jobs can be ordered through precedence constraints, or the time between their start and end may be limited. In such cases, jobs may be viewed as some kind of super-activities that may be decomposed into normal activities.

Resources Apart from precedences, the other main type of constraints in scheduling problems is related to resources. A resource is something that is required for an activity to be executed. Examples of resources are a crane, a team of workers, fuel, position of a truck. These four examples cover the four main kind of resources that are modeled in scheduling problems. The first one, called machine or unary resource, is a resource that can be used to execute only one activity at the same time. If two activities A and B require the same machine, they induce a so-called disjunctive constraint stating that they cannot overlap in time: $C(A) \leq S(B) \bigvee C(B) \leq S(A)$. It is important to note that this constraint is simply the disjunction of two simple precedence constraints.

The second type of resources is the cumulative (or renewable) resource. This is a resource that can handle several activities at the same time but up to some capacity. Activities require some (integer) amount of the resource and the constraint is that at each time step and for each cumulative resource, the sum of the amount used by the activities that are executing at this step cannot exceed the capacity of the resource. A disjunctive resource is in fact a special case of a cumulative resource whose capacity is set to one.

It is easy to represent a cumulative resource in a graphical way (Figure 2.2). From the representation presented earlier, it suffices to use the vertical dimension for the resource usage. The height of a rectangle is the amount of resource used by the activity. The capacity of the resource is denoted by an horizontal line. The cumulative

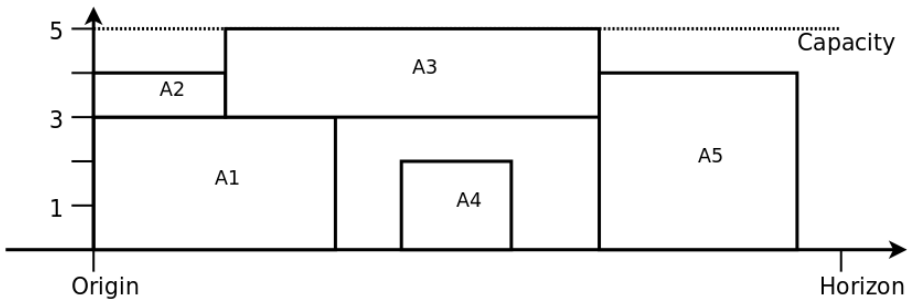


Figure 2.2: Example of a cumulative resource with a capacity of 5.

constraint is graphically seen as placing all the activities below the capacity line.

Another type of resource is called reservoir (or non-renewable resource). A reservoir also has a capacity but, at the difference from renewable resources, activities consume the amount of the resource they needed. Some activities may also produce some amount of the resource. A reservoir comes in fact with an initial capacity, a maximal capacity and a minimal capacity. The constraints state that at any point in time, the current capacity of the reservoir must be between the maximal and minimal capacities. For simplicity, the consumption and production are seen as instantaneous events. The consumption is done at the start of the activity and the production at the end. This is a conservative scheme. More complex evolutions (e.g. linear increase/decrease during the activity processing) exist but are not considered in this work. Hybrid resources also exist that share the features of renewable and non-renewable resource. Some activities then release the amount of resource they used at the end of the execution, while others do not.

The last kind of resources is the state resource. Resources of this type define a set of states in which they may be. Activities may require some state in order to be executed. Two activities that require different states cannot overlap in time, leading to a disjunctive constraint as for the unary resources. There is no constraint on two activities requiring the same state. Note also that the state resource changes its state by itself but that the same state must be kept during the whole execution of an activity requiring this state.

Some additional constraints may be associated to resources. First, cumulative resources and reservoirs can be subject to variations of their capacity over the time, leading to a partial unavailability of the resource during some periods. The variation over the time is called the profile of the resource. At the extreme, the resource is not available at all, which is called a break. Breaks also exist for unary resources. The second special feature are the setup times for unary and state resources. In the case of unary resources, setup times are minimal times between the execution of two successive activities on the resource. If this time depends on the pair of activities that is executed, we call them sequence dependent setup-times. Sometimes, activities have a type, and the setup-times depend on the types of the activities, not the activities

themselves. In the case of state resources, the setup-time is a minimal time between the execution of activities that require different states. Again the setup-time may depend on the states that are required before and after the change.

In the simplest and usual settings, each activity requires some well defined resource. However, alternatives may be possible. An activity may need one of different resources to be executed. Also different modes of a multi-mode activity may require different sets of resources.

Objective There are many different objective functions for scheduling problems. However, most of them are aggregations of simple functions that depend only on one decision variable, the aggregation being a maximum or a weighted sum. For some of the simple functions, it is necessary to define a due-date of an activity ($dd(A)$), which is the time point at which the activity should end. Note that this is a preference, while the deadline is a hard constraint. The simple functions depending on the end of the activity are the completion time ($C(A)$), its lateness ($L(A) = C(A) - dd(A)$), its tardiness ($T(A) = (C(A) - dd(A))^+$), its earliness ($E(A) = (dd(A) - C(A))^+$) and a unit cost ($U(A) = reify(C(A) > dd(A))$). In the previous formula's, $(f(x))^+$ stands for $max(0, f(x))$, and $reify(bf)$ is an indication function whose value is 1 if the boolean formula bf is true and 0 otherwise. Other simple functions associate costs with modes of a multi-mode activity or with the absence of an optional activity.

Some well known objective functions are the minimization of the largest completion time, or makespan ($C_{max} = \max_A C(A)$), the weighted sum of the tardiness ($T_{\sum} = \sum_A t(A) * T(A)$, where $t(A)$ is the tardiness cost per time unit) and the weighted sum of earliness and tardiness ($ET_{\sum} = \sum_A (t(A) * T(A) + e(A) * E(A))$, where $t(A)$ and $e(A)$ are respectively the tardiness and earliness costs per time unit for each activity). Other types of objectives exist but are not covered in this work.

2.1.2 Problem Examples

With the set of abstractions just presented it is possible to represent a large variety of problems. We will review and define some of the well known problems that will be used later. We start with a large review of shop problems, because they have been extensively studied and they illustrate how it is possible to have many variations on a common base.

Shop Problems A prototypical problem is the Job-Shop Problem (JSP) which can be defined as follows. There are N jobs and M machines. Each job is composed of a sequence of M activities, each one having a fixed duration and requiring a given machine. An additional feature that is often cited for JSPs is that all the activities of a job require a different machine. The objective is to minimize the makespan C_{max}. This problem has been studied for a long time, and solved with a large variety of techniques that will be presented later. It also gave rise to a large number of variations. Among them, we can cite:

- Open-Shop Problem (OSP): The order inside each job is arbitrary, but two activities of the same job cannot be executed at the same time. Alternatively, the jobs may be replaced by a second set of machines, and each activity requires two machine, one from each set.

- Flow-Shop Problem (FSP): A JSP where the sequence of the required machines is the same for all the jobs. This is a particular case.

- No-wait Job-Shop Problem: This is a JSP, with the additional constraint that there cannot be any free time between the end of an activity and the start of the next activity of the same job.

- Preemptive Job-Shop Problem: A variation of the JSP where all activities may be interrupted and restarted later.

- Job-Shop Problem with Sequence-Dependent Setup-Times (JSPwST): Machines are subject to setup-times that depend on the order of the activities that are executed.

- Group-Shop Problem (GSP): The order inside each job is only partly fixed. Each job is divided into sub-jobs. The activities inside a sub-job may be executed in any order (but without overlap). The sub-jobs are ordered and the activities inside a sub-job cannot start before all the activities of the preceding sub-job are finished. OSP and JSP are two variants of the GSP, where the sub-jobs are composed respectively of all the activities of a job (no order), and of exactly one activity (total order).

- Flexible Job-Shop Problem (FJSP): In this problem, the machine requirements are changed. Each activity may use any machine among a given set. Different degrees of flexibility exist, depending on how many machines are given as alternatives to each activity.

- Cumulative Job-Shop Problem (CJSP): The unary resources are replaced by cumulative ones. Each activity requires one unit of a resource. Typically all resources have the same capacity, which may be 2 or 3.

- Job-Shop Problem with release dates: Release dates are added to the problem.

Other variations exist that mix the different features. The following problems are JSP where only the objective is changed. They may also be combined with the other variants.

- Job-Shop Problem with Weighted Tardiness (JSPwT): The objective is not to minimize the makespan but the weighted sum of the tardiness of each job or each activity with respect to its due-date. The due-dates are different from job to job. Some problems impose a due-date only on the jobs (that is the last activity of the job), other on all the activities.

- Job-Shop Problem with Weighted Earliness and Tardiness (JSPwET): The objective is to minimize the earliness and the tardiness with respect to some due-date. The due-dates may be different for each activity. Different variations exists, depending on which activities have a due-date: the last of each job, the last for the tardiness and the first for the earliness, or all activities. The last case is also called Just-In-Time Job-Shop Problem (JITJSP).

RCPSP Another type of well studied problem is called the Resource Constrained Project Scheduling Problem (RCPSP) [KS97]. This problem is composed of a set of activities that are partially ordered by precedence constraints. Each activity is non-preemptive, has a fixed duration and needs some amount of a set of cumulative resources. The objective is to minimize the makespan, the weighted lateness of the last activity, or the weighted sum of earliness and tardiness (this variation is often called RCPSPET). This problem is also modified to form new types of problems. The Multi-Mode RCPSP (MMRCPSP) is similar to the RCPSP but each activity has several modes with different durations and resource requests. Additionally reservoirs are introduced and some modes consume some amount of the reservoirs. The RCPSP/-max and MMRCPSP/max are variants of their respective problems, where additional precedence constraints are added that limit the maximal distance between activities. In the case of the MMRCPSP/max, the delay of a precedence arc may depend on the modes of the involved activities.

Other Problems Other classes of problems that have been extensively studied include one-machine problems and parallel machines problems. In the one-machine problem, there is only one machine to process a set of activities. In the parallel machines problem, there are several machines and all the activities may be processed on any of the machines. Many variations are formed by introducing release dates, deadlines, equal processing time, preemption, sequence-dependent setup-times, precedence constraints, and by changing the objective. Some of these problems may be solved in polynomial time.

Yet another problem is the Trolley problem. That problem, described in [Van99], makes use of sequence-dependent setup-times, state resources and reservoirs. It models the problem of a trolley transporting goods between different parts of a factory to be processed. The state resource represents the position of the trolley and the reservoir its capacity. The setup-times represent the time used to move the trolley from a position to another.

A last example of problems is given in the MaScLib [NBF+04] (Manufacture Scheduling Library) that proposes problems of increasing difficulty, starting from academic problems and introducing features to make them look like industrial problems. Even the simplest categories use multi-mode and optional activities, and non-regular objectives.

2.1.3 Analysis of the Problems

In this section, we briefly perform an analysis of the set of scheduling problems. First we will review what are the borders of scheduling, and the limitations that we have put on the scope of problems we study. Next we will show how scheduling problems may be classified, according to different criteria. We will finish with a short complexity analysis of the scheduling problems, together with first steps toward solution procedures.

The Borders of Scheduling Scheduling is a research area that is constantly growing, as researchers explore new problems, introducing new features and requiring new solutions. In this regard, the border between scheduling and planning, between scheduling and timetabling, or between scheduling and vehicle routing may become more and more fuzzy.

In this work, we want to limit ourselves to typical scheduling problems. Figure 2.3 shows the main similarities and differences between scheduling and some other areas of research. As the exact borders are hard to define in a concise way, lets say that we restrict our consideration to problems that are naturally defined using the abstractions described in Section 2.1.1.

Another limitation we have put on our work is that we only consider offline and deterministic problems. There is no uncertainty on the values of the data.

Classification The best known classification of scheduling problems is Graham's $\alpha|\beta|\gamma$ classification [GLLR79]. Established in the 70s it has been extended over the years to take into account new types of problems. The α part describes the resources; β covers the activities, the precedence constraints and the jobs; γ presents the objective function. We will not enter the details of this classification, which is explained in other books (e.g. in [Bru04]).

Additionally, rough classifications give rise to definitions of families of scheduling problems. Problems involving only machines are called disjunctive problems, while a problem involving cumulative resources as well is called cumulative. A problem where all activities have a unique mode are called single-mode problems; otherwise, they are called multi-mode problem. A problem where no activity is preemptive is called non-preemptive and a problem where all activities are preemptive is called preemptive. Problems whose objective value is monotonically non-decreasing with the completion time of all the tasks are said to have a regular objective function.

Complexity Most scheduling problems are NP-complete. However some problems may be solved in polynomial or pseudo-polynomial time. The complexity of many problems remains open. Again we will not review the whole computational studies of scheduling problems but we may cite some useful results:

- Single-mode problems without resource constraints and with a regular or convex objective function may be solved in polynomial time.

Scheduling

- Place activities in time
- Set of activities mainly fixed
- Main question is "When?"

Planning

- Place activities in time
- Set of activities to be determined
- Main question is "What?"

- Boundaries more and more fuzzy: Scheduling adds optional activities, Planning adds duration for activities.

Scheduling

- Place activities in time
- Limited resources (e.g. crane)
- Activities take different times

Timetabling

- Place activities in time
- Limited resources (e.g. rooms)
- Activities take one slot of time

- Side constraints often different

Scheduling

- Place activities in time
- Limited resources (e.g. fuel tank)
- No geographical question

Routing

- Serve activities in time
- Limited resources (e.g. trucks)
- Distances between activities

- Some scheduling problems (e.g. with transition time) may be represented as VRP problems and vice-versa. Different numerical datas (transition time vs. serving time) lead to different algorithms (see e.g. [BPS03]).

Scheduling

- Affect activities to machines
- Minimize the makespan
- Precedence constraints

Bin-Packing

- Affect objects to bins
- Minimize the bin sizes
- Collocation constraints

- Natural mapping of some problems, possible to reuse algorithms from one field in the other (in particular, global constraints).

Figure 2.3: Main common points and differences between scheduling and some other optimization areas

- All the problems presented in the previous section are NP-hard in their general form.

The first result gives an interesting hint on how to solve a large number of scheduling problems. Indeed, this means that if we can get rid of the resource constraints, it is easy to solve the problem. To get rid of such constraints, it is convenient to add precedence constraints between activities such that the resource constraint become satisfied. The problem reduces then from searching in the space of the possible starting times of the activities, to the space of the precedence constraints that may be added. This second space may be much smaller than the original one. This technique is used in many approaches, some of which are detailed later.

The second result means that there are very few chances to find a polynomial algorithm that is able to solve these problems. To solve such problems in reasonable time, it will be necessary to use heuristics, and sometimes be satisfied with non-optimal solutions.

For a more in depth description of scheduling problems, classification and complexity analysis, we refer the interested reader to the three following reference books: [Bru04], [BLN01], [LKA04].

In the next sections, we will review different approaches to solve scheduling problems. We start with the two that we mainly used in our research, Constraint Programming and Local Search. We complete the picture with a short description of other successful techniques.

2.2 Constraint Programming

Constraint Programming (CP) is a paradigm to solve hard combinatorial problems [RBW06]. Its strengths are the modularity, the compositionality and the expressiveness. In CP, problems are described as Constraint Satisfaction Problems (CSP) or Constrained Optimization Problems (COP) . As most scheduling problems are minimization ones, we will focus on the minimization problems.

A COP is a quadruple (V, D, C, O) where V is the set of decision variables, D is a function associating each variable to its set of possible values, called its domain, C is a set of constraints telling which (partial) assignments of variable to values are allowed, and O is a function giving the value of a total assignment of the variables to values. A (partial) assignment is a function associating (some) variables to a value in their domain. A solution is an assignment that satisfies all constraints in C. An optimal solution is a solution that minimizes the value of the objective function O.

An example of COP is the following:

$$minimize\, X + Y$$

$$s.t.\, X = 2 * Z$$

$$X \neq Y$$

$$Y > \max\{X, Z\}$$

$$X, Y, Z \in [1..10]$$

There are three variables X, Y and Z. The first line is the objective, the last one is the domain definition and the three other ones are constraints. An advantage of CP languages with respect to other paradigms, such as SAT or Mathematical Programming, is that many different kinds of constraints may coexist into the same problem. For instance, it is possible to express a problem involving linear equations and first order logic formulas. It is easy to see that scheduling problems may be directly casted into COPs.

2.2.1 Search and Propagation

Constraint Programming solves a problem by exploring a search tree. At each node of the tree, propagation is performed. Propagation consists in removing unfeasible values from the domain of the variable, based on the constraints. Each constraint is treated separately. With a constraint is associated a filtering algorithm (or propagator). This filtering algorithm reasons only on the current domain of the variables and on the semantic of the constraint it implements. The filtering of all the constraints is repeated until a fix-point is reached, that is when all filtering algorithms are run without being able to remove any value. Some filtering algorithms do not remove all infeasible values because it would be too costly to do so. This also means that sometimes several filtering algorithms are associated to the same constraint, each with a different trade-off between its temporal complexity and the amount of values that are removed (called the consistency level). The filtering is also sometimes called pruning.

Three outcomes are possible from the propagation step:

- There is exactly one value left in the domain of each variable. This is a solution.
- The domain of some variable is empty; this means that the subproblem of this node is unfeasible.
- No domain is empty and at least one domain contains more than one value. It is necessary to further explore this node, which is called branching.

Branching consists in dividing the problem defined in a node of the search tree into smaller problems, which are the children of this node. The branching creates a series of problems, such that 1) they all are equal to the original one with the addition of one or more constraint, and 2) the disjunction of those problems is equivalent to the original one.

2.2.2 Global Constraints

There are several definitions of Global Constraints. A classical definition is that they are constraints that span on a set of variables whose cardinality is a parameter. Often they can be decomposed into a conjunction of simpler constraints. The idea of a global constraint is that, by reasoning more globally, it is possible to design filtering

algorithms that either are able to remove more unfeasible values than their decomposition, or are able to perform the same pruning as their decomposition but with a better temporal complexity. The best known global constraint is called *alldiff* and states that a set of variables must all have a different value. Its decomposition is a set of binary difference constraints. Existing filtering algorithms for *alldiff* are able to detect many more inconsistent values than the decomposition [Rég94]. A global constraint is also a way to structure the problem into subproblems that can be solved efficiently. It is then easy to put together different global constraints to solve a more complex problem.

Many global constraints exist to solve scheduling problems. They allow to express the problem in a very structured way, and at the same time to use efficient filtering algorithms. The most studied global constraint in scheduling is the global disjunctive constraint. The global disjunctive constraint is applied on a set of activities and forbid them to be executed at the same time. It is equivalent to a set of binary disjunctive constraints between all pairs of activities in the set, and represents a unary resource in the COP representation. This global constraint corresponds to an NP-complete problem and cannot be solved exactly in a reasonable time. A lot of filtering algorithms have been proposed, such as Edge-Finding [CP89, AC91, CP94, CL94] or Not-First/Not-Last algorithms [CP90, DPPH01, Vil04]. They are explained in more details in Chapter 7, which proposes another filtering algorithm for this global constraint.

Other global constraints for scheduling problems are used to represent other kinds of resources and some structured objectives such as $T_{\sum}$ or $U_{\sum}$. Chapter 8 introduces a global constraint for $ET_{\sum}$, together with a filtering algorithm. Note that the minimization of a maximum (e.g. the makespan C_{max}) does not need a dedicated filtering algorithm, as the normal formulation already performs the maximum possible pruning.

A structure introduced in scheduling, that may be considered as a kind of global constraint, is the precedence graph. It is also called a Simple Temporal Network [DMP89]. The precedence graph is a structure that collects all the precedence constraints of the problem. Using a precedence graph, it is easy to propagate the effect of a new precedence, or to check the temporal consistency of a problem.

With respect to classical COPs, scheduling introduces other structures that are not considered as global constraints but that allow a simpler reasoning. They are global variables, also called structured domains, that group together simpler variables and enforce some constraints on them. Examples of global variables are set variables and graph variables. In scheduling, activities are represented by global variables that are composed of simpler variables describing, for instance, the start time, the end time and the duration. The global variable enforces then that $C(A) = S(A) + P(A)$. Using global variables also allows to declare constraints on these, for instance, that an activity precedes another one.

2.2.3 Search Heuristics

In addition to the filtering algorithms, a key point for CP to solve hard problems lies in the right choice of the search procedure. Choices may be made at different levels that may interact: branching heuristics (form of the search tree), tree exploration strategies,

meta-search strategies.

Branching heuristics are used to choose what are the constraints that are added to create the children of a node. In classical COPs, it often amounts to choose an unassigned variable V and a value v in its domain, and create two children with the added constraints $V = v$ in the first and $V \neq v$ in the second. The choice of the variable and the value are called variable heuristic and value heuristic respectively. A common principle is to choose a variable such that one can detect failures as soon as possible and to choose a value such that it minimizes the chance of failures.

In scheduling, it is rather inefficient to branch on the starting time variables and to try all the values. More complex schemes have been proposed. The first one still works with the starting times but uses dominance rules to avoid useless values. This strategy is called setTimes [LCVG94]. Other branching heuristics are based on a fact mentioned earlier. That is, it is possible to add precedence constraints until all resource constraints may not be violated and find the optimal starting times in polynomial time. The branching consists in choosing the order of the activities that conflict for the same resource. For machines, ranking heuristics have been established. In general, Min-Conflict heuristics may be used [Lab05].

Tree exploration strategies define the way the search tree is visited. A classical strategy is Depth-First Search, but other strategies exists, such as the Limited Discrepancy Search [HG95]. Some strategies are also developed for parallel and distributed computing.

Finally, there are strategies to control the search in a high-level way. We suppose that we are facing an optimization problem. The simplest strategy is called Branch-and-Bound (B&B). In B&B, the search tree is explored once. Each time a solution is found, a constraint is added to the problem, to force subsequent solutions to have a better objective value than the current solution. When the search completes, the last found solution is proved to be optimal. In practice, however, it is necessary to stop the search before its completion, as it may take a very long time. In such cases, the search is incomplete. The last found solution may be suboptimal, and we only have an upper bound on the optimal value of the objective.

Another strategy is to solve successive feasibility problems with the value of the objective constrained to be less than some value. This value is initially set to a lower bound of the optimal value (e.g. zero in many cases). Each time the search completes without solution, the value is incremented. The first solution that is found is optimal. If the search is stopped before any solution is found, we have a lower bound on the optimal value of the objective.

Yet another strategy is to use a dichotomic search, iteratively improving the upper and lower bounds until they are equal.

A more complex scheme is the Large Neighborhood Search (LNS) [Sha98, LG07]. In LNS, the search procedure iteratively solves subproblems based on the solution of the previous subproblem. A subproblem is defined as the original problem with the addition of constraints that fix a part of the variables to the values they had in the previous solution. Alternatively, it can be seen as a two steps approach, where the first step consists in relaxing part of the current solution, and the second step locally reoptimizes the relaxed part. The first solution may be found by a first CP

search without added constraint, or with any other procedure. LNS is not a complete search, meaning that it is not able to prove optimality of a solution. For scheduling, the relaxation for LNS consists often in transforming a solution into a Partial Order Schedule (POS), which is a set of precedences constraints (without assignment of the starting time variables), and removing part of these precedence constraints.

2.3 Local Search

One of the main defects of CP to solve hard problems is that it is designed to be a complete method. However, this makes hard to solve problems of larger size, as the search space is too large to be exhausted. An alternative is to explore only a part of the search space, as is done with Local Search (LS) methods [VM05, HS04].

The idea behind LS is that good quality solutions share many features with other good quality solutions. The idea is then to start from a solution and to try to improve it repeatedly, performing local changes on the solution. This is repeated until some criterion is met. Local Search introduces the notions of neighborhood and move that will be discussed first. Then we will present some neighborhoods used in scheduling, and review some useful metaheuristics.

2.3.1 Principles

In LS, the neighborhood defines the set of solutions that can be produced from a given solution. Most often, the neighborhood is defined through the use of move operators. A move operator describes how to transform a solution into a neighboring solution. For instance, a move operator is to change the value of exactly one variable. In this case, the neighborhood of a solution is the set of solutions that differ from the current solution by the value of exactly one variable. A move is the transformation from a solution to one of its neighbors.

The basic local search algorithm is to repeatedly perform the three following steps: construct the neighborhood, choose a neighbor, apply the move. These three steps need to be as efficient and effective as possible. For this reason, the size of the neighborhood, the choice of the neighbor, and the incrementality of the underlying data structures are critical elements. The choice of the neighbor may be done at random, but most often, one performs informed moves, that is, it evaluates the interest of each neighbor. This can be done by constructing the neighbors and computing the value of the objective function for each of them but this may be costly (if there is a lot of data structure to update). A better approach is to evaluate the cost of the moves without actually applying them. For some objective functions and neighborhoods, it is possible to compute the exact difference (delta) between the values of the current and the neighboring solutions in a small amount of time (constant or sublinear time). For other neighborhoods, it is necessary to approximate the delta to keep a low complexity. Regarding the application of the selected move, it is desirable to have incremental data structures that allow to compute the neighbor with only local modifications of the data structure.

The aforementioned simple algorithm faces a lot of problems. They can be summarized as the need for LS to find the right balance between diversification and intensification of the search. Diversification means to explore many different regions of the search space and avoid cycling. On the contrary, intensification means to explore more in detail promising regions of the search space. Good LS algorithms need to have both diversification and intensification. This is the goal of metaheuristics. Metaheuristics are general strategies developed to guide a LS in order to reach good quality solutions. Prominent metaheuristics are Hill-Climbing, Tabu Search, Simulated Annealing, Variable Neighborhood Search. Metaheuristics applied to scheduling problems are detailed later in this section.

2.3.2 Neighborhoods for Scheduling

Neighborhoods need to be chosen in a problem specific way. Rather than reviewing all the neighborhoods that have been introduced in scheduling, we will explain the principles underlying many neighborhoods.

Many neighborhoods (mainly in disjunctive scheduling) are based on the notion of critical path in the precedence graph. As in CP, the precedence graph collects all the precedence constraints. Some of them are defined by the problem, others have been added by the search procedure. A critical path is a path that determines the value of the objective. For instance, if the problem is to minimize the makespan, a critical path is a longest path in the precedence graph. To improve the value of the solution, it is necessary to modify a critical path [Bal69]. This is done by removing some arcs (that can be removed) and replacing them by other arcs to keep feasibility of a solution, that is to avoid resource constraint violations.

Simple moves consist in inverting the order of two successive activities on a critical path, or to move an activity on a critical path at some other place. Different neighborhoods may be defined, in particular to reduce the number of neighbors that need to be considered. Efficient data structures allow to query or apply the effect of a move very efficiently. More complex moves consist in removing all the precedence constraints that are critical (and that can be removed) and then repair all the resource violations [COS00]. This is done for instance using a greedy search or CP. Note that the underlying principle is the same as in LNS.

2.3.3 Metaheuristics

Metaheuristics are responsible for deciding how to choose the move to apply, in order to guide the search appropriately. The simplest metaheuristic is called Hill-Climbing, or Greedy Local Search. (It might even not be considered as a metaheuristic.) In this search, the move that improves the most the value of the objective is chosen and applied. The search stops when there is no improving move. The algorithm is then said to have reached a local optimum. That is, it is better than all its neighbors. However, it is possibly far from a (globally) optimal solution. For this reason, other metaheuristics have been developed.

Local Search with Restarts consists in repeating several times the same local search but starting from different initial solutions, increasing the diversification of the search.

Tabu Search (TS) [DT93] always chooses the best move, like Hill-Climbing, but it allows to move to degrading solutions. To avoid cycling between a local optimum and its best neighbor, a tabu list is maintained that forbids to perform moves that have been performed recently. The algorithm thus chooses the best move that is not tabu. Many variations exist: tabu lists with random or dynamic length, restarts, aspiration criteria, elite solutions.

Variable Neighborhood Search (VNS) [HMMP10] works with several different neighborhoods. The algorithm starts with one neighborhood and performs hill-climbing to reach a local optimum of this neighborhood. At this point, it uses another neighborhood that hopefully does not define the same local optimum, and performs again a hill-climbing with the new neighborhood. Different variations exist as how to organize the neighborhoods (round-robin, hierarchical...).

Simulated Annealing (SA) [LAL92] works very differently. It chooses a random neighbor. If the move is improving, it is performed anyway. Otherwise, it is performed with some probability. The probability of accepting a bad move decreases with the degradation of the objective incurred by the move. The probability also decreases during the search, until only improving moves are accepted. SA may be very effective but its main problem is to choose accurately the parameters that control the probability distribution.

LNS can also be seen as a LS, where each step is performed by a CP search. The neighborhood is defined by the relaxation of the previous solution, and CP explores this neighborhood in a very effective way. From this point of view, the exploration of the neighborhood may be performed using another technique instead of CP.

We close the current section by introducing Constraint Based Local Search (CBLS) [VM05]. CBLS is a principled way to perform LS, that takes the ideas of CP and applies them on LS. In particular, it introduces compositional constraints and objectives. Those building blocks are used to model the problem, and they are subsequently used in the search by means of a differentiation API.

2.4 Other Optimization Techniques

There exist many other techniques to solve scheduling problems. Many are specific to restricted types of problems, and make use of the specificity of the problems for their correctness and efficiency. This is of course the case for all the polynomially solvable problems such as, (among others), [Bru04]:

- the Job-Shop Problem with two jobs and regular objective is solved by a reduction to a shortest path problem;
- the parallel machines problem with unit processing time, no precedences and minimization of the weighted number of late activities. This problem can be

solved by a greedy algorithm that adds activities to the set of early ones by increasing due-dates, and removing minimal weighted activities from the set.

Other problems that are NP-hard may be solved with the same kinds of techniques. But they only provide some approximation guarantees in the best cases. Scheduling problems may also be solved using Mathematical Programming, in particular Linear Programming and Mixed Integer Programming.

Linear Programming (LP) [Dan98] aims at solving COPs where all the constraints and the objective function are linear equations or inequalities and the variables take their value in the reals. Linear programs are polynomially solvable but the most used algorithm, Simplex, is exponential in the worst case (although it performs very well on average).

Mixed Integer Programming (MIP) [WN99, Wol98] is similar to LP but it adds an integrality constraint on some of the variables. Such variables may only take their value among the integers. Solving a MIP problem is in general NP-hard. Different techniques exist, such as Branch&Bound (somewhat similar to the B&B of CP), Branch&Cut (Cutting planes are added) [PR91] or Branch&Price (B&B with column generation) [BJN$^+$96]. We will not enter of the details of these techniques. The main point is that the relaxation is the key operation to help solving MIPs. A relaxation removes some of the constraints of the original problem. In a linear relaxation, the integrality constraints are dropped, leading to an LP. Other relaxations are Lagrangian relaxations, where some constraints are removed and incorporated in the objective function.

Some applications of Mathematical Programming in scheduling are the following:

- In [SW92], a one-machine problem is solved with a time indexed formulation. In such a formulation, there is a variable for each activity and each time point, indicating whether the activity is executed at this time or not. This formulation avoids the big-M formulation of the disjunctive constraint but induces a very large number of variables.

- Lagrangian relaxation is used in [BFS08] to solve the JITJSP. In this work, they propose two relaxations. The first one is based on a basic MIP formulation of the problem, where they relax the precedence constraints of the jobs. The second one is based on a time-indexed formulation, and the resource constraints are relaxed. In both cases, they are able to decompose the relaxed problem in several independent subproblems. They use a subgradient procedure to optimize the whole relaxed problem. Their experiments show that both approaches yield better lower bounds than Ilog CPLEX 9.1, and that the relaxation of the resources is better when the number of machines is large.

Finally, it is worth saying that a lot of work is done on the hybridization of different techniques, such as LS and CP, LS and MIP or CP and MIP. The hybridization schemes are mainly through collaboration inside a federating paradigm, or through master-slave combinations. More insight about combinations of different solving technologies may be found e.g. in [Hoo06] or [Mil03]. These books highlight also the fact

that hybrid methods often outperform the simple methods they are based on. LNS, presented earlier, is such a hybridization between LS and CP that works very well.

2.5 The COMET Programming Language

As we developed our code in COMET, we devote this Section to a short description of that programming language. As stated by its authors , COMET is "an hybrid optimization system, combining constraint programming, local search, and linear and integer programming. It is also a full object-oriented, garbage collected programming language, featuring some advanced control structures for search and parallel programming" [Dyn09].

COMET has a C++-like syntax. It features simple inheritance, and methods, functions and operators overloading. It can be also used as a scripting language. COMET can interact with C/C++ code in both directions. It contains several modules that can be imported only when needed. It comprises the CP, CBLS, LP and MIP solvers, as well as facilities for visualization, XML Input/Output and databases handling.

With respect to C++, COMET offers some abstractions that facilitate the developer's life (e.g. ranges as arrays indexes, built-in sets, array and set comprehension, conditions and ordering on loops). However, the main strength of COMET lies in the search modules (CP, CBLS, LP and MIP) and the underlying technology that is state-of-the-art. Moreover, the CP and LS modules may be extended with new constraints in a very straightforward way. To illustrate the basics of COMET, we present here two pieces of code to solve the famous 8-queens problem. The left code makes use of CP, while the right one makes use of LS. It is interesting to see the striking similarity between the two models (up to line 9, while the search procedure is of course different).

```
1  import cotfd;
2  Solver<CP> cp();
3  range S = 1..8;
4  var<CP>{int} q[i in S](cp,S);
5  solve<cp> {
6   cp.post(alldifferent(q));
7   cp.post(alldifferent(all(i in S)q[i]+i));
8   cp.post(alldifferent(all(i in S)q[i]−i));
9  }using{
10
11  forall(i in S:!q[i].bound())
12                          by (q[i].getSize())
13    tryall<cp>(v in S:q[i].memberOf(v))
14
15      label(q[i],v);
16
17 }
```

```
import cotls;
Solver<LS> ls();
range S = 1..8;
var{int} q[i in S](ls,S) := i;
ConstraintSystem<LS> c(ls);
c.post(alldifferent(q));
c.post(alldifferent(all(i in S)q[i]+i));
c.post(alldifferent(all(i in S)q[i]−i));
ls.close();
int it = 0;
while(c.violations()>0 && it<50∗n) {
 selectMax(i in S)(c.violations(q[i]))
   selectMin(v in S)
             (c.getAssignDelta(q[i],v))
     q[i] := v;
 it++;
}
```

The above code shows how the modeling and search parts are supported in COMET with the use of well designed abstractions. In particular, the CP search procedure is extremely easy to describe by means of the non-deterministic instruction `tryall`, and the sorted loop (with the `by` keyword). In LS, the selectors (e.g. `selectMax`) make the search procedure code close to a textual description (e.g. "select the queen with the largest violation").

On top of these search modules, COMET offers libraries for scheduling. The library for scheduling in CP introduces global constraints such as Not-First-Not-Last and Edge-Finding, embedded in modeling objects. It provides also scheduling oriented search heuristics. The LS library for scheduling proposes several incremental structures to efficiently evaluate the effect of classical moves for scheduling problems.

2.6 Modeling Systems

We propose a modeling layer disconnected from the underlying search technology. This is also the case of several modeling languages such as ZINC [MNR+08] and ESSENCE [FHJ+08].

ZINC is a modeling language aimed at supporting natural and extensible modeling, and solver-independence. It is declarative and quite high-level. The designers of ZINC chose not to provide modules for specific domains (e.g. scheduling), but rather to propose an extension mechanism of the language. This means that it is possible to represent scheduling problems in a natural way but it will be mapped to low level constraints.

ESSENCE is another formal language for specifying combinatorial problems. It is higher-level than ZINC but does not allow extensibility. It proposes abstraction such as multisets, relations or partitions. However there is not direct support for scheduling.

Both languages make use of a rewriting system to transform the high-level model to a representation usable by a background solver technology (mainly CP, MIP and SAT at this time) [FJMHM05, BDPS07]. It is possible to map a high-level model to different low-level models, sometimes many of them and with different strengths or weaknesses. Mapping to the right model is still an issue.

The main differences between our approach and these systems are the following ones.

- We define a library of modeling abstractions (classes, methods and functions) respecting the syntax of the implementation language, COMET. Our library does not define its own syntax and semantics.
- We specifically target scheduling problems, while ZINC and ESSENCE are general purpose modeling languages.
- As an effect of the previous point, and of the fact that scheduling libraries exist in many solvers (see the next section), our prototype features an internal representation that keeps most of the original structure of the problem, without needing to use a low-level general representation.

- We propose an advanced analysis of the model to detect characteristics and patterns in the problem that can be exploited by the search procedures. This is explained in Chapter 3.

This last point makes AEON close to the system presented in [VM07], where high-level models are analyzed in order to generate an ad-hoc local search procedure. An example of analysis is that if an all-different constraint on the decision variables is tight (as many values as variables), it is better to use a neighborhood swapping the values of two variables, rather than to change the value of only one variable. This system is build on top of COMET, like our prototype, and is in fact the work that started ours. The main differences are that we are restricted to scheduling, but we are not tied to local search.

2.7 Scheduling Systems

It is possible to solve scheduling problems in a general purpose CP or MIP solver. However it does not take advantage of the specificity of scheduling. To overcome this limitation, there exist scheduling modules in such system. Such a module allows to model the problem using the abstractions described earlier in the chapter, and to use the algorithms specifically developed for scheduling (e.g. global constraints, heuristics, temporal networks). Among the existing systems, we will briefly describe OPL, ILOG SCHEDULER and COMET.

OPL [Van99] (standing for Optimization Programming Language) is a language for constrained optimization problems that allows to describe both the models and the search procedures. OPL features a scheduling module to write models in a straightforward way. ILOG SCHEDULER [ILO05] is an extension of ILOG CP SOLVER developed specifically for solving scheduling problems with CP. It features global constraints and specialized branching heuristics. As presented in Section 2.5, COMET [VM05] is a programming language for combinatorial optimization featuring CBLS, CP and MIP solvers. It contains two modules for scheduling, one based on CBLS, the other one based on CP.

AEON is related to those systems in that it features roughly the same set of abstractions. A key difference is that the modeling layer is totally separated from the underlying search techniques. The advantage is that the user does not have to bother with the search part to solve its problem. A downside of our approach however is that it is not possible to mix the scheduling model with external variables and constraints.

2.8 Generic Search Algorithms

To deal efficiently with many problems, an orthogonal approach to the analysis of the model that we propose, is to design a generic and robust algorithm. This is e.g. the case of the Self-Adapting Large Neighborhood Search (SA-LNS) [LG07] , which is a LNS with several different relaxation and reconstruction strategies. The relative use of each strategy and their parameters are continually upgraded during the search

through a kind of on-line learning. Another approach [Ref04] makes use of so-called impacts to drive the search procedure. Impacts are a measure of the importance of variables and of variable-value assignments to reduce the search space. The impacts are learned during search and restarts are used in order to apply the most up-to-date information on the whole search tree. [CJ06] extends the work on impacts, with the use of explanations to detect implied structure in the problem during the search. SA-LNS has been applied on scheduling problems (and only on it), while the two other searches have not.

While the goal of these works is to find a search procedure robust across a variety of models, our objective is to exploit the model structure to derive an effective search procedure for the model at hand. We view these approaches as orthogonal since robust search procedures must also be available for various classes of problems. However, revealing and exploiting the model structure is one of the main contributions of constraint programming, and the search algorithm may significantly benefit from a structural synthesis.

3

ANALYSIS AND CLASSIFICATION

The main goal of the analysis and classification of a problem is to give the synthesizers enough information to generate a good search procedure adapted to the faced problem. To this end, two main tasks are performed. They are a simplification of the model wherever possible, and a retrieval of the characteristics of the problem. The current chapter explains in detail the process of creating an internal representation from a model and analyzing it to retrieve its features. It also shows how the features are described and related to each other.

The first section presents a summary of the features used by the synthesizers and why they are needed. Then, Section 3.2 presents the internal representation, the transformation from the model to that representation and the retrieval of the characteristics. Afterward, Section 3.3 shows how features are structured, and how it is possible to extend the classification using XML files.

3.1 Features of the Problems

The features of a problem are divided into three types: classes, labels and (numeric) values. Classes represent families of problems for which there exist well defined algorithms (e.g. the Job-Shop Problem with weighted sum of tardiness). Labels represent binary characteristics of the problem (e.g. all due-dates are common). Values are other characteristics of the problem that take a numeric value (e.g. the maximum processing time). Such a splitting corresponds to three needs in the generation of an algorithm. The class of the problem decides the strategy that will be used. The labels are used to turn on or off some features of the strategy, and values to adjust some other features.

As detailed in Section 3.3, the features are build on top of each other and on top of characteristics of the problem. These characteristics must be directly retrieved from

the internal representation. For now, we are interested in defining what are those characteristics that we want to retrieve directly, and why they would be interesting. We may divide the characteristics into four main parts, that are related to:

1. activities,
2. precedences,
3. resources and requirements,
4. objective.

The detailed list of features is presented in Appendix C. We present here an overview with their context.

Activities Considering the set of activities in isolation from the rest of the problem, it is necessary to know their number, that mainly determines the size of the problem, and the use of an exact or heuristic algorithm. We also want to know whether they allow preemption, have several modes, are optional, and have a fixed processing time. Most algorithms differ depending on this is the case or not. Additionally, we want to know if all activities have an equal or unit processing time because it is possible to take advantage of this to produce improved search techniques. For instance, with unit processing times, some intractable problems may become polynomially solvable.

Precedences There are several characteristics of interest directly related to the set of precedence and temporal constraints. The first one is to know whether the problem is time-feasible or not. Next, it is interesting to know what kind of precedence constraints appear in the problem. Are they all "simple", meaning that they link the end of an activity with the start of another and don't define delay? Having only "simple" precedences allows to use a simple representation of the precedence graph. Another question is to know whether there are release-dates and deadlines, and if they are equal for all activities. A last characteristic to retrieve is the form of the precedence graph. Indeed, it is useful to know whether it is cycle-free, or composed of chains of activities, rather than a general precedence graph. Often, it is possible to use more efficient data-structures if the precedence graph has some of these features.

Resources and Requirements The resources strongly determine the type of a problem. In particular, "no resource" and "only disjunctive resources" are two characteristics that simplify a lot the resolution of the problem. The presence and number of resources of each kind are also indicators, as are the maximum capacities and the presence of breaks and profiles.

Requirements make the link between activities and resources. We are interested in knowing the characteristics of this link. In particular, do the activities require alternative resources? If this is the case, it is necessary to have a search algorithm that decides which alternative is to be used. Other interesting measures are how many resources are used by each activity, whether all activities need the same resource, and whether the requirements depend on the modes for multi-mode activities.

Objective The objective function defines what the good solutions are. Some characteristics of the objective allow to focus the search on some part of the search space. This is the case, for instance, for regular functions, but also for convex functions. Related to the objective, it is interesting to know if there are due-dates, and if yes, if they are all equal. It is also interesting to know if the objective function contains penalties for modes and for non-execution. Knowing whether the objective is defined on all activities or not and whether it has a maximum or weighted sum form will guide the type of search procedure. Finally, recognizing a particular objective enables to use an associated global constraint (in the case of a Constraint Programming search procedure).

3.2 Internal Representation

The internal representation is used for analysis. This internal representation is designed to ease the analysis process, that is the inspection of all the characteristics defined above. It also allows to get a simple transformation from the model to the representation, and from the representation to the solvers input.

Even with a fixed and limited set of modeling abstractions (see Section 5.3), it is possible to represent a given problem in many different ways. Indeed the modeling library offers some flexibility to propose natural abstractions, and the user may choose among these different abstractions to solve a particular problem. However, we want that these variations lead to the same search procedure. The model is thus transformed to reach the internal form, such that different models of the same problem are mapped to the same internal representation.

To fulfill our goals, the internal form will largely share the abstractions used for the modeling. The transformation as well as the analysis will be greatly facilitated by the conservation of the structure. In the remainder of this section, we present the internal form, the transformation from the model, the analysis that is performed, and we discuss the choices that have been made. This will be done part by part in the same order as in the previous section: activities, precedences, resources and requirements, and objectives.

3.2.1 Activities

The modeling objects allow single-mode and multi-mode activities. For uniformity, the internal form contains only one kind of activities, the multi-mode ones. An activity contains a set of modes, a boolean flag for the preemption and an integer representing the type of the activity. Each mode has a maximal and a minimal processing time. To ensure uniqueness, each mode of an activity must have a different pair of processing times.

A single-mode activity is transformed into a multi-mode activity with just one mode. Optional activities are treated by the addition of a mode with a zero processing time and no resource requirement. Two modes with the same minimum and maximum

processing times are merged into a single mode. The resource requirements of the two modes are combined using a disjunction.

The characteristics related to activities can be retrieved by inspection of all the activities. We are not only interested in knowing whether the activities uniformly share a characteristic (e.g. preemptive or optional), but also whether a large part of them share this characteristic. Indeed, even if the problem is not pure, it is maybe close enough to a pure problem to reuse the knowledge of the pure problem in the impure one.

3.2.2 Precedences

The internal representation of the precedence constraints naturally features a precedence graph. In this graph, the nodes correspond to the starts and ends of the activities, and the arcs represent precedences between the nodes. They are labeled with a value. The meaning of an arc (i, j) whose value is d is that the moment at which j occurs must appear at least d units of time after the moment at which i occurs. The labels of the arcs may be negative, so that there may appear cycles (of non-positive length[1]) in the precedence graph.

Jobs are available at the modeling level, but they are not present anymore in the internal form. Their goal is to ease modeling but all the constraints expressed on a job may be replaced by constraints on the activities of the job. The sequence of activities in a job is replaced by a chain of precedence constraints. The non-overlapping constraint of a job is handled by the addition of a disjunctive resource that is required by all the activities of the job. Objectives defined on the completion time of a job are replaced by a maximum over the completion time of all the activities of the job. To deal with the precedences that are posted on the jobs, the start and end of the jobs are introduced in the precedence graph, with their respective precedence arcs.

The precedence graph also contains one node to represent the origin of the time, in order to represent the release dates and deadlines of the activities with arcs.

Graph construction The precedence graph is constructed from the start and end of every activities. Nodes are added to represent the end and the start of jobs, the end and the start of the schedule, and the origin of the time axis (called "origin"). The arcs linking activities, jobs and schedule are added for different reasons. The first one is the most natural, it consists of the precedences explicitly added by the user between activities. Other arcs are added for the (min- and max-) processing time of the activities, for the inclusion in jobs and in the schedule. There are also arcs related to the order of activities inside a job, and to the possible maximum distance between the start and the end of a job (maximum slack).

Finally, arcs are added because of release dates and deadline definitions. These arcs are linked to the "origin" node. If an activity A_i has release date r_i and deadline d_i, two arcs are added. The first is from the "origin" to the start of the activity and has length r_i. It corresponds to the equation $0 + r_i \leq S(A_i)$, where 0 is the time at

[1] Otherwise the problem is not feasible.

which the "origin" node occurs. Regarding the deadline, it adds an arc from the end of the activity to the "origin" node with a length of $-d_i$. This enforce the constraint $C(A_i) - d_i \leq 0$, which is equivalent to $C(A_i) \leq d_i$, the definition of the deadline.

There are many different ways to declare equivalent precedence graphs. By equivalent, we mean that the temporal constraints define the same solutions. For this reason we compute the transitive closure which is a canonical representation of equivalent graphs. In addition, we compute a reduced graph which is comparable to the transitive reduction of the precedence graph (while not being a strict reduction). Those two representations give a lot of useful and complementary information. The difficulty however is that the transitive reduction of a graph may not be unique. The remaining of this section presents how we deal with this problem to ensure a reduction that best suits our needs.

Let's first introduce some definitions of the concepts.

A *precedence graph* $G = (N, A, L)$ is a directed graph with a set of nodes N, a set of arcs $A \subseteq N \times N$, and a function $L : A \rightarrow \mathbb{Z}$ that associates an integer length with each arc in A.

The *transitive closure* $C(G)$ of a precedence graph $G = (N, A, L)$ is the graph $C(G) = G' = (N, A', L')$ such that there is an arc $a = (n, m)$ in A' whenever there is a path from n to m in G and $L'(a)$ is equal to the length of the longest path from n to m in G.

The previous definition requires that G does not contain cycles of positive length. This is the case whenever the precedence graph represents a feasible problem. We apply the algorithm of Floyd-Warshall to construct the transitive closure of the graph. This algorithm is able to detect cycles of positive length. Its temporal complexity is $\mathcal{O}(n^3)$.

A *transitive reduction* of a precedence graph $G = (N, A, L)$ is a graph G'' such that $C(G) = C(G'')$ and there does not exist a graph with fewer arcs that has the same transitive closure.

When there exist cycles of length zero, the transitive reduction of a graph is not unique as shown in Figure 3.1. However, it is possible to design a procedure that computes deterministically a given reduction. This problem has been solved by Aho et al. in [AGU72]. In their work, the arcs had no length and non-uniqueness was due to any cycle. In our setting, we are only interested in cycles of length zero (zero-cycles) but it is possible to largely reuse their results.

A *zero-cycle* is a set of nodes Z such that for all $n_1 \in Z$ and $n_2 \in Z$, the sum of the longest path from n_1 to n_2 and from n_2 to n_1 is equal to zero. It is a maximal zero-cycle if it is not possible to add a node such that the set still respects the condition. We have the properties that the transitive closure of a zero-cycle is a complete graph (a clique), and its transitive reduction is a (zero-length) cycle going through all the nodes of the set.

An *almost transitive reduction* of a precedence graph $G = (N, A, L)$ is a graph G'' such that $C(G) = C(G'')$ and there exists a transitive reduction of G that has the same arcs as G'', except inside the zero-cycles. The interest of this relaxed definition is for analysis purposes and will be made clear later.

We now introduce an algorithm that takes a precedence graph and returns an al-

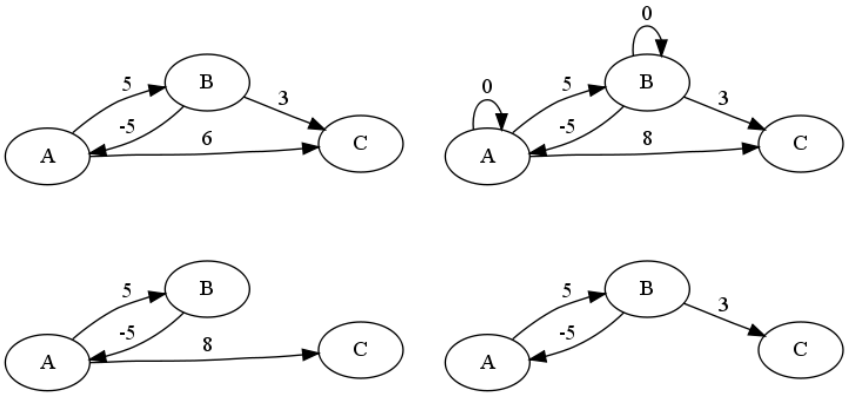


Figure 3.1: Example of a graph having two different transitive reductions. The original graph is shown in the upper left corner and its closure in the upper right corner. The two lower corners show two possible reductions.

most transitive reduction of this graph. The main idea of the algorithm is to contract zero-cycles in order to reach a graph for which the transitive reduction is unique. After having computed this reduction, there remains to expand the cycles to get back the original set of nodes. The expansion of those zero-cycles may be done in several ways, which lead to the non-uniqueness of the (almost) transitive reduction. Indeed we must choose

1. which arcs of the clique[2] are kept (called internal arcs).
2. which arcs linking the clique to the remaining of the graph are kept (called external arcs).

In [AGU72], they solve this problem by fixing an arbitrary order on the nodes and expand the clique by a single cycle where nodes are visited in increasing order, and the external arcs are linked to the node of the clique coming first in the ordering. In the case of precedence graphs, there is a meaning associated to the arcs. We can take advantage of this fact to fix an expansion that is unique, while not depending on an arbitrary ordering.

As previously said, the cycles form cliques in the transitive closure. The zero-cycles lead to cliques where each node is linked to itself by a loop of length zero (see upper right corner of Figure 3.1) and where each pair of nodes is linked by two arcs whose sum of the lengths is zero. The meaning of such a pair of arcs is that fixing the instant of one of the nodes uniquely determine the instant of the other. As an effect, we know the exact ordering of the nodes of the clique in any schedule. We thus have a natural ordering for the expansion of the clique. The only exception is for nodes that

[2]Remember that a zero-cycle is a clique in the transitive closure

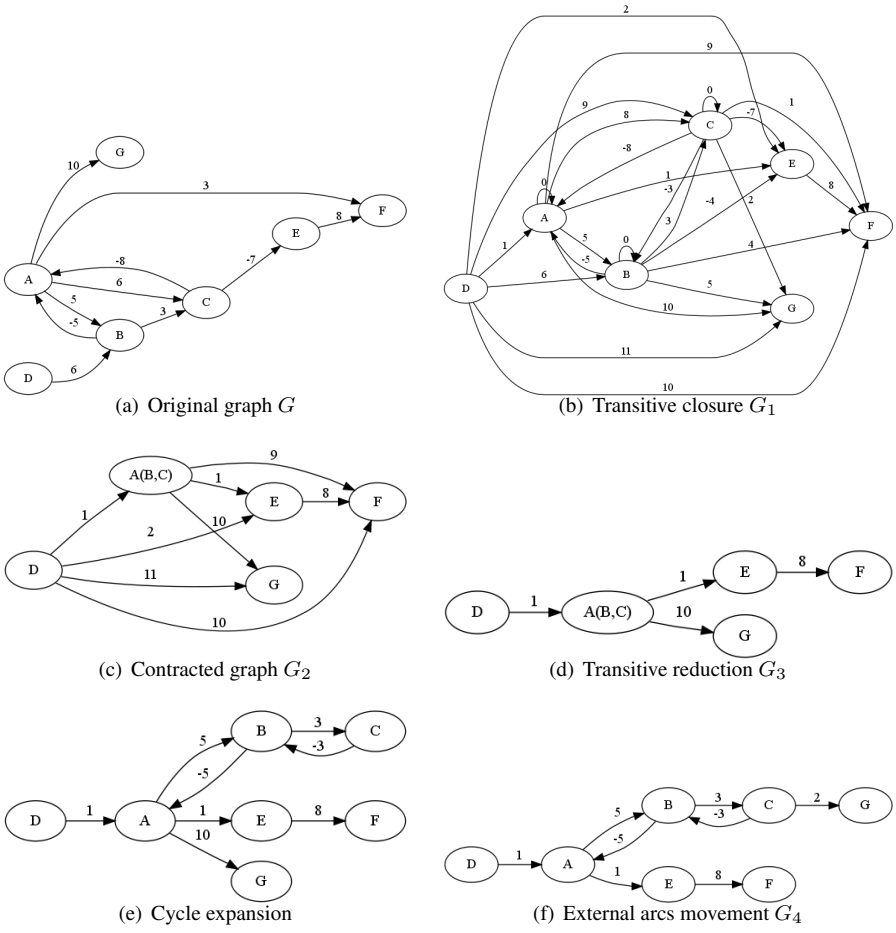
must occur at the very same time, i.e. linked by a pair of arcs of length zero. To avoid such cases, we merge such nodes and replace them by a single node.

Regarding the external arcs, it would be possible to keep only those that are incident to the first node of the clique but this may lead to rather unnatural structures. For each node linked to the clique, we only keep the arc whose length is non-negative and minimal. If such an arc does not exist, we keep the arc whose length is minimal (closest to the negative infinite). The rationals behind this choice are that it is simpler to reason on a problem only with arcs of positive length, and that it is even simpler to reason about the precedences when there is no delay, or at least with only short delays. This can be seen for instance in Figure 3.1 where the reduction of the lower right corner looks more natural than the one of the lower left corner. In particular, if nodes A and B are respectively the start and end of an activity, it is more natural to state that C may happen no less than 3 time units after the end of the activity, rather than 8 time units after the start of the activity.

The entire procedure to come up with the internal form of the precedence graph is the following :

1. Construct the precedence graph $G = (N, A, L)$ from the model.
2. Compute the transitive closure $G_1 = (N_1, A_1, L_1)$ of G.
3. If there are cycles of positive length, the problem is unfeasible; stop.
4. Merge nodes that must occur at the same time (that is nodes n_i, n_j such that $L_1((n_i, n_j)) = L_1((n_j, n_i)) = 0$).
5. Compute $G_2 = (N_2, A_2, L_2)$, the graph resulting from the contraction of the zero-cycles of G_1.
6. Compute the transitive reduction G_3 of G_2 (removing arcs (n_i, n_j) such that there is a longest path in G_2 from n_i to n_j that does not include (n_i, n_j)).
7. Expand the zero-cycles of G_3, to produce G_4. For each zero-cycle:
   (a) Order the nodes of the zero-cycle such that $n_i < n_j \Leftrightarrow L_1((n_i, n_j)) > 0$. This is a total order.
   (b) Add two opposite arcs between each node and its successor in the order.
   (c) Move the external arcs to their best position (such that the length is minimal, and positive if possible).

It is important to note that the resulting graph (G_4) is not a real transitive reduction but an *almost* transitive reduction according to our definitions because of step 7(b). Indeed the clique (with $k > 1$ nodes) is replaced by $(k - 1)$ pair of arcs, that is $2 * (k - 1)$ arcs. To have a transitive reduction, it would be sufficient to create a cycle with all nodes, that is k arcs. We however prefer this representation that more clearly shows that two successive nodes are clamped together, something that requires to travel all the cycle in a real transitive reduction. This is very useful for the analysis. Figure 3.2 illustrates the execution of the algorithm on a small precedence graph. It must be read line by line, from left to right. The entire procedure has a temporal complexity in $\mathcal{O}(n^3)$, due to the steps 2 and 6 that are based on the Floyd-Warshall algorithm.



(a) Original graph G (b) Transitive closure G_1

(c) Contracted graph G_2 (d) Transitive reduction G_3

(e) Cycle expansion (f) External arcs movement G_4

Figure 3.2: Example of execution of the almost transitive reduction algorithm. The node marked "A(B,C)" is the node corresponding to the contraction of A, B and C, where A is the representative in the contracted graph.

Analysis Using the transitive closure and almost reduction of the precedence graph, it is easy to answer questions about precedences. Here are the main questions we want to answer and how they are solved.

- Are there delays different from zero between activities? This question is answered by inspection of the almost transitive reduction and looking for arcs of non-zero length between two nodes that correspond to different activities.
- Are there no-wait between some/all activities? This question is answered by inspection of the almost transitive reduction and looking for pairs of arcs of opposite length between two nodes that correspond to different activities.
- What is the form of the precedence graph? A lot of algorithms take advantage of special forms of the precedence graph and it is important to recognize them. The different forms that we are looking for are the following : chain, set of chains, in-tree, set of in-trees, out-tree, set of out-trees, acyclic graph, cycle, set of cycles, general graph.
  To perform this analysis, we first remove from the almost transitive reduction graph the special nodes (start, end and origin of the schedule and of the jobs) and the arcs corresponding to maximal processing time of activities. We then compute the weakly connected components of the result and look for patterns in those components. Those patterns are discovered through inspection of the degrees of the arcs. For instance, all the out-degrees of the nodes of an in-tree are smaller or equal to one. This condition and the acyclicity are sufficient to characterize an in-tree. A similar reasoning is performed for the other forms.
- Do activities have release-dates and/or deadlines, and are they common to all activities? This is easily discovered by looking at the arcs going to or coming from the origin node in the almost transitive reduction graph.

For the simplification and analysis processes, the transitive closure and reduction have complementary interests. From the closure, it is possible to get in constant time the information about the relative position of any two nodes. From the reduction, we compute the aggregated information presented right above.

It is worthy of note that the precedences in the almost transitive reduction are enough to define the problem. As there are possibly less arcs than in the original problem description, this may allow further gains in efficiency if the search algorithm is dependent on the number of precedence arcs.

3.2.3 Resources and Requirements

Resources The resources can be represented by two general types : reservoirs and state-resources. A cumulative resource is a particular case of a reservoir and a disjunctive resource can be viewed as a particular case of a cumulative resource or of a state-resource. We choose here to view it as both a cumulative resource (and hence a reservoir) and a state-resource. This duality allows to put the information about the disjunctive resource at the best place.

A reservoir is defined by its minimum, maximum and initial capacities, as well as by a profile of the maximum capacity. This profile is a sequence of pairs (a, b) where

b is the maximum capacity of the resource from the instant a until the instant of the next pair (or the end of the schedule). A State-Resource has a set of possible states and a transition matrix between those states.

There is a close matching between the modeling objects and their internal counterparts. The profile of a resource can be computed from the profile components and the (periodic) breaks. The only special case is the one of the machines. They are represented by a pair of a reservoir and a state-resource. The reservoir has a maximal capacity set to one. If there is no transition time defined, the state resource can contain only one state. Else, there is one state per type of activity and the transition time matrix is defined accordingly.

Note also that in the internal form of the resources, the distinction between a reservoir and a cumulative resource is lost. It is necessary to look at the requirements to see the difference.

Requirements In the internal representation, the requirements of the activities are stored independently of the activities and of the resources. A third structure collects all the requirements. It is a collection of requirement trees, each one being associated with exactly one mode of one activity. Each mode of each activity is associated to exactly one tree, possibly empty.

The requirement tree is a three levels tree. The root node represents a conjunction of its children requirements, the second level is composed of nodes representing the disjunction of their children. The last level, composed of leaves, represents the individual requirements for each resource. The leaves contain each a triple (r, t, d), where r is the required resource, t is the type of demand and d is the amount for this resource. The value of t may be "state", "consumption" or "utilization". The production and supply are represented by negative values for the types "consumption" and "utilization" respectively. Each leave also contains an identifier to keep track of which requirement it originally represents.

The requirement tree is similar to a Conjunctive Normal Form (CNF), that is a conjunction of disjunction of literals. Although this representation may lead to an exponential explosion of the formula, it is convenient to have such a normal form to reason about. We expect that there are few practical cases where such an explosion would occur and lead to computational problems. We chose the CNF rather than a disjunctive normal form (DNF) for two reasons. The first one is that most often requirements have a form which is closer to a CNF than to a DNF. The second reason is that disjunctions are hard to tackle for many combinatorial optimization procedures (in particular CP and MIP), and it seems preferable to have disjunctions on smaller parts of the problem.

In the model, the requirements are already in the form of trees. The main task to get the internal form is to transform them to the CNF. First, the trees are moved from the single-mode activities to their unique mode, empty trees are created for optional modes and trees of merged modes (because of same processing time) are linked by a disjunctive operator.

The normalization of the tree is done by simple rules of the boolean algebra (ex-

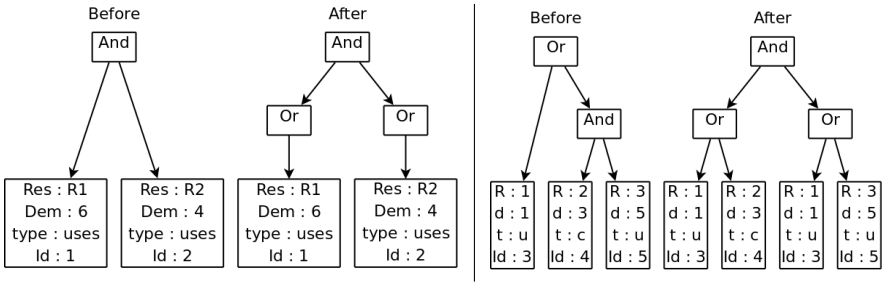


Figure 3.3: Two requirement trees before and after normalization.

cept that there is no negative form). In particular, a disjunction of disjunctions can be replaced by a single disjunction, a conjunction of conjunctions can be replaced by a large conjunction, a disjunction of equal formulas can be replaced by a single occurrence of the formula, and a disjunction can be distributed over a conjunction. Through repetitive application of these rules, we obtain a normal form.

Figure 3.3 shows two examples of requirement trees before and after normalization. In the first example, disjunctions with one term are introduced to comply to the normal form. In the second example, the disjunction is distributed over the conjunction. Note in the second example that two leaves have the same identifier because they represent the same requirement. This is necessary to keep the original meaning intended by the user.

After the requirement simplification, some resources can be further transformed. First, resources that are not requested at all may be removed from the set of resources. Second, resources that induce disjunctive constraints are transformed into disjunctive resources. This is the case of cumulative resources for which all demands are larger than half the maximum capacity, and of state resources for which each activity requires a different state. Third, two machines for which it is certain (from the precedence constraints) that the activities that request them cannot overlap in time can be merged into one single machine. This is the case for instance if all the activities that require the first machine have a deadline which is smaller than the smallest release date of the activities that require the second machine. We perform the search for such pairs of resources in an exhaustive manner, leading to an algorithm in $\mathcal{O}(n^2)$, where n is the number of machines.

Analysis The following questions are representative of how we characterize resources and requirements.

- What is the number of resources? Are there any resources?
- What are the kinds of resources?
- What are the capacities?
- Are there some non-constant profiles? Are there only breaks?
- What are the largest and smallest numbers of states?
- Are there sequence dependent setup times different from zero?

- Are the requirements only conjunctive or only alternatives?
- Is there production or consumption of resources?

All those questions are answered almost by direct lookup into the structures defined above: the set of resources, and the set of requirement trees. To answer some of the questions, it is also necessary to know the set of requirements associated to a particular resource, so we maintain this information as well.

3.2.4 Objective

The objective function is represented, as for the model, by a rooted tree. The leaves of the tree correspond to basic functions depending on the completion time or on the mode of a single activity. A weight is associated to each leave. This weight is required to be positive. The internal nodes aggregate their children, representing the sum or the max operators.

The mapping between the model and the internal form is direct. The internal tree is created by a Depth-First Search through the tree representing the objective in the model. Basic simplifications are performed. First, all multiplicative factors are moved down to the leaves. The sum of sums are simplified to a single large sum. The same happens for maximum of maximums. Second, functions that are always equal to zero are removed. This is the case of functions with a null weight, and of absence costs for activities that are not optional. Third, the sum or the maximum over one function is replaced by the function. Other simplifications should take place but are not yet implemented. For instance, in a maximum, it is possible to remove functions that are dominated by other ones. Likewise, in a sum, we can remove two functions that always sum to zero. Also, it should be necessary to replace a Tardiness function with respect to the origin by a Completion Time function.

Unlike for resource trees where conjunction and disjunction can distribute one over the other, for objectives we cannot distribute the maximum over the sum, although the opposite is possible. However, although having a normal form for the objective (a maximum of sums of weighted basic objectives) would be interesting from an analysis point of view, we do not use it to avoid an exponential explosion of the formula. Such an explosion is more likely to occur than for the requirements. Imagine for instance the sum over n activities of the maximum between the earliness and tardiness (a classical objective). If we put the formula into a maximum of sums, we end up with a maximum over 2^n sums of n terms.

The retrieval of the information is performed through depth-first exploration of the tree. The object of the analysis is to check if the formula has some properties such as regularity (monotonic increase with completion times), convexity, unique due-date. It also looks at the kind of basic functions that are presents and if the objective falls in some known patterns (e.g. Makespan, minimization of the weighted sum of tardiness).

<feature>	::=	<class-or-label> | <numeric-value> .
<numeric-value>	::=	<numeric-function> |
		<numeric-value> <binop> <numeric-value> |
		<numeric-value> <binop> <numeric-constant> .
<class-or-label>	::=	<predicate> | $\neg$ <class-or-label> |
		$\bigwedge_i$ <class-or-label>$_i$ | $\bigvee_i$ <class-or-label>$_i$.
<predicate>	::=	<boolean-function> <comp> <boolean-constant> |
		<numeric-value> <comp+> <numeric-value> |
		<numeric-value> <comp+> <numeric-constant> |
		<string-function> <comp> <string-function> |
		<string-function> <comp> <string-constant> .
<binop>	::=	$+$ | $-$ | $*$ | $/$.
<comp>	::=	$=$ | $\neq$.
<comp+>	::=	$=$ | $\neq$ | $\leq$ | $<$ | $\geq$ | $>$.
<boolean-constant>	::=	true | false .
<numeric-constant>	::=	1 | 2.34 | -12.67 | ...
<string-constant>	::=	"some" | "example" | ...
<boolean-function>	::=	ConvexObjective | ...
<numeric-function>	::=	NumberOfResources | ...
<string-function>	::=	FormOfThePrecedences | ...

Figure 3.4: BNF-like notation for the definition of features.

3.3 Classification

The classification of a problem outputs a set of features in a `Classification` object, from the analysis of the internal form presented in the previous section. The current section shows how the process of classification is achieved. It starts by presenting how the features are described.

3.3.1 Language for Features Description

The analysis of the characteristics presented in the previous section may be seen as a set of functions. They return different kinds of values: numbers, booleans and strings. Features are built on top of these characteristics and of each other.

As shortly described in Section 3.1, the features are of three sorts: classes, labels and numeric values. Classes and labels may be seen as boolean variables, and numeric values as real variables. All those variables are instantiated for each problem. A class or a label evaluated to true means that the problem belongs to the class or exhibits the label.

To define the features, we introduce an abstract language. We identify features to sentences of this language. Figure 3.4 presents a grammar for the definition of the language in a BNF syntax. In this grammar, no distinction is made between classes and labels. The distinction is only in their use in the synthesis process.

An example of feature is the *disjunctive* label which corresponds to the following sentence: *nbStatesResources = 0* $\wedge$ *reservoirConsumption = false* $\wedge$ *reservoirProduction = false* $\wedge$ *maxMaxCapacity = 1*. This means that there is no state resource, nor consumption/production of resources, and that the largest capacity among the resources is one. As another example, the *proportion of preemptive activities* is a numeric value defined as being the quotient of the number of preemptive activities and the total number of activities, that is (in terms of the BNF syntax) *nbPreemptiveActivities* / *nbActivities*. All the features are described in Appendix C.

In general, numeric values (<numeric-value> in Figure 3.4) may be numeric characteristics functions of the problem, or the product of simple arithmetic operations involving one or two numeric values. The defined operations are the four classical binary operations ($+$, $-$, $*$ and $/$) involving numeric characteristics, other numeric values and constants.

Classes and labels are build as boolean formulas whose predicates involve characteristics and numeric values (see <class-or-label> in Figure 3.4). A formula may be a predicate, the negation, the conjunction and the disjunction of formulas. The predicates take one of the following forms:

- Testing if a string characteristic is equal or different to a given string or to another characteristic.
- Testing if a boolean characteristic is true or false.
- Comparing a numeric value to a constant or to another numeric value. The comparison sign may be one of the following ones: $=$, $\neq$, $\leq$, $<$, $\geq$, and $>$.

3.3.2 Structure of the set of features

The language for feature description presented in Section 3.3.1 is abstract because its syntax never appears in the system. Rather, features are implemented into a set of objects (see Section 3.3.4). However, it is convenient to reason about features in terms of this language.

In particular, the set of classes and labels form a partial order. We say that a formula (or sentence) F is more specific than another formula F' if F' appears as a subformula of F. Most of the time, classes and labels are declared as conjunction of subformulas. If F is a conjunctive formula $F = F' \wedge F''$, we say that F is a *specialization* of (or specializes) F' (and F''). If F and F' are classes, we say that F is a subclass of F', meaning that all the problems that are member of F are also members of F'. As any partial order, this *specialization* relation is transitive. This also means that all the classes are subclasses of an hypothetical class containing all the problems that can be modeled using the set of modeling abstractions and that would correspond to the constant boolean formula *true*, or to an empty conjunction. We denote this class ϕ.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE Features SYSTEM "features.dtd">
3  <Features>
4  <Value ID="%PreemptiveActivities"
5          V1="nbPreemptiveActivities" Op="/" V2="nbActivities"/>
6  <Label ID="Preemption">
7    <Extends ID="SomePreemption"/>
8    <Predicate V1="%PreemptiveActivities" Op="=" V2="1.0"/>
9  </Label>
10 <Label ID="NoPreemption">
11   <Extends ID="SomePreemption"/>
12   <Predicate V1="%PreemptiveActivities" Op="=" V2="0.0"/>
13 </Label>
14 <Label ID="SomePreemption">
15   <Not> <IsA ID="Preemption"/> </Not>
16   <Not> <IsA ID="NoPreemption"/> </Not>
17 </Label>
18 </Features>
```

Figure 3.5: Example of XML description for the preemption

3.3.3 Feature description through XML

The description of features in AEON is done through XML files. We chose that way to allow a user to easily add new features that could be used in synthesizers. Another advantage is that XML defines tree-like structures, which is exactly the case of the formulas defining the features.

An XML feature description file consists of one or more features. Each feature (Class, Label, Value) is defined by a unique name (the name of the feature it describes). The classes and labels contain the definition of the formula of the feature (based on predicates, other features ("IsA" keyword) and composition operators). The top-level operator of the definition of the formula is a conjunction. The values define numeric values based on other numeric values and functions. In the current version, it is only possible to define a value using one binary operator, meaning that all the partial products have to be named.

The system is able to detect relations of *specialization* that result from the syntactic definition of classes and labels in simple cases ("IsA" keywords on the top-level of the definition of a class or label). Other relations of specialization inferred from the semantics must be explicitly added by the user. The keyword "Extends" is used to add such relations.

Figure 3.5 presents an example of XML file. It first describes the ratio of preemptive activities (a numeric value, defined in line 4). Then we define three labels that are respectively for problems with no preemptive activities (lines 5-8), all preemptive activities (lines 9-12), and some preemptive activities (lines 13-18). Note that the last

```
1  <?xml version="1.0" encoding="UTF−8"?>
2  <!DOCTYPE Class SYSTEM "features.dtd">
3  <Class ID="PCmax">
4      <IsA ID="Disjunctive"/>
5      <IsA ID="Makespan"/>
6      <IsA ID="SimpleTemporalConstraints"/>
7      <IsA ID="NoPreemption"/>
8      <IsA ID="MaxAlternatives"/>
9      <Predicate V1="graphForm" Op="=" V2="empty"/>
10 </Class>
```

Figure 3.6: Example of XML description for the $P||Cmax$ problem

case is defined as being neither of the two other cases. But on the opposite, preemption and no-preemption specialize the some preemption case. Indeed in general, a strategy that could solve a problem with some preemptive activities can solve a problem with all activities being preemptive, or all activities being not preemptive.

The main asset of the XML description language is the reuse of previously defined features to describe a new feature, avoiding to write the whole formula. A good example is provided in Figure 3.6 that shows the XML description of a parallel machines problem with makespan minimization ($P||Cmax$ in Graham's notation). It reuses several features defined in other files and just adds two predicates.

The whole range of analysis functions and features is described in details in Appendix C.

3.3.4 Features implementation

In our system, the features are implemented using a set of classes whose structure follows the BNF definition of the features. That is, there is a class to represent each left-hand side symbol of the rules, and a subclass of this class for each alternative on the right-hand side. An example of such a class is given in Figure 3.7. A notable exception is for the constants that are directly represented using built-in types of COMET.

Each formula is represented using an object, and this object contains references to its subformulas. A unique name is associated to some formulas, they correspond to named features.

The transformation from the XML description to the set of objects is done very simply, by reading the XML files and creating the objects as they appear. As an example, we show a part of this traversal for the "And" element in Figure 3.8. The only difficulty is the reuse of named features, as they are maybe defined after they are used. We solve this problem by using an object to represent the named feature. That object contains a reference to its definition. This reference may be initially null.

There are some potential problems with our approach. The first one is when a

```
1  class ClassificationAnd extends ClassificationElement{
2      set{ClassificationElement} _elements;
3      ClassificationAnd():ClassificationElement(){[...]}
4      void addElement(ClassificationElement elem){
5          _elements.insert(elem);
6      }
7  }
```

Figure 3.7: Example of class for the features representation. It corresponds to the case $\bigwedge_i$ `<class-or-label>`$_i$.

```
1  ClassificationElement addCharac(XMLElement elem){
2      [...]
3      if(elem.getName().equals("And")){
4          ClassificationAnd cand();
5          forall(e in elem.getParts()) cand.addElement(addCharac(e));
6          return cand;
7      }[...]
8  }
```

Figure 3.8: Example of transformation from XML to Classification objects.

named feature F used in the definition of another feature F' is never actually defined. A second problem is when two features F and F' are defined using the other one. For instance, they are defined as being the negation of each other in the two directions ($F := \neg F'$ and $F' := \neg F$). A way to solve this problem is to impose an order on the definition of the features in the XML files. Any problem would be then discovered at the time of the creation of the features. In our approach, problems are discovered at the time of classification, because some features would not get an evaluation. Although the error detection is made later, we prefer this approach that lets more freedom in the writing of the XML files.

3.3.5 The Classification Process

To classify a problem, it is necessary to give a value to each feature (a truth value for classes or labels, a real for numeric values). This evaluation of the features is performed in a bottom-up approach. A top-down approach would correspond to a call to a function `evaluate` on each object that corresponds to a named feature. The `evaluate` method would be recursively called on each of the children until reaching a function corresponding to a characteristic directly evaluated by analysis of the internal representation. This might be quite inefficient as a characteristic may appear in the evaluation of many features and would be evaluated repetitively, yielding always the same result.

For this reason, we prefer a bottom-up approach, where we evaluate each characteristic in turn and construct the evaluation of the features from the innermost subformulas and going to the outer ones as the result for their components are evaluated. To implement this, we use an event-notification scheme (which is built-in in COMET). A formula is registered for the events notified by its subformulas. As soon as those subformulas are evaluated, they send an event describing their value. When the formula has enough information to determine its own value, it sends in turn an event that could be listened by further formulas.

Another advantage of this approach is to avoid infinite cycling. If, by mistake, two formulas F and F' are defined as $F := \neg F'$ and $F' := \neg F$, upon evaluation the top-down approach would execute an infinite loop as it needs F to evaluate F' and vice-versa. Using the bottom-up approach, the two formulas will simply not be evaluated. The fact that they are not valued means that they are independent from the problem. As stated before, this is the sign that there is a mistake in the description of the features.

After the evaluation, all the features have a value (save the exception cited right above). This value is a boolean for the classes and labels, and a real for the numeric values. They are then collected to form the classification of the problem. The numeric values are simply put in the `Classification` object. The boolean features are however treated to add missing features (not evaluated to "true" during the evaluation but known to be true by the *extension* relation) and to order the classes.

All the steps of the classification consist in:

1. Evaluate all the characteristics of the problem.
2. Propagate the evaluation of the features (through events and notifications).
3. Collect in S all the classes and labels that hold (the evaluation assigns them the value `true`).
4. Add to S the classes and labels that are extended by those in S but not yet present.
5. Separate classes and labels from S to form C and L.
6. Order C such that it respects the partial order defined by the extension relation.
7. Create a `Classification` object that contains C, L and the evaluation of all numeric features.

The goal of the sixth step is to feed the synthesizer with an ordered list of classes, from the most specific to the most general (ϕ), such that the synthesizer can produce an algorithm for the most specific class that it knows about (more details are given in the next chapter). The ordered list must respect the partial order of the extension relation. The ordered list is however a total order, meaning that there are possibly several total orders that are compatible with the partial order. The total order that is created in practice is arbitrary and without heuristic information. Part of our future work consists in resolving this limitation. A solution is to present the partial order to the synthesizer, and let the synthesizer choose. But for now we prefer to keep synthesizers unaware of such choices.

The final result of the classification of a problem is thus a `Classification`

object containing an ordered list of classes, a set of labels, and a set of values. In the next chapter, we explain how this is reused to generate the search algorithm.

4

SYNTHESIS AND COMPOSITION

The classification of a problem permits to generate an adapted search algorithm to solve the problem. In the present chapter, we detail how this generation is performed. In the first section, we present the synthesis, using synthesizers, strategies and views. Then, we show how it is possible to synthesize hybrid algorithms through composition, and how it is possible to extend the system.

4.1 Synthesizers and Strategies

The goal of the synthesis is to run an algorithm that will solve a given problem of a given class. Our approach consists in having a series of algorithms that can be instantiated for a particular instance. Those algorithms are called *strategies*. A strategy is designed to solve a given class of problems. A *synthesizer* associates classes of problems with strategies. Each synthesizer assigns at most one strategy to each class of problems.

4.1.1 Synthesizers

The work of the synthesizer is very simple when it receives a problem with its classification (classes, labels and values). It looks at the sequence of classes the problem belongs to, and looks for a class that has an associated strategy, from the first (and most specific) to the last (and most general) class. The work is then delegated to the given strategy.

Before turning to strategies, there are some points that must be noted about synthesizers:

- A synthesizer is simply a mapping from classes of problems to strategies. There may be several synthesizers, as there are several ways to solve problems. In particular, we decided to declare a synthesizer for each underlying technology

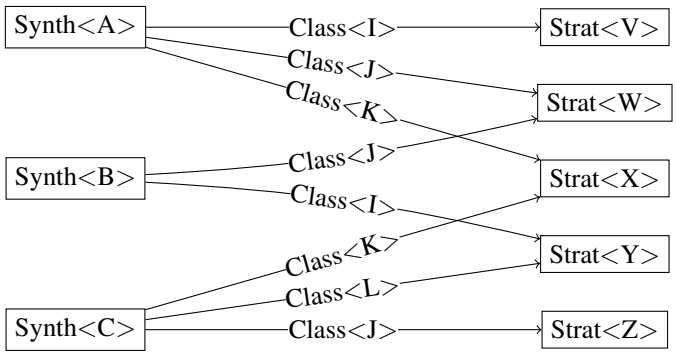


Figure 4.1: Synthesizers and Strategies

```
1  class ScheduleSynthesizer{
2      ScheduleSynthesizer();
3      void setParameters(Parameters param);
4
5      Solution<Mod> resolve(Schedule<Mod> sched)
6      Solution<Mod> resolve(Classification classif);
7      Solution<Mod> resolve(Classification classif, Solution<Mod> initSol);
8      Classification getClassification(Schedule<Mod> sched);
9      void registerStrategy(string name, ScheduleStrategy strategy);
10     Event improvingSolution(Solution<Mod> sol);
11 }
```

Figure 4.2: The synthesizer root class.

(Constraint Programming (CP), Local Search (LS), greedy...), and a default synthesizer. The user may then choose the technology he wants to use, if he is aware of. Figure 4.1 schematically shows how synthesizers are mapping classes to strategies.

- A synthesizer does not have to associate a strategy to each single problem class. This is useful when there are many classes that are small restrictions of a general class. If there is an algorithm that takes advantage of a given reduction, it can be associated to the class, but when this is not the case, the synthesizer simply switches to the general strategy. This also means that there should be a default strategy that solves any problem. As this is not the case in general, this means that not all problems may be solved with all synthesizers.
- Synthesizers may be extended and composed quite simply. This is detailed in Sections 4.2.1 and 4.2.2.

Figure 4.2 presents the main methods of a synthesizer. There are three `resolve` methods (lines 5-7). The first one takes a `Schedule<Mod>` object in input, while

```
1  class ScheduleStrategy{
2      ScheduleSolver(){}
3      void setParameters(Parameters param);
4      Solution<Mod> resolve(Classification c);
5      Solution<Mod> resolve(Classification c, Solution<Mod> initSol);
6      Event improvingSolution(Solution<Mod> sol);
7  }
```

Figure 4.3: The strategy root class.

the two others take a `Classification` object. The first one is intended for the user, and the two others for internal use. Note that the `Classification` object contains a reference to the schedule it is related to. The three methods return a `Solution<Mod>` object, which is explained in Section 4.1.3.

The method on line 8 takes a schedule and returns its classification (delegated to the `ScheduleClassifier`, and constructed as shown in the previous chapter). The method of line 9 makes it possible to associate a new strategy with any class. If there was already a strategy defined for this class, it is replaced by the new one. The last line introduces an event. This event is notified every time a new improving solution is found. This event may be fired several times by the search algorithm.

4.1.2 Strategy Instantiation

A strategy is responsible for instantiating and running a particular algorithm on a given instance. For this, it has access to the problem (through views), its classification (in particular labels and values) and a set of parameters chosen by the user. This set of parameters is used to give indications to the search algorithm, but they are not part of the problem description. It is the case of the time limit allowed to solve the problem, or different requests about optimality conditions.

Figure 4.3 shows the main methods of the `ScheduleStrategy` class, the parent class of all strategies. There are only three methods, in addition to the constructor. The first one sets the parameters, and the two others are responsible for solving a problem (given by its classification), without or with an initial solution. Comparing with the synthesizer methods, the methods of the strategy have exactly the same signatures. This underlines the fact that the synthesizer simply delegates the solving to a strategy. The last line declares an event that can be notified every time a new solution is found by the search algorithm.

Inside the `resolve` method, the strategy instantiates and runs an algorithm. The instantiation of the algorithm can follow several patterns on how to use the information:

1. The algorithm is fully determined. There remains to feed the input data, using views of the problem. Views are detailed in Section 4.1.4.

2. The algorithm has parameters to fix in function of labels and values. For instance, this may be the case of the tabu length based on the number of activities in the problem.
3. The algorithm contains building blocks that are used or not, in function of the presence of a label, or of a value being over a threshold. Such an example is the use of a LNS in CP, based on the number of resources. Each building block may in turn need further adaptation based on the features of the problem. Parameters given by the user may also be used to force the use of a particular block, e.g. for test purposes.

4.1.3 Solutions

The class `Solution<Mod>` stores solutions for scheduling problems. Objects of the class `Solution<Mod>` assign a value to each decision variable of the problem. This assignment is expressed in terms of the modeling objects. For instance, the method `getStartingTime(Activity<Mod> act)` returns the starting time of an activity. Beside the starting time, other decision variables of activities are the completion time, the set of resources effectively used, the mode (for multi-mode activities), and a boolean that tells whether the activity is effectively executed (for optional activities). The solution also records the value of the objective function under this assignment. The main benefit of solution objects is that the model stays independent. It can thus have several solutions that can be compared. Moreover, solutions serve to communicate between cooperating strategies. They can be used to perform an initial assignment, to provide an upper bound, or to guide heuristics. This is why the `resolve` methods of the synthesizers and strategies may take an additional argument, which is an initial solution.

4.1.4 Views

The views are the interface between the internal form of the problem and the solvers. They have two interests. The first one is that they present a simple and uniform interface to the solver writer. All the objects of the schedule are represented by integers so that the solver does not have to deal with other kinds of objects than its own. The second advantage of the views is that several different views are proposed to retrieve the same information. This allows to use the most appropriate view for each problem. For classical problems, there are simplified views giving direct access to the important information, while for more complicated problems, the interface is more complex to deal with all possibilities. We defined a (set of) view(s) for each part of the problem definition (activities, precedences, resources, requirements and objective function), and a unique view to access a solution in the same terms as the other views.

Views for Activities The same view is used if activities are single or multi-mode. Each activity and each mode receives a unique integer identifier that is used to retrieve the information. For single mode problems, modes and activities are equal.

Views for Precedences This view gives access to the set of precedences. The precedences are represented as 5-tuples "(source-id, source-start, target-id, target-start, delay)". The "source-start" and "target-start" are boolean flags telling whether the precedence is defined on the start or the end of the activity. The precedences returned correspond to the ones found in the *almost* transitive reduction. It is also possible to retrieve only the precedences related to a subset of the tasks. In addition, this view gives access to the release dates and deadlines of the activities.

A more specific view, the JobView adds the ability to get the (ordered) activities of a job and the job containing an activity. A job is detected as being a chain or a cycle of activities. This view is useful to instantiate algorithms for Job-Shop-like problems.

Views for Resources The view for the resources gives access to the information available in the resources of the internal form. Like for activities, the resource view associates a unique identifier with each resource. This identifier is used to access the characteristics of each resource. There is no special view if the problem is composed only of machines, as there is no special way to access such information. There is just less information to gather.

Views for Requirements The views for the requirements may take different form, depending on whether there are alternatives (disjunction) or not. In the general form, it is possible to access the particular requirements by the index of the mode, the index in the conjunction and the index in the disjunction. If there is no disjunction, we can remove one level, leading to a slightly shorter interface.

Views for Objectives There are two views for the objective functions. The first is a general one that allows to retrieve the whole objective tree node by node (each node having a unique identifier). The second view is intended for uniform objectives, where the objective is the sum or the maximum of a simple function over all the activities. This is the case for many classical objectives. Very simple objectives, as the makespan, don't even need a view.

Views for Solutions The view for the solution allows to fill a solution in terms of the other views. That is the activities, modes and resources are represented by their identifiers.

In Figure 4.4, we show a small piece of code of a typical `resolve` method of a strategy, showing the use of views. This is taken from a Job-Shop Problem solving strategy. Lines 3-9 illustrate the creation of the input of the algorithm. The actual search algorithm is not shown. In lines 15-17, the solution view is created. Line 18 returns the result as a `Solution<Mod>`.

4.2 Extension and Composition

The architecture used for the resolution of scheduling problems allows to easily extend its capabilities. A first way is to add new algorithms to the set of strategies, as

```
1  Solution<Mod> resolve(Classification c){
2    //Data initialization
3    CanonScheduleView sv(c.getSchedule());
4    range Acts = sv.getActivitiesView().getActivities();
5    range Jobs = sv.getJobsView().getJobs();
6    range Machines = sv.getResourcesView().getResources();
7    int[] duration = all(i in Acts) sv.getActivitiesView().getProcessingTime(i);
8    int[] machine =
9        all(i in Acts) sv.getConjunctiveRequestsView().getResource(i,1);
10   int[][] jobAct = all(j in Jobs) sv.getJobsView().getOrderedActivitiesOfJob(j);
11
12   //Actual search algorithm not shown
13   int[] startdates = //start dates computed by the search algorithm
14   //Solution creation
15   SolutionView sol(sv);
16   forall(i in Acts) sol.setStartingDate(i,startdate[i]);
17   sol.setValue(max(i in Acts)(startdate[i]+duration[i]));
18   return sol.getModelSolution();
19 }
```

Figure 4.4: Using views to instantiate a strategy.

described in Section 4.2.1. Other ways are to combine existing strategies and synthesizers, which is explained in Section 4.2.2. Finally, we show how it is possible to reuse this architecture to introduce side effects, such as visualization (Section 4.2.3).

4.2.1 Adding new Strategies

It is easy to plug-in new underlying algorithms, as they are located in one (or two) `resolve` method, and only need to use the views to access the information. Naturally, writing a search algorithm is maybe not easy, but at least the interface with the system is simplified.

To exemplify that claim, we show how we can add a CP approach for the problem $P||Cmax$, which amounts to minimize the makespan of a set of activities that can all execute on a set of parallel machines. This problem reduces to a bin-packing problem, where the bins are the machines and the objects are the activities. Small bin-packing problems can be solved using CP and dedicated global constraints.

To integrate this, the first thing to do is to create the strategy to solve the problem. We show the corresponding code in Figure 4.5, on page 55. We override the two methods of the super class by a call to a third method doing the job, as the code will be the same whether we have an initial solution to work with, or not. If there is an initial solution, the algorithm takes its value to get an upper bound. Views are used to get the data (number of activities, number of machines and duration of each activity).

```
1  class PCmaxStrategy extends ScheduleStrategy{
2      PCmaxStrategy():ScheduleStrategy(){}
3      Solution<Mod> resolve(Classification c){
4          return resolve(c,System.getMAXINT());
5      }
6      Solution<Mod> resolve(Classification c, Solution<Mod> initSol){
7          Solution<Mod> sol = resolve(c,(int) initSol.getValue());
8          if(sol!=null) return sol; else return initSol;
9      }
10     Solution<Mod> resolve(Classification c, int ub){
11         CanonScheduleView sv(c.getSchedule());
12         range Activities = sv.getActivitiesView().getActivities();
13         range Machines = sv.getResourcesView().getResources();
14         int[] duration =
15             all(i in Activities) sv.getActivitiesView().getProcessingTime(i);
16         int horizon = min(ub,sum(i in Activities)duration[i]);
17         SolutionView sol(sv);
18         Boolean found(false);
19         Solver<CP> cp();
20         var<CP>{int} bin[Activities](cp,Machines);
21         var<CP>{int} load[Machines](cp,0..horizon);
22         minimize<cp> max(i in Machines)load[i]
23         subject to{
24             cp.post(multiknapsack(bin,duration,load));
25         }using{
26             forall(i in Activities: !bin[i].bound())
27                                             by (bin[i].getSize(),-duration[i]) {
28                 int ms = max(0,maxBound(bin));
29                 tryall<cp>(j in Machines: j <= ms + 1 && bin[i].memberOf(j))
30                                                     by (load[j].getMin())
31                     cp.label(bin[i],j);
32             }
33             int m[i in Machines] = 0;
34             forall(i in Activities){
35                 sol.setStartingDate(i,m[bin[i]]);
36                 sol.addUsedResource(i,bin[i]);
37                 m[bin[i]]=m[bin[i]]+duration[i];
38             }
39             sol.setValue(max(i in Machines)load[i]);
40             found := true;
41             notify improvingSolution(sol.getModelSolution());
42         }
43         if(found) return sol.getModelSolution(); else return null;
44     }
45 }
```

Figure 4.5: A complete strategy to solve the $P||Cmax$ problem.

```
1  class PCmaxSynthesizer extends ScheduleSynthesizer<CP>{
2      PCmaxSynthesizer():ScheduleSynthesizer<CP>(){
3          registerStrategy("PCmax", new PCmaxStrategy());
4      }
5      PCmaxSynthesizer(Parameters p):ScheduleSynthesizer<CP>(p){
6          registerStrategy("PCmax", new PCmaxStrategy());
7      }
8  }
```

Figure 4.6: A new synthesizer to solve the $P||Cmax$ problem.

Then a CP model is build (there is only one constraint, on line 26) and solved. The branching strategy puts activities in the bins. It starts with the largest activities and puts them in the less loaded bins. If there are several empty bins, it only tries one. The starting time of the activities is fixed arbitrarily once all the activities are placed. Each time the solver finds an improving solution, that solution is recorded in the solution object, and the solution is notified. After the completion of the algorithm, the solution is returned.

There remains now to associate the new strategy with a class of problems. In Section 3.3.3, we showed an XML file to classify the problem we are tackling under the identifier "PCmax". Associating the class and the strategy is done via the following method of a synthesizer:

```
1  registerStrategy(''PCmax'', new PCmaxStrategy());
```

To have a complete example, Figure 4.6 shows the creation of a new synthesizer that extends the CP one and adds the new strategy. This new synthesizer can subsequently be used by declaring it and calling its `resolve` method on a scheduling problem:

```
1  PCmaxSynthesizer synth();
2  Solution<Mod> sol = synth.resolve(sched);
```

If the problem (`sched`) complies to the constraints of the "PCmax" class (meaning that it is a $P||Cmax$), it will be solved with the new strategy. Otherwise, it will be solved with another strategy that was previously defined in the `ScheduleSynthesizer<CP>` class (see Section 5.4.2 for details).

4.2.2 Strategies and Synthesizers Composition

Beside adding new strategies, it is also convenient to compose existing strategies to produce more robust algorithms. Hybridization of algorithms have proved to be a powerful direction to improve the resolution of hard problems. In this section, we show how it is possible to compose simple hybrids. We limit ourselves to sequential hybrids, but it is easy to extend the idea to parallel search algorithms.

The simplest composition is to chain two algorithms and to use the output of the first one as an initial solution for the second one. This initial solution can be used to

```
1  class ScheduleSynthesizer<Chain> extends ScheduleSynthesizer{
2      ScheduleSynthesizer _s1;
3      ScheduleSynthesizer _s2;
4      ScheduleSynthesizer<Chain>(ScheduleSynthesizer s1,
5                      ScheduleSynthesizer s2):ScheduleSynthesizer(){
6          _s1 = s1;
7          _s2 = s2;
8      }
9      Solution<Mod> resolve(Schedule<Mod> sched){
10         Classification c = getClassification(sched);
11         return _s2.resolve(c,_s1.resolve(c));
12     }
13 }
```

Figure 4.7: A new synthesizer to chain two synthesizers.

add a constraint on the objective value (as it is done in the $P||Cmax$ example of the previous section), as a starting solution of a local search, or to guide a search heuristic. A chaining strategy is created very simply by implementing the `resolve` method as follows:

```
1  Solution<Mod> resolve(Classification c){
2      return _s2.resolve(c, _s1.resolve(c));
3  }
```

where $\_s1$ and $\_s2$ are the two strategies that are chained. This can be done at the synthesizer level as well, and the code would be almost the same, as shown in Figure 4.7, where strategies do not appear.

In the direction of chaining synthesizers, it is possible to chain more than two algorithms or to iteratively switch between two or more algorithms. It is then often necessary to put a time limit on each of the algorithms (using the `Parameters` class). Starting from there, it is easy to implement portfolios of algorithms with more complex schemes, as e.g.:

- Adaptive round-robin, where each algorithm is run for a given time but the time or the place in the queue changes depending on the performance of the algorithm.
- Randomized search algorithms with restarts, where an algorithm with random behavior is repeatedly run, possibly reusing the previous knowledge.

In the current state of our prototype, there are two main limits to the composition of algorithms. The first one is that there is no way to suspend an algorithm and restart it later. This means that, when called, the strategies restart each time from scratch. The second one is that the system is sequential and the time limit enforcement is up to the strategies. This means that there is no way for a synthesizer to stop an algorithm

if it takes too long, nor to limit its running time if the strategy does not take it into account. We are currently working on solving those problems, which should also allow to design parallel hybrid search algorithms[1].

4.2.3 Visualization and Side Effects

Strategies can be seen as black-boxes that take as input a classified problem and output solutions to this problem (both using the notification mechanism and the return statement). As seen right above, those building blocks can be combined to yield hybrid algorithms. But we can also introduce some building blocks that introduce side-effects and combine them with the other strategies. We identified several such side-effects of interest:

- Visualization of solutions.
- Printing and saving of solutions.
- Printing the Graham's classification of the problem.
- Collecting statistics.

For instance, the code below declares a synthesizer that will solve problems using Constraint Programming for 60 seconds and print the solutions to the console, then improve the solution with Local Search and show the improvements graphically.

```
1 Parameters p();
2 p.setParameter(``timeLimit'',60);
3 ScheduleSynthesizer<Chain> synth(
4       PrintSynthesizer(ScheduleSynthesizer<CP>(p)),
5       VisualSynthesizer(ScheduleSynthesizer<LS>()));
6 Solution<Mod> sol = synth.resolve(sched);
```

These effects could be introduced where needed in individual existing strategies but it would be cumbersome to repeat existing work. They could also be introduced outside the system (like printing a solution, for instance). However we believe that proposing them as additional building blocks favors a declarative style which is more flexible.

As a typical example, the code of `PrintSynthesizer` is shown in Figure 4.8, except the body of the `print` method which is not relevant. The code to introduce printing is very simple. One important thing is that the synthesizer must forward the solutions after having printed them, so that this synthesizer can in turn be used inside other synthesizers. In this example, there is only one print method for all scheduling problems but it is entirely possible to implement several strategies for printing differently the solutions of different classes of problems. For instance, Job-Shop-like problems would be preferably printed job by job, and for single-mode problems it is useless to print the mode of each activity.

A last remark is that synthesizers and strategies are stateful objects. This means that side-effects may include modifications of the internal state of the objects. Although we did not investigate that direction, it might be interesting to use this feature

[1] Note however that a strategy can run a parallel search but this is internal to the strategy.

```
1  class PrintSynthesizer extends ScheduleSynthesizer{
2      ScheduleSynthesizer _s1;
3      PrintSynthesizer(ScheduleSynthesizer s1):ScheduleSynthesizer(){
4          _s1 = s1;
5          whenever _s1@improvingSolution(Solution<Mod> sol){
6              print(sol,false);
7              notify improvingSolution(sol);
8          }
9      }
10     Solution<Mod> resolve(Schedule<Mod> sched){
11         Solution<Mod> sol = _s1.resolve(sched);
12         print(sol,true);
13         return sol;
14     }
15     void print(Solution<Mod> sol, boolean finalSol){ ... }
16 }
```

Figure 4.8: The `PrintSynthesizer` class.

for learning purposes, in which the synthesizer adapts itself using the results of past solving rounds.

5

THE AEON PROTOTYPE

This chapter presents AEON, the tool we developed during our thesis. The first section recapitulates our contributions. It presents how AEON supports them, the design decisions we made, and the main strengths and weaknesses of the system. The second section presents the general architecture of the system. In the third section, the modeling interface of AEON is presented, followed by some examples of models. The fourth section shows how to solve a problem using AEON and details the underlying synthesizers. Finally, the last section summarizes how AEON can be extended.

5.1 Implementing our contributions

Independence between the model and the search algorithms. We implemented a modeling layer totally independent from the underlying solving technologies. The advantages are that it is possible to solve the same problem with different technologies, to plug different solvers under AEON, and to reason about the model independently from the technologies. The main weakness is that the user has no abstraction at the modeling level to guide the search. To partially overcome this problem, we provide in AEON additional mechanisms to specify the search (see Chapter 4). It is not mandatory to use them, but we keep the door open.

The modeling abstractions of our prototype are regular COMET code (classes, methods and functions). The advantages of this approach are that the modeler can make use of the facilities of the COMET language (e.g. array comprehension), and that the system can be embedded in a larger application written in COMET. This second advantage makes sense because the solution of a problem solved by AEON is returned in an object that can then be queried and used further.

The set of modeling abstractions defined in AEON is presented in Section 5.3 with examples of models. They are abstractions very similar to those found in OPL or COMET, for instance. The set of abstractions also allows to express things in different fashions, such that the models look natural.

Structural analysis and classification of problems. We propose basic characteristics that can be used to analyze and classify a problem. The goal is to have a global picture of the problem to choose an adequate search algorithm. To implement this, we made the following choices:

- We defined an internal representation of the problem that uses a smaller set of abstractions than the modeling layer, while still being high-level enough (see Chapter 3). This representation is central to the system. Indeed, to solve a problem, the model is first transformed into the internal representation, which is in turn simplified before being analyzed. The search algorithms also indirectly access the internal representation to initialize the value of the constants (e.g. the processing time of the activities). Consequently, the design of the internal representation is meant to ease the translation from the model, the mapping to the solvers, and the retrieval of characteristics.

- At the modeling level, there is no separation between the model and the data. This approach contrasts with classical modeling languages where the data is located in a separate file. We consider that the data is explicitly part of the problem, and the analysis is performed on the internal representation that contains the data. It does not prevent to have different data sets for the same model, but this must be done through explicit reading of different files containing the data (see the examples in Appendix B).

- The set of modeling abstractions cannot be extended by the user. It is indeed necessary to know what is representable, to allow the analysis to behave correctly. The downside is that it is not possible to model problems that do not fit in the proposed interface. Our future work should focus on removing this limitation.

- For the result of the classification, we choose to mark problems with a set of labels, a set of classes, and a set of numeric values. Altogether they characterize the problem and its main features (see Chapter 3). Labels represent binary characteristics of the problem (e.g. "disjunctive"), classes represent defined types of problems (e.g. JITJSP), and numeric values are for non-binary features (e.g. mean number of activities per machine). The set of possible features can be extended, to define new features to recognize.

Automatic generation of search algorithms. The automatic creation of the search is supported by the classification of the problem. The structure of the problem and its features mainly define the algorithm to use to solve it. For instance, particular classes of problems can be solved in polynomial time with an appropriate algorithm. It is necessary to have an association from classes to algorithms. In AEON, this association is defined in objects called synthesizers. There exist several synthesizers, each one using a particular search paradigm (e.g. Constraint Programming (CP) or Local Search (LS)).

A synthesizer associates each class of problems with a strategy. Each strategy is responsible for the creation of the algorithm for a particular class of problems. A strategy has access to the labels and numeric values defined for the problem, such that it can adapt the search algorithm to particularities of the problem at hand. Chapter 4 explained in depth synthesizers and strategies.

Unlike for the modeling abstraction, the synthesis can be extended. It is possible and simple to plug in new search algorithms for any class of problem. We use the concept of view to give access to the data defined in the internal representation, independently of the modeling abstractions. Views hide the complexity of the model, such that the algorithm writer can focus on the search algorithm.

Loosely coupled hybrid algorithms. Having a model decoupled from the search algorithms, and synthesizers that automatically generate these search algorithms, it is easy to generate different search algorithms from the same model. It is then possible to combine several search algorithms and to create loosely coupled search algorithms. Each synthesizer can be seen as a black-box. The only interaction between the algorithms is performed through solutions. The result of a search algorithm can be used in another one, as the initial solution of a local search, or to bound the value of the objective function. We explained this composition in Section 4.2.

Propagators for global constraints. We developed propagators for two global constraints for scheduling. The first one propagates the disjunctive constraint using the position information. The second one propagates the Just-In-Time objective. The first one was initially implemented in Gecode [Gec06], and is currently being integrated in AEON. The second is implemented in COMET, on top of the CP scheduling module, and is used by AEON, when presented with problems presenting the Just-In-Time objective. These propagators will be presented in Chapters 7 and 8.

5.2 Architecture

The present section describes how the whole system is organized and how each component is related to the other ones.

The resolution of a scheduling problem in AEON goes through the following steps:

1. The problem is modeled by the user, using the proposed abstractions.
2. The model is transformed to the internal representation, which is simplified, and then analyzed to get characteristics.
3. The problem is classified using the value of the characteristics. This yields a set of features of the problem.
4. An algorithm is chosen and adapted, based on the features of the problem. This is done by the synthesizers and strategies.
5. The algorithm is executed, and outputs a solution.

According to this path of execution, the architecture of AEON can be decomposed into five main modules:

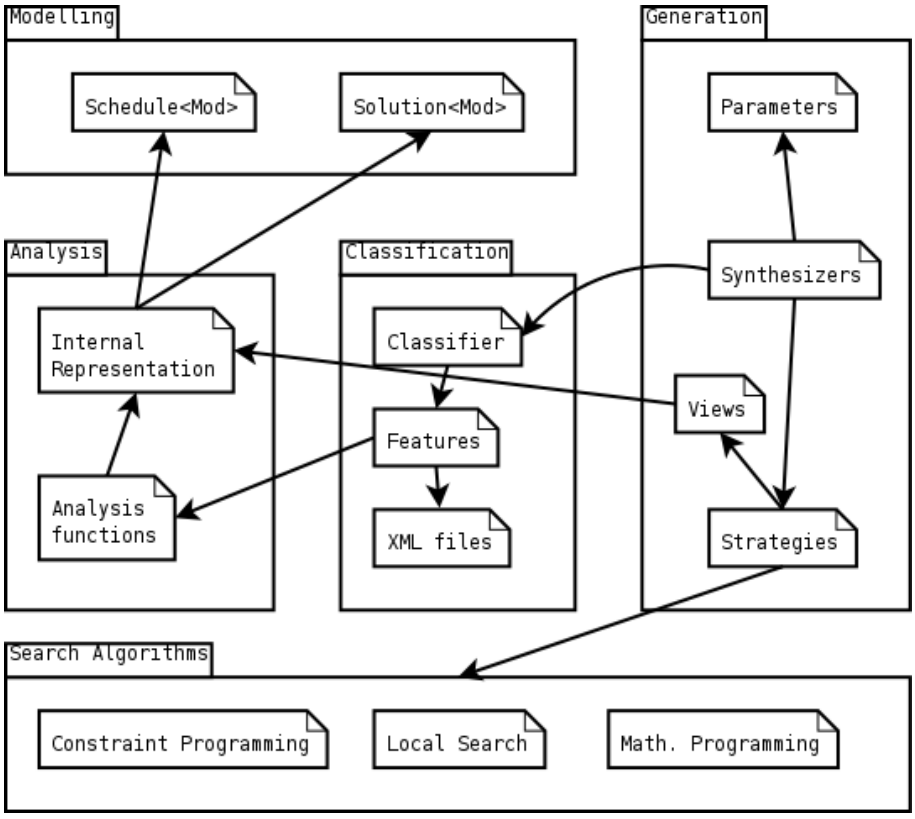


Figure 5.1: Overview of AEON's architecture. Each module contains a set of classes and functions. An arrow denotes a "use" link. Details and explanations are provided in the text.

1. Modeling: All the classes and functions related to the modeling of a problem.
2. Analysis: The internal representation of a problem and the analysis functionalities.
3. Classification: The (COMET) classes used to classify a problem.
4. Generation: The classes that perform the generation, and the model views.
5. Algorithms: A set of separate modules for the underlying search algorithms.

In the remaining of this section, we will review the main capabilities of each module and how they interact with the other modules. Figure 5.1 shows a summary of the modules and their interactions. Note that COMET does not feature namespaces or packages. This means that the decomposition presented here is conceptual but is not explicit in our implementation.

5.2.1 Modeling

The modeling classes are organized around the central class `Schedule<Mod>` that provides access to the whole problem description. It comprises all the other classes to represent activities, resources, requirements, precedences, and objective functions. The modeling module also contains the class `Solution<Mod>`. That class stores solutions of scheduling problems. Each solution object depends on a `Schedule<Mod>` object and describes the value of the different decision variables, namely starting and ending time of activities, modes, and resource alternatives.

The modeling module is the place of interaction with the user. From this point of view, the modeling module has very limited interactions with the remaining of the system. It has no knowledge of what is inside the system. Rather, all the model is accessed through the internal representation (see below). The modeling classes are presented in details in Section 5.3.1.

5.2.2 Analysis

The analysis functionalities form a big part of the system. Indeed, it contains all the classes that define the internal representation, the transformation from the model to this representation, and all the functions that retrieve the pieces of information from the internal representation. Details on the internal representation and the analysis were given in Chapter 3. This module is the only one to access the model directly. In turn, it is queried by the classification and generation modules for their respective tasks.

5.2.3 Classification

The classification module is in charge of marking a problem with a set of labels, classes and numeric values. The definition of these features is done in XML files. The tagging of a problem with a set of features is based on the analysis of the internal representation. The explanations on the classification was the object of Chapter 3, together with the analysis part. The classification process is triggered by the generation module, and makes use of the functions offered by the analysis module.

5.2.4 Generation

The generation module is the second big part of AEON. It contains all the synthesizers and strategies that are currently defined. A `ScheduleSynthesizer` associates classes of problems with strategies. A `Strategy` is responsible for running a given algorithm for a class of problems. The strategy must be instantiated, which is done via `Views`, a set of classes that allow a simple and direct access to the information contained in the internal representation.

The module also contains the `Parameters` class, which is a place of interaction with the user. It stores additional pieces of information that are forwarded to the solvers, without being part of the problem. The main parameter is a time limit on the algorithms, but the class allows to store other preferences that may overwrite the choices of the system.

This module is also the entry point to the whole solving procedure. When a user asks a synthesizer to solve a problem, the synthesizer delegates the classification to the classification module, and the generation to the right strategy that is instantiated based on the internal representation (through views). More details on the synthesis may be found in Chapter 4.

5.2.5 Algorithms

This layer contains a set of algorithms to solve scheduling problems. It is composed of the CP and LS modules of COMET, extended with our own global constraints and heuristics. Those modules may be used without AEON. From AEON's point of view, the interaction with the search algorithms is located only in the strategies. Other modules may be defined, such as e.g. a MIP one. They are detailed in the last section of this chapter.

5.3 Modeling with AEON

In the present section, we present the modeling layer for scheduling problems in AEON. As already stated, AEON is built on top of COMET. However, as we want to be as independent as possible from the underlying search algorithms, we chose not to use any of the existing classes in the scheduling modules, and build our own modeling layer on top of the AEON system. Our modeling classes are postfixed with "<Mod>", to denote their difference with the other classes of COMET. In the remaining of this section, we review in details the available abstractions, and we provide examples of complete models for some well-known problems.

5.3.1 Model Abstractions

To simplify reading, the post-fix "<Mod>" is omitted in the remaining of this section (Section 5.3.1). Furthermore, we don't present the detail of the methods and functions that can be used. They can be found in Appendix A, that contains the exact API of modeling.

The central class of a model is `Schedule` (i.e. `Schedule<Mod>`). This class is a placeholder for all the objects and constraints of a problem. This class defines the main features of the problem, which are the considered time horizon (starting from zero), the objective type (maximization, minimization or satisfaction), and the preemption.

To represent activities, there are two classes. `Activity` and `MultiModeActivity` represent single- and multi-mode activities respectively. At creation time, an `Activity` receives as input a `Schedule`, a processing time and a name. The processing time is either fixed or defined by lower and upper bounds. A `MultiModeActivity` is given the `Schedule`, the number of modes, and a name. The processing time of the modes are given separately for each mode. The methods available on

activities (single- and multi-mode) allow to specify preemption, the membership to a `Job`, the resource requirements, and the precedences between activities. The requirements are mode-dependent but the remaining constraints are common to all modes of an activity. Precedence constraints can involve the start and the end of activities and jobs. They can also define delays. The aforementioned `Job` class represents groups of activities logically related. The activities of a job can be executed at the same time, unless it is stated otherwise. Jobs share some features with activities: They can be grouped into other jobs and their ends and starts can be constrained with precedences. The precedences between activities and jobs may involve positive and negative delays, making possible to impose a maximum distance between two activities. Lastly, activities can be defined as optional, meaning that their execution is not required.

Resources are represented by four classes, depending on the type of resource under consideration. The `Machine` class represents unary resources. Two activities that require the same machine cannot overlap in time. The `CumulativeResource` class represents renewable resources. At every moment, the sum of the requests of the activities being executed cannot exceed the capacity of the resource. On the contrary, the `Reservoir` class is used for non-renewable resources whose capacity is decreased (or increased) by the execution of the activities. A minimum capacity can be defined for both the `CumulativeResource` and `Reservoir` classes. For these two classes and the `Machine` class, it is possible to define (periodic) breaks, i.e. time intervals of unavailability. The last kind of resource is the `StateResource` that represents a state of the world. The resource can only be in one state at a time. Two activities that require different states cannot overlap in time. For machines and state resources, it is possible to define sequence-dependent setup times. The set of requirements of an activity (or of a mode of a multi-mode activity) has the form of a tree whose internal nodes are either conjunctions or disjunctions of simpler requests. External nodes are the basic requirements: a required machine, some required or provided amount of a resource, some consumed or produced amount of a reservoir, or a particular state of a state resource.

Objective functions are subclasses of `ScheduleObjective`. The subclasses are either simple or compound functions. Compound functions are obtained by summing or taking the maximum of other functions, or multiplying a function by a constant. Simple functions are the classical lateness, tardiness, earliness, and, more generally piecewise-defined linear functions based on the completion time of activities and jobs. The set of simple functions includes also cost functions associated with the modes of multi-mode activities and with the absence of optional activities. The global objective function is passed to the `Schedule` object with a method that specifies if the function must be minimized or maximized.

5.3.2 Model Examples

The abstractions we just presented may be used to represent a large variety of problems. It is often possible to define a same problem in different ways. In this section, we present some models of well-known problems. For the details of the methods used in those models, we refer the reader to the API in Appendix A.

```
1  range jobs = 1..nbjobs;
2  range machines = 0..nbmachines−1;
3  range tasks = 1..nbjobs∗nbmachines;
4  int proc[tasks];
5  int mach[tasks];
6  int job[jobs,machines];
7
8  Schedule<Mod> s();
9  Activity<Mod> A[i in tasks](s, proc[i], IntToString(i));
10 Job<Mod> J[i in jobs](s, IntToString(i));
11 Machine<Mod> M[i in machines](s, IntToString(i));
12 forall(i in tasks) A[i].requires(M[mach[i]]);
13 forall(i in jobs) J[i].containsInSequence(all(j in machines) A[job[i,j]]);
14 s.minimizeObj(makespanOf(s));
```

Figure 5.2: Direct model for the Job-Shop Problem

Figure 5.2 presents the classical model of a Job-Shop Problem (JSP)[1]. The initialization of the parameters (the number of jobs `nbjobs`, the number of machines `nbmachines`, the processing time of each activity `proc`, the machine required by each activity `mach` and the order of activities in each job `job`) from a file is not shown but can be found in Appendix B. The modeling of the problem actually occurs in lines 8-14. At first a schedule object is created (line 8). Then the objects populating this schedule are created (lines 9-11). For the JSP, there are activities, jobs and machines. They are all initialized with the schedule object and a name. In addition, activities have a fixed processing time. Next the constraints are stated: Machine requirements (line 12) and ordering inside jobs (line 13). Finally the objective is given as the minimization of the makespan.

A model for the Open-Shop Problem can be derived from the one of the Job-Shop, just replacing line 13 to become

```
13 forall(i in jobs) J[i].contains(all(j in machines) A[job[i,j]]);
14 forall(i in jobs) J[i].noOverlap();
```

In this case, each job contains the same activities but in any order. Line 14 adds the constraint that no two activities of the same job can execute at the same time.

It is also possible to define alternative versions of these two classical problems. For instance, Figure 5.3 represents a Job-Shop in a quite indirect way (using multi-mode activities and reservoirs). In this model, each activity has two modes (line 9), but they are both assigned the same processing time (lines 12 and 14). The method `setProcTime(1,proc[i],proc[i])` means that the first mode has a processing time that must be between `proc[i]` and `proc[i]`, that is equal to `proc[i]`.

[1] The JSP is to minimize the makespan of a set of jobs. Each job is composed of a sequence of activities, each activity requiring one machine.

```
1  range jobs = 1..nbjobs;
2  range machines = 0..nbmachines−1;
3  range tasks = 1..nbjobs∗nbmachines;
4  int proc[tasks];
5  int mach[tasks];
6  int job[jobs,machines];
7
8  Schedule<Mod> s();
9  MultiModeActivity<Mod> A[i in tasks](s, 2, "Act"+IntToString(i));
10 Reservoir<Mod> M[i in machines](s, 0, 5, 5, IntToString(i));
11 forall(i in tasks){
12     A[i].setProcTime(1, proc[i], proc[i]);
13     A[i].requires(1, M[mach[i]], 3);
14     A[i].setProcTime(2, proc[i], proc[i]);
15     A[i].requires(2, M[mach[i]], 4);
16 }
17 forall(i in tasks:i%nbmachines!=0) A[i].precedes(A[i+1]);
18 s.minimizeObj(maxOf(all(i in tasks) completionTimeOf(A[i])));
```

Figure 5.3: Indirect model for the Job-Shop Problem

Each machine is replaced by a reservoir whose capacity is equal to 5. However there is no consumption or production of resource, such that the reservoirs behave as cumulative resources. Furthermore, as every requirement is larger than half the capacity of the resource (lines 13 and 15), this is equivalent to a disjunctive problem.

Line 17 defines the precedences. It creates chains of precedence constraints. This is equivalent to the creation of a job and a call to `containsInSequence`. Finally, line 18 creates the objective which is to minimize the largest completion time of all activities, that is the definition of the makespan.

Other models for the Job-Shop and the Open-Shop are based on the Group-Shop Problem (GSP). In the GSP, the activities are grouped into sub-jobs. The sub-jobs are ordered but there is no order inside a sub-job. This problem boils down to the JSP or OSP, when the size of the groups is respectively equal to one or to the number of activities in the job. Figure 5.4 shows the declaration of a GSP.

In this model, lines 9-18 are similar to the first model of the Job-Shop: the schedule, activities and machines are created, and each activity requires a machine. The code is slightly larger because in GSP, all jobs don't not always have the same number of activities (which is the case in classical benchmarks for the JSP). In lines 20-31, the precedence structure of the problem is created. A `Job<Mod>` object is created for each group of activities. The activities of each group cannot overlap (line 21). Line 25 defines which activity goes in which group and line 27 declares the precedences between groups. If there is a precedence between two groups, this means that all activities of the first one must be completed before any activity of the second one can be

```
1  range jobs;
2  range groups;
3  range machines;
4  int[][] group = new int[][jobs];
5  int[][] proc = new int[][jobs];
6  int[][] mach = new int[][jobs];
7  int nbt[jobs];
8
9  Schedule<Mod> s();
10 Machine<Mod> M[i in machines](s, "M"+IntToString(i));
11 Activity<Mod>[][] A = new Activity<Mod>[][jobs];
12 forall(j in jobs) {
13     A[j] = new Activity<Mod>[1..nbt[j]];
14     forall(t in 1..nbt[j]) {
15         A[j][t] = Activity<Mod>(s, proc[j][t], "A");
16         A[j][t].requires(M[mach[j][t]]);
17     }
18 }
19
20 Job<Mod> SJ[i in groups](s, "SJ"+IntToString(i));
21 forall(i in groups) SJ[i].noOverlap();
22 forall(j in jobs){
23     int n = group[j][1];
24     forall(t in 1..nbt[j]){
25         SJ[group[j][t]].contains(A[j][t]);
26         if(n!=group[j][t]){
27             SJ[n].precedes(SJ[group[j][t]]);
28             n = group[j][t];
29         }
30     }
31 }
32
33 s.minimizeObj(makespanOf(s));
```

Figure 5.4: Model for the Group-Shop Problem

```
1  range tasks;
2  range resources;
3  int duedate;
4  float tardCost;
5  int[][] succ = new int[][tasks];
6  int proc[tasks];
7  int req[tasks, resources];
8  int capa[resources];
9
10 Schedule<Mod> s();
11 Activity<Mod> A[i in tasks](s, proc[i], "Job"+IntToString(i));
12 CumulativeResource<Mod> R[i in resources](s,capa[i],"Res"+IntToString(i));
13 forall(i in tasks){
14     forall(j in succ[i].getRange()) A[i].precedes(A[succ[i][j]]);
15     forall(j in resources : req[i,j]!=0) A[i].requires(R[j],req[i,j]);
16 }
17 s.minimizeObj(Tardiness<Mod>(s,A[nbjobs],duedate)*tardCost);
```

Figure 5.5: Model for the RCPSP

executed.

Beside the makespan, other objectives may be defined for any problem. Here are some classical ones (weighted sum of the tardiness of the jobs, and weighted sum of the earliness and tardiness of the activities).

```
1  s.minimizeObj(sumOf(all(i in jobs)(tardinessOf(J[i],dd[i]) * w[i])));
```

```
1  Tardiness<Mod> T[i in tasks](s, A[i], dd[i]);
2  Earliness<Mod> E[i in tasks](s, A[i], dd[i]);
3  s.minimizeObj(sumOf(all(i in tasks)(T[i] * tc[i] + E[i] * ec[i])));
```

The first code uses a facility function defining the tardiness, while the second one directly declares the Tardiness and Earliness objects.

Figure 5.5 presents how to model another very known problem class, the RCPSP . In this problem, activities are linked by simple precedence constraints (line 14), and each activity requires some amount of several cumulative resources (line 15). The objective is to minimize the tardiness cost of the last activity with respect to some due date (line 17).

The RCPSP model can be easily extended to become Multi-Mode or to introduce maximum time lags between activities (see Appendix B). A limitation of AEON is however for the MMRCPSP/max. This problem indeed defines different precedence constraints depending on the modes of the activities. Our framework only considers fixed precedence constraints.

Many variations of the One-Machine problem may be defined and have been studied in the literature. It is easy to represent them in AEON. As a last example, the code

```
1  range acts;
2  int p[acts];
3  int r[acts];
4  int w[acts];
5
6  Schedule<Mod> s();
7  Activity<Mod> A[i in acts](s,p[i],"A"+IntToString(i));
8  Machine<Mod> M(s,"M");
9  M.isRequiredBy(A);
10 forall(i in acts) A[i].isReleasedAt(r[i]);
11 s.minimizeObj(sumOf(all(i in acts)(w[i]*completionTimeOf(A[i]))));
```

Figure 5.6: Model for the $1|r_i|\sum w_iC_i$

in Figure 5.6 defines the minimization of the weighted sum of completion times on one machine, and subject to release dates ($1|r_i|\sum w_iC_i$ in Graham's notation). Line 9 declares in one instruction that all activities in the array "A" require the machine. Line 10 adds release dates for each activity. It would be easy to add or replace lines to define different problems (introducing deadlines, allowing preemption, minimizing tardiness, and so on).

More models may be found in Appendix B.

5.4 Solving with AEON

5.4.1 The Synthesizers interface

The aim of the previous section was to present how straightforward it is to represent classical problems in AEON. It showed also that such a simple problem as the Job-Shop may be represented using many different models. What is not visible here, but that is very important, is that all the versions of this problem are effectively recognized as a Job-Shop Problem by the system and solved as such.

What has not been shown either in the code fragments is how to solve the model. This is done very simply through the declaration of a synthesizer and the call to the method `resolve`, as shown next.

```
1 ScheduleSynthesizer<CP> synth();
2 Solution<Mod> sol = synth.resolve(s);
3 if(sol!=null) sol.printSolutionToFile(''somefile.txt'');
```

This excerpt supposes that "s" is a `Schedule<Mod>` object, as defined by one of the models of the previous section. The first line can be replaced by the call to another synthesizer. The ones currently defined are the following ones:

- `ScheduleSynthesizer<CP>()` uses CP.

- `ScheduleSynthesizer<LS>()` uses LS.
- `ScheduleSynthesizer<Greedy>()` uses greedy approaches.
- `ScheduleSynthesizer<LNS>()` uses Large Neighborhood Search.
- `ScheduleSynthesizer<Sequence>(ScheduleSynthesizer[] s)` executes the sequence of synthesizers "s" one after the other. The solution of a synthesizer is given to the next synthesizer.
- `ScheduleSynthesizer<Repeat>(ScheduleSynthesizer s, int n)` repeats "n" times the synthesizer "s". Each execution is independent, and the best solution among all executions is kept.
- A `ScheduleAnimator(ScheduleSynthesizer s)` graphically shows the solutions given by synthesizer "s".

All the complexity of the system is hidden in the `resolve` method of the synthesizers. Whatever the used synthesizer, a call to `resolve` performs the following tasks:

1. Transform the model into the internal representation.
2. Analyze the representation to retrieve the characteristics, and marking it with labels, classes and numeric values.
3. Choose the right strategy, based on the class of the problem.
4. Delegate to the strategy, that instantiates to the particular problem, runs the underlying algorithm and returns a solution.
5. Forward the solution to the user level.

5.4.2 Current Prototype Synthesizers

In this section, we present the synthesizers currently included in AEON. Four synthesizers embed one of the following underlying technologies:

- Constraint Programming
- Large Neighborhood Search
- Local Search
- Greedy Search

The CP prototype The CP synthesizer is able to solve a large range of problems. Currently, this is the most complete synthesizer, primarily because the underlying module of COMET is able to deal with a large range of problems, and because we extended it with some new abstractions, including global constraints. The synthesizer delegates the work to a set of strategies that are tailored for specific classes of problems, and a general strategy for problems that do not fit in one of these classes. This general strategy makes use of labels and values to activate different branching heuristics and global constraints.

The specific strategies are the following:

- Job-Shop with Makespan: It uses a standard CP model and branches by ranking the machines.

- Open-Shop with Makespan: It uses a standard model that defines machines also for the jobs. The branching is performed with a ranking of all the machines.
- Job-Shop with sum of weighted Earliness and Tardiness: We use a dedicated global constraint to propagate the objective, and to inform the branching strategy (see Chapter 8).
- Parallel machines with Makespan ($P||Cmax$): We use the bin-packing model shown earlier in this chapter.
- Cumulative Job-Shop with Makespan: The strategy uses a simple model with cumulative resources. The branching is performed with `setTimes`.
- Resources constrained Project Scheduling Problem: We use a simple model with cumulative resources and the `setTimes` strategy to branch.

The general strategy uses the standard modeling abstractions for scheduling with global constraints for the different kinds of resources. For the objective of weighted Earliness and Tardiness, a global constraint is also introduced. The branching is done in the following order:

1. Fixing the modes of the activities with several modes.
2. Fixing the alternatives among requested resources.
3. Ordering activities needing the same machines (using the ranking procedure).
4. Ordering activities needing cumulative resources (by solving potential conflicts).
5. Fixing optimal starting times of all activities.

The LNS prototype The Large Neighborhood Search (LNS) synthesizer is able to solve the same problems as the CP one, because it is build on top of it. The CP strategies feature LNS, but it is not activated. The LNS synthesizer activates this feature, using the `Parameters` class.

The LNS schemes all fix a limit on the number of failures, and are based on a relaxation of a Partial Order Schedule (POS). They differ in how the problem is relaxed upon restart and how the limit is updated. The available schemes are the following ones:

- Relaxation on the machines: The precedences associated to all the activities requiring a subset of the machines are relaxed. The number of relaxed machines is increased or decreased if the last search was complete or not. This scheme is used for the Just-In-Time Job-Shop (see Chapter 8 for details).
- Adaptive random relaxation: Each precedence of the POS is relaxed with some given probability. The probability is increased or decreased if the last search was complete or not. This scheme is used for the strategies solving the Job-Shop and Open-Shop problems with makespan, and the RCPSP.
- Adaptive random relaxation with restart: This scheme is used in the general strategy. It follows the same pattern as the previous one but it also changes the limit on the number of failures. This limit is initially fixed to 10 failures. If there are ten searches without producing an improved solution, the limit is doubled. When a new better solution is found, the limit is reset to 10. When the limit

reaches 5000, a full restart is performed, allowing to look in other parts of the search space.

The LS prototype The LS synthesizer is currently able to solve problems in some well defined classes which are :

- Open-Shop with makespan, using a simple tabu search
- Job-Shop with makespan, using the tabu search of Dell'Amico and Trubian [DT93]
- Job-Shop with weighted tardiness, using a simulated annealing [VM05]
- Cumulative Job-Shop with makespan, using IFlat-IRelax [VM05]

The greedy prototype This one also solves specific problems. Some strategies are deterministic, while other ones are randomized. The greedy synthesizers are often used to initialize other synthesizers. Using a randomized greedy heuristic allows to start from several different initial solutions. As greedy algorithms are very fast, we run them many times and keep the best starting solution.

The existing strategies are able to solve the following problems:

- Group-Shop and its subclasses (Open-Shop and Job-Shop) with makespan (randomized algorithm)
- RCPSP (randomized algorithm)
- Job-Shop with sum of weighted tardiness (randomized algorithm)
- Job-Shop with sum of weighted earliness and tardiness (deterministic algorithm)

5.5 Extending AEON

In this section, we review how our prototype can be extended, and what are the limits of the extensions. Most of this material has been covered in other sections, but we find it useful to gather it in one place.

Extending the modeling library This is not possible without touching many parts of the system. Indeed adding a new abstraction (for instance a new kind of resource), requires to deal with this new abstraction in the internal representation of the problem, and in the analysis and classifications steps. In particular, for all existing classes of problems, we need to add the constraint that the new kind of resource, to follow the example, is not used. Finally, it is necessary to add strategies that are able to solve problems that contain the new kind of resource.

What is easy, however, is to add abstractions that can be casted directly in terms of the existing objects of the internal representation, for instance shortcuts for usual objectives.

Extending the classification Adding new features of problems is easy, as long as they can be expressed in terms of existing features. This is done using XML files that describe the constraints of a classes or label, or how to compute a numeric value (see Section3.3.3). On the contrary, if new basic characteristics are needed, it is necessary to write some COMET method to access the internal representation and retrieve the values of the new characteristics. A name of characteristic must then be associated to the result of this function such that it can be used to declaratively describe features.

Extending the synthesis The extension of the synthesis can take multiple forms that are covered in details in Section 4.2. The three main possibilities are:

- Writing a new strategy. This is made simple through the use of views to access the data of the problem to solve. It is anyway necessary to write the search algorithm, which can be tedious.
- Writing a new synthesizer, for instance to use a search technology different from existing ones. As a synthesizer maps classes to strategies, it suffices to create this mapping, once the classes are described and the strategies are written as explained above.
- Writing a composed synthesizer, that reuses the strategies of other synthesizers but combine them in any fashion. This can be written in less than 15 lines of code.

As shortly explained in Section 1.5, the whole synthesis part can be considered as the back-end of the language interpreter that would be AEON. Transforming it to a compiler would amount to replace the synthesizer and strategy classes by other ones that would write the code of a search algorithm, instead of executing it. This extension is quite large but would be easy to integrate in AEON, as it would just be a new kind of synthesizer to include.

6

EXPERIMENTAL VALIDATION

Our evaluation of AEON will be done in two parts. The first part consists in showing that AEON is able to solve standard problems efficiently. In the second part, we evaluate the time spent to classify problems in function of the size of the problem.

6.1 Evaluation on Benchmark Problems

Our goal in this section is to show that AEON is able to solve problems efficiently. For this evaluation, we work with the following problem classes:

- Job-Shop problem with makespan
- Resource-constrained project scheduling problem (RCPSP)
- Job-Shop problem with total tardiness
- Group-Shop problem with makespan
- Just-in-time Job-Shop problem

The choice of those problems responds to the following criterion:

- We want problems for which there exist established benchmarks, and that have been solved previously. Otherwise, it would be difficult to draw conclusion on the efficiency of AEON.
- We want problems with varying features. Cumulative and disjunctive problems, different objective functions, changing precedence structures.
- We limit ourselves to problem for which we have implemented search algorithms in AEON. As shown in Section 5.4.2, not all modelable problems can currently be solved within our prototype. In particular, there are no working strategies for problems with any of the following features: alternative requirements, variable processing times, multi-mode and optional activities. This is the subject of future work.

For each of the problems, we compare different searches generated by AEON, together with searches directly coded in COMET, and results of state-of-the-art techniques provided in the scientific literature.

The evaluation of the algorithms will focus on the value of the solutions found in a bounded time limit. For the comparison between algorithms, we use a measure called Relative Error (RE), which can be computed as follows. For a problem instance I, and an algorithm A, we compute their RE as:

$$RE(I, A) = 100\% * \frac{Sol(I, A) - LB(I)}{LB(I)},$$

where $Sol(I, A)$ is the value of the solution returned by A on I, and $LB(I)$ is a lower bound on the value of the optimum. We use the best known lower bounds, which may be the value of the optimal solution. A proper use of this formula assumes that all values are strictly positive, which is the case in most scheduling problems.

The RE is a way to normalize the results over instances with sometimes very different objective values. If an algorithm solves an instance to optimality, its RE on this instance is equal to 0. Larger values mean that the algorithm is farther from the optimum.

We can compute the mean RE (MRE) of an algorithm over a set of instances. This value gives a good idea of the relative performance of different algorithms on the same set of instances.

The execution of the experiments is performed on a high-throughput environment, made of 30 cores distributed on a set of Intel(R) Core(TM) 2 Quad CPU at 2.40GHz. Each algorithm is allocated one core, and the distribution is managed by Condor(R) [1]. We use a time-out for each execution set to five minutes (300 seconds). Unless otherwise stated, each algorithm is run once per instance.

6.1.1 The Job-Shop Problem with Makespan

This problem is a classical problem in the scheduling community. It consists of sequences of activities that must execute in a given order (jobs). Each activity has a fixed duration and requires one machine. The objective is to minimize the completion time of the latest activity (the makespan).

This problem has been solved with a very large number of techniques. The best ones currently known use Tabu Search (TS).

A Job-Shop model is stated in AEON in less than 30 lines of code, including the reading of the input data from a text file (see the code in Appendix B).

Experimental Setup

We compare the following approaches:

- the Constraint Programming (CP) synthesizer of AEON,
- the Large Neighborhood Search (LNS) synthesizer of AEON,

[1] `http://www.cs.wisc.edu/condor/`

- the Local Search (LS) synthesizer of AEON,
- a synthesizer chaining LS and CP,
- the greedy synthesizer of AEON repeated 100 times and keeping the best found solution (G_{100}),
- a CP model written in COMET (CP′),
- a LNS model written in COMET (LNS′),
- the results from the *i*-TSAB algorithm reported in [NS05],
- the results from the TSSA algorithm reported in [ZLRG08].

The comparison is performed on the 69 instances tested in [NS05]. They are hard instances from different benchmarks and sizes:

- TA01-10: 10 instances with 15 jobs and 15 machines, i.e. 225 activities.
- TA11-20: 10 instances with 20 jobs and 15 machines, i.e. 300 activities.
- TA21-30: 10 instances with 20 jobs and 20 machines, i.e. 400 activities.
- TA31-40: 10 instances with 30 jobs and 15 machines, i.e. 450 activities.
- TA41-50: 10 instances with 30 jobs and 20 machines, i.e. 600 activities.
- SWV01-05: 5 instances with 20 jobs and 10 machines, i.e. 200 activities.
- SWV06-10: 5 instances with 20 jobs and 15 machines, i.e. 300 activities.
- SWV11-15: 5 instances with 50 jobs and 10 machines, i.e. 500 activities.
- YN01-04: 4 instances with 20 jobs and 20 machines, i.e. 400 activities.

All instances are available from the OR-Library [Bea90]. Each algorithm we run is given a maximum of 300 seconds per instance. The *i*-TSAB algorithm has been executed on a Pentium III at 900 MHz. Its running time as reported by the authors lies between 25 and 975 seconds, from the smallest to the largest instances. The TSSA algorithm has been run on a Pentium IV3.0G, and its running time ranges between 11 seconds for the smallest instances and 910 seconds (15 minutes) for the largest ones.

Results

Figure 6.1 presents the results obtained with the different techniques. The five first ones are synthesizers from AEON. The four others are dedicated algorithms. Several analysis can be drawn from this table.

The first one is that the CP and CP′ columns are really similar (and quite bad). Their similarity is natural, as they both use the scheduling<CP> module of COMET with an equivalent model. The results are of course of bad quality as exact methods are in general not able to tackle those large scale problems (from 200 to 600 activities). The comparison between the LNS searches (generated by AEON and hand-written) shows better results for the generated approach. The main reason is that the LNS component in AEON is common to many strategies, and for this reason it includes an adaptation scheme. This scheme gives better results than a non-adaptive (but tuned by hand) LNS′.

The best results obtained with AEON are produced with the hybrid approach LS+CP. Those results are still worse than the best known search algorithms (*i*-TSAB and

	AEON synthesizers					Dedicated algorithms			
Class	CP	LNS	LS	LS+CP	G_{100}	CP′	LNS′	*i*-TSAB	TSSA
TA01-10	75.62	1.19	2.06	1.41	16.52	75.77	1.87	0.11	0.01
TA11-20	123.77	6.09	5.11	5.07	23.86	123.87	13.81	2.81	2.37
TA21-30	100.93	9.23	9.15	8.67	25.95	100.99	20.61	5.68	5.43
TA31-40	141.07	6.32	3.85	3.44	21.70	143.40	36.66	0.78	0.55
TA41-50	166.49	18.29	9.04	9.40	31.06	176.96	68.42	4.70	4.07
SWV01-05	106.97	4.64	5.47	4.81	27.47	98.78	14.97	1.01	0.78
SWV06-10	115.62	16.30	13.85	12.57	35.94	120.41	31.36	7.49	6.91
SWV11-15	123.58	24.41	4.34	4.66	32.55	105.02	63.53	0.51	-
YN01-04	99.85	12.22	8.84	9.43	23.16	106.26	21.21	5.18	6.40
Overall	118.97	9.95	6.46	6.20	25.56	119.65	29.68	2.98	2.94

Figure 6.1: Results for the Job-Shop Problem. Mean relative error by class.

TSSA) but they are better than a user-produced code written in COMET, such as the LNS′ column. A way to improve those results is to replace the current Local Search algorithm (which is based on the Tabu Search of [DT93]) by the state-of-the-art algorithms *i*-TSAB or TSSA.

6.1.2 RCPSP

The Resource-constrained Project Scheduling Problem (RCPSP, described in Section 2.1.2) is a broad class of problems covering many other special cases (such as e.g. the JSP). The problem consists of a set of activities related with simple precedence constraints, and requiring some amount of different resources. The goal is to minimize the completion time of the last activity.

This problem has been studied by many researchers. We refer to [KH06] for a survey of the best approaches.

Our AEON model for the RCPSP is 13 lines long, plus about 50 lines for the parsing of the input data (see the code in Appendix B).

Experimental Setup

We experiment with three approaches generated by AEON: CP, LNS, and a Greedy approach repeated 100 times (G_{100}). We compare those approaches with CP-based searches written in COMET (CP′ and LNS′), and the results of two other approaches:

- the best approach reported in the survey of [KH06]. This approach was introduced in [VBQ03]. No running time is reported.
- SA-LNS, introduced in [LG07], and presented in Section 2.8. They ran experiments on a Dell Latitude D620 laptop, 2 GHz, 2GB RAM, with a maximal running time of 680 seconds.

We run the algorithms on the J120 benchmark set. This set, available on the PSPLIB [KS97], is composed of 600 problems with 120 activities and 4 resources.

	AEON synthesizers			Dedicated algorithms			
Class	CP	LNS	G_{100}	CP′	LNS′	Valls	SA-LNS
J120	42.84	39.07	46.99	42.82	40.78	31.24	32.4

Figure 6.2: Results on the RCPSP benchmark. The relative error is computed with respect to the resource-relaxation lower bound.

Results

The results are presented in Figure 6.2. We compute the relative error with respect to the lower bound given by the relaxation of the resources (i.e. the makespan is the length of the longest path in the precedence graph). This is standard for RCPSP benchmarks.

As for Job-Shop, this table shows that the synthesized algorithms do as well as their plain COMET counterparts, but are not able to catch the state-of-the-art algorithms. The gain of AEON is that no search procedure has to be written by the user, while yielding the same results as hand-written models in COMET.

6.1.3 Job-Shop Problems with Total Tardiness

The Job-Shop Problem with total tardiness (JSPwT) is a Job-Shop with a different objective. In this problem, each job j has a due-date by which it should be finished $dd(j)$ and a unit tardiness cost $t(j)$. The function to minimize is $\sum_{j \in jobs} t(j) * T(j)$. From Section 2.1.1, remember that $T(j) = max(0, C(j) - dd(j))$, where $C(j)$ is the completion time of the last activity of job j.

Experimental Setup

We run our comparison on a standard set of instances, which are adapted from Job-Shop instances, by the addition of due-dates. This benchmark contains 22 instances with 10 jobs and 10 machines. The due-date of each job is equal to $1.3 * \sum_{i \in acts(j)} p(i)$, where $acts(j)$ is the set of activities of job j, and $p(i)$ is the processing time of activity i. The tardiness cost of the two first jobs is equal to 1, the two lasts to 4, and the remaining ones to 2.

The following algorithms are compared:

- the CP synthesizer of AEON,
- the LNS synthesizer of AEON,
- the LS synthesizer of AEON,
- the greedy synthesizer of AEON repeated 100 times (G_{100}),
- a synthesizer chaining 10 greedy searches and LS (G_{10}+LS),
- a CP model written in COMET (CP′),
- a LNS model written in COMET (LNS′),
- a LS model written in COMET (LS′),

	AEON synthesizers					Dedicated Algorithms				
	CP	LNS	LS	G_{100}	G_{10}+LS	CP′	LNS′	LS′	LSRW	GLS
Mean	275.56	13.01	4.26	125.26	3.65	94.99	14.72	4.17	2.05	1.27
Best	275.32	5.44	1.27	89.78	0.94	94.44	3.69	0.49	-	0.05

Figure 6.3: Relative Error on the JSPwT benchmark for the mean and best results over 10 runs. The relative error is computed with respect to the best known upper-bounds. [Kre00] only provides the average over 5 runs for LSRW.

- the results of a Large Step Random Walk (LSRW) introduced in [Kre00]. This algorithm was run on a Pentium 233MHz, with a time limit of 200 seconds.
- the results a Genetic Local Search (GLS), reported in [EMDP08]. They set a time limit of 18 seconds on a 2.8 GHz CPU with 512 MB RAM.

Each of our algorithms has been run 10 times. Our results consider the best and average over those 10 runs.

Results

The results are reported in Table 6.3. Comparing the synthesized algorithms with their COMET counterpart, the results are comparable between the LNS and LNS′, and between the LS and LS′. Notice, however, the difference between the CP and CP′ columns. This is due a different used search strategy. The CP synthesizer of AEON does not use a particular strategy for the JSPwT, and hence ranks the machines. For the COMET model, we observed that a `setTimes` strategy give better results and we used it.

In the case of the JSPwT, few advantages are obtained by the use of a greedy algorithm to initialize other algorithms (such as for G_{10}+LS). This is due to the bad performance of this greedy algorithm, that should be improved in the future. The hybrid algorithm gives nevertheless the best results among the synthesized algorithms.

Comparing our results with the state-of-the-art, we see that AEON is not able to match them, while providing good quality results (less than 1% from the best known solutions for the best solutions found by G_{10}+LS). It is worth noting that the LS (and LS′) algorithm is the one presented in [VM05] for the JSPwT, which was inspired by the LSRW algorithm of [Kre00]. The difference in performance might be imputed to implementation differences, rather than to differing approaches.

6.1.4 Group-Shop Problems

The Group-Shop Problem (GSP) is an hybrid problem between the Job-Shop and Open-Shop Problems. It is composed of a set of jobs, each composed of a number of activities. Each activity has a given processing time and requires a machine. The particularity of the GSP is that each job is divided in groups of activities. There is a fixed order on the groups of a job but not inside the groups. All the activities of a group must be completed before the next group starts. The activities of the same group

can be executed in any order, but not at the same time. The Group-Shop boils down to Job-Shop when the groups contain one activity each, and to the Open-Shop when there is only one group per job. The objective is the usual makespan minimization.

This problem has been less studied than the three previous ones but it gained attention, as the importance of generalizing successful algorithms for the Job-Shop and Open-Shop appeared.

The model of the GSP in AEON is around 40 lines, including the parsing of the instance data (see the code in Appendix B).

Experimental Setup

The standard benchmark for this problem contains 41 problems:

- ft10: 10 instances with 10 jobs of 10 activities and 10 machines (10 by 10), with a uniform group size ranging from 1 to 10.
- la38: 15 instances 15 by 15, with group size from 1 to 15.
- abz7: 15 instances 20 by 15, with group size from 1 to 15.
- whizzkids97 (whiz): 1 problem with 197 activities, 20 jobs and 15 machines, with various group sizes.

We compared the performance of the following algorithms:

- the CP synthesizer of AEON,
- the LNS synthesizer of AEON,
- the greedy synthesizer of AEON repeated 100 times (G_{100}),
- a synthesizer chaining 10 greedy searches and CP (G_{10}+CP),
- a synthesizer chaining 10 greedy searches and LNS (G_{10}+LNS),
- a CP model written in COMET (CP′),
- a LNS model written in COMET (LNS′),
- the results from the Tabu Search (TS) reported in [LON05],
- the results from the Ant Colony Optimization (ACO) reported in [BS04].

As already mentioned, our algorithms are given 300 seconds per instance. The algorithms from [LON05] and [BS04] had no explicit limit on the running times. Their running times range from less than a second to thousands of seconds (up to 684 for TS, up to 1158 for ACO). TS was run on a Intel Pentium IV 1.8 GHz CPU, with 256 Mb RAM, and ACO was run on a AMD Athlon 1.1 GHz CPU. Each of our algorithms have been run only once. For a fair comparison, we took the best given results for the two other algorithms (and not the average).

Results

The results are presented in Figure 6.4. As for the previous problems, we can see that CP is not able to present good results. However, the LNS exhibits very good results, especially when given a good initial solution provided by a simple greedy

	AEON synthesizers					Dedicated Algorithms			
Class (nb)	CP	LNS	G_{100}	G_{10}+CP	G_{10}+LNS	CP′	LNS′	TS	ACO
ft10 (10)	12.62	0.27	9.10	7.39	0.00	13.98	0.16	0.26	0.83
la38 (15)	68.26	0.98	8.67	10.83	0.75	57.67	2.53	0.45	1.14
abz7 (15)	75.34	1.13	4.52	6.75	0.74	81.96	2.09	0.11	0.18
whiz (1)	87.63	6.40	23.67	29.64	5.76	72.07	4.05	0.00	3.62
All (41)	57.75	0.99	7.62	8.96	0.68	56.25	1.83	0.27	0.77

Figure 6.4: Relative Error on the GSS benchmark. The relative error is computed with respect to the best known upper-bounds

algorithm (column G_{10}+LNS). In particular, this approach is able to match state-of-the-art algorithms, and even beat them on the smaller instances (class ft10) where it improved several best known upper bounds.

As the average relative error values are quite close, we perform a Wilcoxon signed-rank test to compare G_{10}+LNS with TS and ACO. This test is a non-parametric statistical hypothesis test. Our null hypothesis are they have the same performance in terms of Relative Error. According to this test, and with a confidence of 5% we can claim that TS and ACO does not have the same performance (V = 24, p-value = 0.0025). On the contrary, we cannot claim that TS and G_{10}+LNS have significantly different performances (V = 60, p-value = 0.093), nor have ACO and G_{10}+LNS (V = 176, p-value = 0.71).

On the Group-Shop benchmark, the G_{10}+LNS approach reaches the best known solution for 27 instances out of the 41. The 14 remaining instances are mainly instances with a structure closer to the Job-Shop (with smaller groups and more precedences). This calls for an even more discriminative classification of these problems to solve the Job-Shop-like problems differently.

Comparing the results of the generated searches with their hand-written counterparts (CP vs. CP′ and LNS vs. LNS′) shows one more time that on average they perform the same.

6.1.5 Just-In-Time Job-Shop Problem

The experimental results for the problem of Just-In-Time Job-Shop Problem (JITJSP) are reported in Chapter 8, that introduces a new global constraint for the Just-In-Time objective. We report here the Mean Relative Error for the best approaches that are:

- a Constraint Programming approach using the new global constraint and a greedy local search to improve found solutions (CP),
- a Large Neighborhood Search embedding the previous search (LNS),
- the results of a simple local search given in [BFS08],
- the results of the SA-LNS approach of [LG07].

The results are reported in Figure 6.5. The relative error is computed with respect to the best lower bounds given in [BFS08], or the optimal solution when proved by

	AEON synthesizers		Dedicated algorithms	
Size	CP+ls	LNS	BF&S	SALNS
10x2	3.03	8.79	2.80	-
15x2	10.06	10.98	10.11	4.46
20x2	18.09	18.73	12.32	7.66
10x5	32.07	27.09	33.55	-
15x5	37.49	37.32	46.60	36.67
20x5	39.92	42.54	34.50	28.30
10x10	21.75	19.06	21.48	-
15x10	56.67	51.30	67.69	58.84
20x10	81.92	78.53	84.44	81.06
All	33.44	32.70	34.83	-

Figure 6.5: Relative Error on the JITJSP.

our "CP+ls" algorithm (see Chapter 8 for the details). We see there that our approach is competitive with state-of-the-art searches. It is also worth noting that the AEON model expresses the objective without introducing a global constraint. Writing the same model in CP would lead to a very inefficient model (leading to a MRE around 400%).

6.2 Classification Time Evaluation

An important part of the usability of AEON is that the simplification and classification should not take too long. We show here that it is the case, with a total classification time around 6 seconds for instances with 600 activities.

During our experiments in the previous section, we registered the total CPU time between the start of the program and the creation of the search algorithm. This comprises the setup of the synthesizer, the creation of the scheduling problem, its transformation to the internal form, its analysis and the whole classification. We call it in short the C-time (standing for classification time). For each instance, we registered the number of activities in the model, the number of precedences in the model, and the number of resources of the model. These three values give a good idea of the size of the problem.

We observe how the C-time increases with respect to the three indicators of problem size. First, Figure 6.6 shows the C-time as a function of the number activities. We only plotted the time for instances for which the number of resources is between 5 and 20. They are instances from the Job-Shop, Group-Shop and Just-In-Time Job-Shop Benchmarks. It is clear that the number of activities is the main factor that affects the C-time. The C-time increases with a very slow cubic curve in function of the number of activities. The polynomial found by regression analysis is the following (where x is the number of activities, and y the C-time in seconds): $y = -3.41 + 2.87 * x + 1.01e-2 * x^2 + 4.59e-6 * x^3$. The cubic degree comes from the transitive closure and reduction algorithms, that have a complexity in $\mathcal{O}(n^3)$.

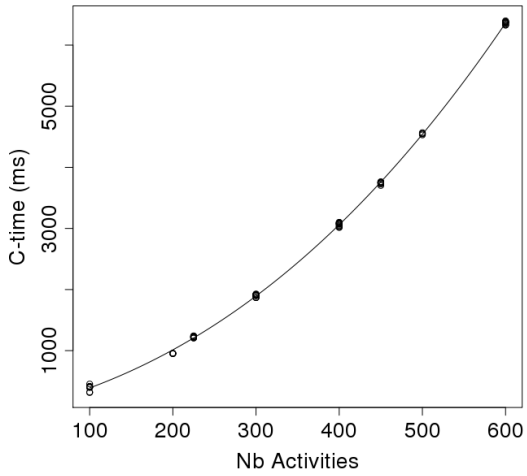


Figure 6.6: C-time (in milliseconds) as a function of the number of activities.

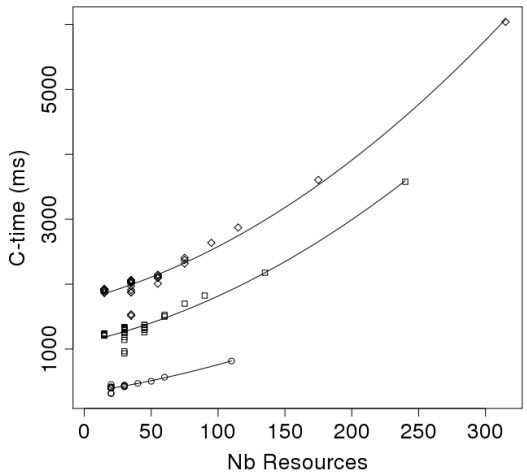


Figure 6.7: C-time (in milliseconds) as a function of the number of resources, for problems with 100, 225 and 300 activities.

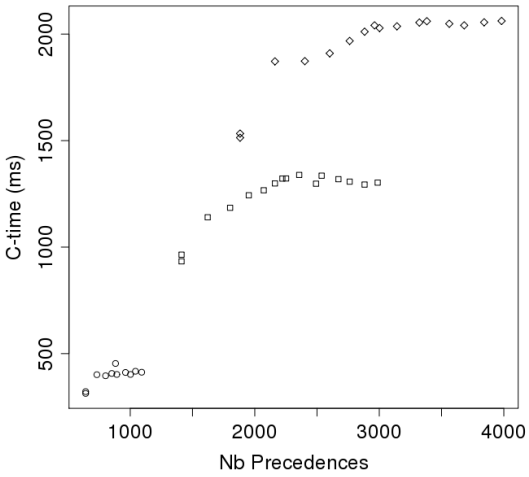


Figure 6.8: C-time (in milliseconds) as a function of the number of precedences.

The number of activities is not the only influencing factor. The second most important factor is the number of resources, according to Figure 6.7. This plot shows the evolution of the C-time with respect to the number of resources, for fixed number of activities. The C-time follows a quadratic curve as a function of the number of resources. Indeed, all resources are compared with each other trying to merge them (see Section 3.2.3). By regression analysis, we found that the polynomials fitting best the data are (where x is the number of resources, and y the C-time in seconds):

- $y = 298 + 4.55 * x$ for 100 activities,
- $y = 1104 + 4.79 * x + 2.32e - 2 * x^2$ for 225 activities,
- $y = 1766 + 5.54 * x + 2.59e - 2 * x^2$ for 300 activities.

The value of the constant term corresponds approximately to the effect of the number of activities.

As shown on Figure 6.8, the number of precedences has little impact on the C-time. We highlight in this plot three sets of problems that have the same number of activities and resources. They respectively contain problems with 100 activities and 20 resources, 225 activities and 30 resources, and 300 activities and 35 resources. This plot shows well that for a same number of activities and resources, the C-time does not increase much with the number of precedences.

6.3 Experiments Conclusion

We can draw several conclusions from the experiments:

- The time spent by AEON to analyze and classify an instance is very small. It takes around 6 seconds for instances with 600 activities, and the curve is a very light cubic curve. This is really acceptable, as the running time of algorithms to solve so large instances is counted in minutes or even in tens of minutes. The main factors influencing the classification time are the number of activities and the number of resources.

- Algorithms runs by AEON are comparable to their plain COMET equivalent. The main advantage of the AEON algorithms is that the user does not have to write the search algorithm. In particular, the user only has to write one model and can experiment with different search paradigms (a complete but slow algorithm, or a heuristic and fast algorithm?). It allows also one to profit of hybrid algorithms at no cost. Finally, the user does not have to be experimented in writing a model, as it is simplified by AEON during the analysis.

- Algorithms runs by AEON are not able to match state-of-the-art algorithms for extensively studied problems such as the Job-Shop or the RCPSP. On the contrary, it is able to get very good results relatively to existing methods for problems that are not as pure, as the Group-Shop Problem. For this problem, AEON was even able to improve the best known upper-bounds of some instances.

- It is possible to extend the synthesizers of AEON with state-of-the-art algorithms. This work is facilitated by the views to access the data of the problem, but also by the underlying layers of COMET providing modules for CP and LS. In the same direction, it will be necessary to extend the synthesizers with strategies for problems not yet covered by the existing strategies.

7

POSITION-BASED MACHINE PROPAGATOR

In the present chapter, we propose a propagator for the One-Machine Non-preemptive Problem that exploits information given by the position of the tasks. We introduce the motivation for this propagator by means of the Open-Shop Problem (OSP, see Section 2.1.2) but this propagator can be used as well for many other disjunctive problems.

The feasibility version of the OSP can be stated as the conjunction of smaller problems called One Machine Non-preemptive Problem (1NP). This problem aims at scheduling a set of tasks on a machine such that there is only one not interruptible task processed at a time. Each task is given a duration, an earliest and a latest starting time. To model the OSP, it is sufficient to define a 1NP for every machine and every job (and to link them with the makespan). Indeed, jobs and machines in the OSP have the same behavior: No two tasks associated with a same job or a same machine can be processed simultaneously. The 1NP is also the basis for other disjunctive problems such as the Job-Shop Problem.

Formally, the 1NP is defined as follows: T is the set of tasks that must be processed and N is its cardinality. For each task $t \in T$, the duration $d(t)$, an initial $est(t)$ and an initial $lst(t)$ are given. They denote respectively the duration, earliest and latest starting times of the task t. The problem is to find for each task t, its starting time $S(t)$ such that $est(t) \leq S(t) \leq lst(t)$ without task overlap, that is, $\forall t_1, t_2 \in T, S(t_1) + d(t_1) \leq S(t_2) \vee S(t_2) + d(t_2) \leq S(t_1)$.

7.1 Related Work

This problem fits very well in the framework of Constraints Programming (CP). Propagators have been developed to remove inconsistent values from the domains as early as possible in order to reduce the size of the search tree. Prominent techniques

are Edge-Finding (EF) and Not-First-Not-Last (NFNL). Edge-Finding [CP89, AC91, CP94, CL94] consists in testing whether a particular task must start before or after a set of tasks. It can be implemented with a time complexity of $\mathcal{O}(N \log N)$ where N is the number of tasks on one machine or one job. Not-First-Not-Last [CP90, DPPH01, Vil04] checks if a task can be the first or the last among a set of tasks. Its smallest time complexity is $\mathcal{O}(N \log N)$.

Shaving [CP90, DPPH01, MS96] is an orthogonal technique that performs well in practice for solving the OSP. It consists in iteratively assigning to a variable its possible values and checking if this assignment leads to inconsistency. In that case, the value is removed from the domain of the variable. Every constraint can be propagated to check consistency until the fixpoint is observed. Since it can be costly to reach this fixpoint, simpler propagators are used. For instance, in [DPPH01], only Edge-Finding is used to look for inconsistencies. Even so, shaving is costly because the size of the domain of the starting time variables can be huge. For this reason, shaving in OSP usually considers only the bounds of the domain.

The idea of using the positions has already been used successfully in [Zho97], [NLP98] and [Wol05]. The first work uses the positions as permutation variables in a sorting constraint. An extension of Edge-Finding is also applied. Secondly, [NLP98] proposes a possible way to decide if a task can start at some position looking at the number of other tasks that can come before and after this task. Finally, [Wol05] extends this idea proposing tighter bounds with an algorithm running in $\mathcal{O}(N^3)$

We present an alternative way to use the position of the tasks based on the idea of shaving. For each possible position of a task, lower and upper bounds on the possible starting time of the task are computed using the duration and the domain of the variables of the tasks in the same job or machine. The resulting propagator is applied on the tasks that are part of the same job or machine with a time complexity of $\mathcal{O}(N^2 \log N)$, where N is the number of tasks that are part of the job or that must be processed on the machine. This propagator permits additional pruning that is not performed by NFNL and EF, and permits to detect about 14 % extra inconsistent nodes of the search tree on a standard benchmark [GP99].

The next section explains the problem under interest and its mapping in CP. Section 7.3 presents the new propagator and Section 7.4 describes experimental results assessing the pruning efficiency of the technique. In the last section, conclusions are drawn as well as directions for future work.

7.2 The One Machine Non-preemptive Problem

7.2.1 Problem Modeling in CP

To model the 1NP, several variables are defined for each task $t \in T$. An integer variable $S(t)$ represents the starting time of the task t. Its domain ranges from the earliest starting time to the latest starting time of the task ($dom(S(t)) = [est(t), lst(t)]$). To model the relative order between the tasks, a set variable $B(t)$ represents the set of tasks that come before the task t. Initially, no task is known to come before

t, and all the tasks might come before t. So, its initial domain is $dom(B(t)) = [\emptyset, \{u|u \in T, u \neq t\}]$. The symbol $\overline{B}(t)$ (resp. $\underline{B}(t)$) represents the upper (resp. lower) bound of the variable $B(t)$. Furthermore, an additional variable $P(t)$ represents explicitly the absolute order (or the position) of the tasks in the machine. The domain of this variable ranges from 0 to $N-1$ with N being the number of tasks to be processed. The link between the relative and absolute orders of the tasks is that $P(t)$ represents the size of $B(t)$.

The starting time and the relative ordering between tasks are commonly used in the modeling of disjunctive scheduling. The use of an absolute order comes from [Zho97] where the author solves the Job-Shop Problem fixing the permutations of task orders. In their proposed formulation, a variable is defined for the starting time of each task, a variable for the starting time of the task in each position and a variable for the position of each task. Those three sets of variables are linked with a sorting constraint and various reduction rules are then defined. As an initial approach, we chose here for simplicity not to use the variables for the starting time of the task in each position.

7.2.2 Constraints

With three complementary representations, the 1NP can be equivalently expressed using anyone of the three following sets of constraints stating that two tasks cannot be processed at the same time.

1. $\forall t_1, t_2 \in T, \quad (S(t_1) + d(t_1) \leq S(t_2)) \quad \vee \quad (S(t_2) + d(t_2) \leq S(t_1))$
2. $\forall t_1, t_2 \in T, \quad (t_1 \in B(t_2)) \quad \vee \quad (t_2 \in B(t_1))$
3. $\forall t_1, t_2 \in T, \quad (P(t_1) < P(t_2)) \quad \vee \quad (P(t_2) < P(t_1))$

Our model uses the three sets of constraints to speed-up propagation. Additionally, the following channeling constraints ensure the consistency between variables of each representation. The position of a task t is the number of tasks that come before t ($|B(t)| = P(t)$). Also, a task t_1 ends before another task t_2 starts if and only if the position of t_1 is less than the position of t_2 ($S(t_1) + d(t_1) \leq S(t_2) \Leftrightarrow t_1 \in B(t_2) \Leftrightarrow P(t_1) < P(t_2)$).

In addition to these basic constraints, other redundant constraints can be defined. First, if t_1 comes before t_2, every task that comes before t_1 comes also before t_2 ($t_1 \in B(t_2) \Leftrightarrow B(t_1) \subset B(t_2)$). An AllDifferent constraint is also defined on the position variables (*alldiff*$(\{P(t) : t \in T\})$), because two tasks cannot have the same order of execution.

This last constraint is a first example of global constraint. Global constraints take into account more than two tasks at a time. NFNL and EF are also such global propagators that allow a much better pruning than the basic constraints. However, NFNL and EF do not use the information given by the position of the tasks to derive their information. This work shows how to use this additional information.

7.3 The Propagator

The main idea of the new propagator is to apply shaving on the position variables. Commonly, shaving is applied on the starting time variables and only on their bounds because of the size of their domains. On the contrary, the domain of the position variables is rather small and could be shaved in a reasonable time. To test if the task can be scheduled in a particular position, we compute bounds on its earliest and latest starting time under this assumption. If the resulting range does not intersect the domain of $S(t)$, the task cannot be scheduled in that position. Furthermore, shaving $P(t)$ permits also to reduce the domain of $S(t)$ to the union of the ranges computed for every position. Following this scheme, two issues need to be addressed. Firstly, the way to use the bounds on the task starting time to reduce the domains of the variables (Section 7.3.1). Secondly, the approximations used to compute ranges as tight as possible (Section 7.3.2). Notice that our approach of shaving is local to this propagator.

Let us first introduce some additional notations. As $est(t)$ represents the earliest starting time of a task t, $ect(t)$ will denote its earliest completion time. Those values are linked by $ect(t) = est(t) + d(t)$. The same quantities can be defined for set of tasks. If U is a non-empty subset of T, $d(U)$ is the sum of the durations of the tasks in U and $est(U)$ is the earliest starting time of the set of tasks U. It is equal to the earliest starting time of any tasks in U ($est(U) = \min_{t \in U} est(t)$). The dual quantity $ect(U)$ is the earliest completion time of the set U, the time when every task in U is finished. This last quantity cannot be computed easily but several lower bounds are known. Especially, the maximum, among every subset U' of U, of the sum of the earliest starting time of U' and the duration of U' will be used in this work to approximate $ect(U)$ (Equation 7.1). This is only a bound because it does not take into account the latest starting time of the tasks. We denote it $b\_ect(U)$.

$$b\_ect(U) = \max_{\emptyset \neq U' \subseteq U} (est(U') + d(U')) \tag{7.1}$$

7.3.1 Shaving on position variables

Shaving enumerates every possible value of $P(t)$. Under the assumption that the position $P(t)$ of a task t takes a particular value p of its domain, the possible starting time of t belongs to an interval $[est(t,p), lst(t,p)]$ where $est(t,p)$ and $lst(t,p)$ denote respectively the earliest and latest possible starting times when t is in position p.

The value $est(t,p)$ is related with $ect(B(t),p)$ that is the earliest time when p tasks among those in $\overline{B}(t)$ have been processed and when all the tasks in $\underline{B}(t)$ have been processed. Indeed, in position p, the task t cannot start before that p tasks among those that can come before t have been processed. Furthermore, t cannot start before the tasks that must come before are completed. This leads to the relation

$$est(t,p) = \max\left(ect(B(t),p), est(t)\right).$$

In this formula, $ect(B(t),p)$ cannot be computed exactly with a reasonable complexity. We propose however to compute a lower bound as tight as possible. Section 7.3.2

details how to approximate the value of $ect(B(t), p)$. A very similar reasoning, not detailed here, can be made to approximate $lst(t, p)$.

Once the ranges $[est(t, p), lst(t, p)]$ have been computed for each $p \in P(t)$, the domain of $P(t)$ and $S(t)$ can be reduced with two simple rules:

$$\forall p \in dom(P(t)) : ([est(t, p), lst(t, p)] \cap dom(S(t)) = \emptyset) \Rightarrow P(t) \neq p \tag{7.2}$$

$$dom(S(t)) := dom(S(t)) \cap (\cup_{p \in dom(P(t))}[est(t, p), lst(t, p)]) \tag{7.3}$$

The first rule removes from the domain of $P(t)$ the values p for which there is no valid starting time, i.e. when the range $[est(t, p), lst(t, p)]$ is empty or when it does not intersect with the domain of $S(t)$. The second rule restricts the domain of $S(t)$ to be included in the union of the computed ranges. Alternatively rule (7.3) could only reduce the bounds of the domain of $S(t)$, while ensuring that $S(t)$ remains a single interval. The latter is standard in scheduling. $S(t)$ must then be greater than the least value among the $est(t, p)$ for valid p's and less than the greatest value among the $lst(t, p)$ for valid p's.

$$dom(S(t)) := [\min_{p \in dom(P(t))} (est(t, p)) , \max_{p \in dom(P(t))} (lst(t, p))] \tag{7.4}$$

Experiments will consider the two versions of the reduction of $S(t)$. The reduction of $S(t)$ (using rule (7.4)) and $P(t)$ can be done with a time complexity of $\mathcal{O}(N)$ where N is the number of tasks to be processed on the machine, thus an upper bound on the size of the domain of $P(t)$.

7.3.2 Bounding the earliest completion time of a task subset

This section presents the approximation of $ect(B(t), p)$ that is useful to evaluate $est(t, p)$. The algorithm to compute $lst(t, p)$ is similar but is not exposed. In order to compute a lower bound of $ect(B(t), p)$, we compute the minimum of the earliest completion time over all the sets U of cardinality p that are superset of $\underline{B}(t)$ and subset of $\overline{B}(t)$. In the following, $b\_ect(B(t), p)$ will denote the lower bound of $ect(B(t), p)$. This is a lower bound because it makes use of $b\_ect(U)$ which is a lower bound itself.

$$b\_ect(B(t), p) = \min_{U} (b\_ect(U)) \tag{7.5}$$

$$\text{where } |U| \geq p \text{ and } \underline{B}(t) \subseteq U \subseteq \overline{B}(t)$$

Interestingly, this lower bound can be computed efficiently using rules similar to the ones in the Jackson Preemptive Schedule [Jac56] for computing the earliest ending time of a set of task supposing preemption. Our algorithm also allows preemption for the tasks but does not take into account the latest starting time of the tasks. Instead, the duration of the tasks is considered to schedule a subset of tasks of fixed size as soon as possible in a preemptive way. It is done respecting the following precedence rules:

- Whenever a task t is available and the machine is free, process t.
- When a task t_1 becomes available during the processing of another task t_2 and the remaining processing time of t_1 is less than the remaining processing time of t_2, stop t_2 and start processing t_1.
- When a task t_1 becomes available during the processing of another task t_2, such that $t_1 \in \underline{B}(t)$ and $t_2 \notin \underline{B}(t)$, stop t_2 and start t_1.

The value $b\_ect(B(t), p)$ is obtained when every tasks in $\underline{B}(t)$ have been processed and at least p tasks in $\overline{B}(t)$ have been processed.

An important property is that, although the algorithm supposes the tasks to be interruptible, the resulting quantities correspond exactly to the ones given by equation (7.5) where no preemption is supposed. Indeed, it is possible to merge the different parts of the completed tasks following the order of their starting times. The result is a non-preemptive schedule of the set of tasks. Preemption is not used here as a relaxation but just as a way to ease the computation. The computed value of $b\_ect(B(t), p)$ is however a relaxation of the exact value because the latest possible starting times of the tasks are not considered.

Moreover a single run of the above algorithm gives the value $b\_ect(B(t), p)$ for every p. Indeed, it suffices to remember the successive times when a task ends to have the $b\_ect(B(t), p)$ value for the successive values of p.

A pseudo-code of the algorithm is presented in Algorithm 7.1. The algorithm uses two priority queues. The first (Q1) sorts the tasks in order of earliest starting time. It permits to put in the second priority queue (Q2) only the available tasks at a particular time (lines 9-14). Q2 sorts the tasks in ascending order of remaining duration. When a task is popped from Q2, two situations arise. Either it can be processed before a new task is available and the time when it ends is recorded (lines 15-21). Or the task must be interrupted to check if a newly available task could not end earlier (lines 23-26).

For simplicity, the outlined algorithm is a shortened version where the fact that some tasks are part of $\underline{B}(t)$ is not considered. Taking it into account can be done simply using a penalty in the second priority queue to ensure that those tasks are chosen first. Two parallel queues can also be used and the one containing the tasks in $\underline{B}(t)$ is emptied first. Additionally a counter must be used to record when all mandatory tasks have been processed.

The time complexity of the algorithm is $\mathcal{O}(n \log n)$ with $n = |\overline{B}(t)|$ which in the worst case is equal to $N-1$ (N is the number of tasks that must be processed). Indeed, the operation put() and pop() of the priority queues can be implemented in $\mathcal{O}(\log n)$. There are exactly n tasks that are put in Q1 (lines 5-7) and at most $2n$ tasks that are put in Q2 because there are exactly n tasks that can be extracted from Q1 (lines 9-14) and at most n reinsertions of task due to interruption (lines 22-26).

Example 7.1 To show the computation of $b\_ect(B(t), p)$, let us suppose the following tasks:

- t_0 which is the task under consideration; $dom(B(t_0)) = [\{t_4\}, \{t_1, t_2, t_3, t_4\}]$ and $dom(P(t_0)) = [1, 4]$

Input: B : the set of tasks
D : vector of the duration of the tasks
EST : vector of the est of the tasks
Output: ECT : vector of the $b\_ect(B(t), p)$ for each position p

```
1  Q1 := new PriorityQueue();
2  Q2 := new PriorityQueue();
3  time := 0;
4  p := 0;
5  forall t in B do
6     RD(t) := D(t) //RD is the remaining duration;
7     Q1.put(t,EST(t));
8  while not Q1.empty() do
9     t := Q1.pop();
10    time := EST(t);
11    Q2.put(t,RD(t));
12    while not Q1.empty() and EST(Q1.top()) = time do
13       t := Q1.pop();
14       Q2.put(t,RD(t));
15    while not Q2.empty() and
16    (Q1.empty() or time + RD(Q2.top()) < EST(Q1.top())) do
17       t := Q2.pop();
18       time := time +RD(t);
19       RD(t) := 0;
20       p := p+1;
21       ECT(p) := time;
22    if not Q2.empty() then
23       t := Q2.pop();
24       RD(t) := RD(t) + time - EST(Q1.top());
25       Q2.push(t,RD(t));
26       time := EST(Q1.top());
27 return ECT
```

Figure 7.1: Simplified Algorithm to Compute $b\_ect(B(t), p)$

- t_1 with $est(t_1) = 0$ and $d(t_1) = 5$.
- t_2 with $est(t_2) = 1$ and $d(t_2) = 3$.
- t_3 with $est(t_3) = 2$ and $d(t_3) = 1$.
- t_4 with $est(t_4) = 3$ and $d(t_4) = 3$.

Following a chronological order, t_1 is scheduled first, starting at the time 0. On time 1, t_2 is available and as its duration ($d(t_2) = 2$) is shorter than the remaining duration of t_1 ($5 - 1 = 4$), t_1 is stopped and t_2 is started. On time 2, t_2 is interrupted to let process t_3 whose duration is shorter than its remaining duration ($3-1 = 2 > 1$). On time 3, t_3 has been fully processed. Tasks t_1, t_2 and t_4 are available but t_4 is chosen as it is the only mandatory task among them. Indeed, by definition of $dom(B(t))$, t_4 is the only task which must be performed before t_0. This task is run for 3 units of time. When it is finished, t_2 is run before t_1 as its remaining duration is less than the remaining duration of t_1. After two more units of time, t_2 is fully processed and t_1 is processed until time 12. The next table gives the processing times of each task in a preemptive way.

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Task	t_1	t_2	t_3	t_4			t_2		t_1				

Recording the values when tasks are fully processed, we obtain the following values:

- $b\_ect(B(t_0), 1) = b\_ect(B(t_0), 2) = 6$. Indeed, the mandatory task (t_4) was only finished in second position.
- $b\_ect(B(t_0), 3) = 8$
- $b\_ect(B(t_0), 4) = 12$

Although the computation interrupts several tasks, the obtained bounds correspond to non-preemptive schedules (as expected by equation (7.5)). The table below shows the reordering for each position.

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
$p = 1$				t_4									
$p = 2$			t_3	t_4									
$p = 3$		t_2			t_3	t_4							
$p = 4$	t_1						t_2		t_3	t_4			

For instance, with the 4 tasks being scheduled, it is possible to run t_1 from time 0 until time 5 where t_2 is run until time 8. At time 8, t_3 is started for 1 time unit and afterward t_4 is being run until the time 12 which corresponds to the computed value. □

Example 7.2 Figure 7.2 presents a small example where the new propagator permits to remove inconsistent values. In this example, there are five tasks to be processed. Their respective domains and duration are the following.

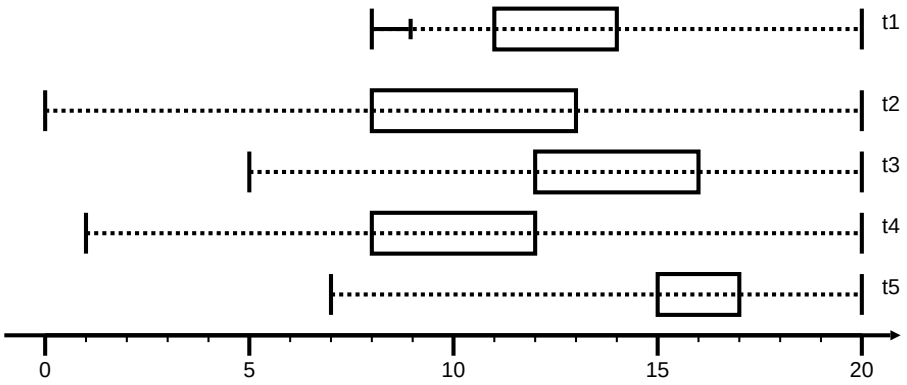


Figure 7.2: Example of reduction, see Example 2 for details

- $d(t_1) = 3$ and $dom(S(t_1)) = [8, 17]$
- $d(t_2) = 5$ and $dom(S(t_2)) = [0, 15]$
- $d(t_3) = 4$ and $dom(S(t_3)) = [5, 16]$
- $d(t_4) = 4$ and $dom(S(t_4)) = [1, 16]$
- $d(t_5) = 2$ and $dom(S(t_5)) = [7, 18]$

Applying NFNL or EF on this set of tasks does not reduce any domain of the starting time variables. However, our propagator allows to remove the value 8 from the domain of $S(t_1)$. Using the algorithm to compute the earliest and latest possible starting time of t_1 in each position, the obtained values are given next.

- $est(t_1, 0) = 8$ and $lst(t_1, 0) = 2$
- $est(t_1, 1) = 8$ and $lst(t_1, 1) = 7$
- $est(t_1, 2) = 9$ and $lst(t_1, 2) = 11$
- $est(t_1, 3) = 11$ and $lst(t_1, 3) = 15$
- $est(t_1, 4) = 15$ and $lst(t_1, 4) = 17$

From those values, it can be derived that t_1 cannot be processed in position 0 or 1. Thus the domain of its starting time can be reduced to the union of the ranges defined in position 2, 3 and 4, resulting in $dom(S(t_1)) = [9, 17]$. In comparison with the initial domain, the value 8 has been removed. □

The computing of $est(t, p)$ and $lst(t, p)$ for each $p \in dom(P(t))$ is done in $\mathcal{O}(N \log N)$ with N the number of tasks. The reduction of the domains can be done in $\mathcal{O}(N)$. The time complexity of the whole reduction algorithm for a task t is thus $\mathcal{O}(N \log N)$. This yields a total complexity of $\mathcal{O}(N^2 \log N)$ for one pass of our reduction algorithm, as there are N tasks to consider. In comparison, the well-known techniques NFNL and EF can be both implemented to run with a time complexity of $\mathcal{O}(N \log N)$.

7.4 Experiments

To assess the practical usefulness of the new propagator, we implemented it in the open constraint environment Gecode [Gec06]. Two versions of the propagator have been written. The first that we will refer to as PS (standing for Position Shaving) may remove values inside the domains of the starting time variables, while the second, PSB (for Position Shaving with Bounds reduction), is limited to reduce the bounds of the starting time variables. We implemented also the NFNL and EF techniques following the algorithms described in [BLN01]. Note that the implementations of EF and NFNL described in that book run in $\mathcal{O}(N^2)$ but they use much simpler data structures than the theoretically most efficient algorithm described respectively in [CP94] and [Vil04]. Finally, we modeled the Open-Shop Problem as described in the first section with the NFNL, EF and PS or PSB propagators and the AllDifferent constraint. PS and PSB are never used together as they are two versions of the same propagator. Concerning the branching, we applied a simple heuristic that uses the position variables. It orders the tasks in the machine before ordering them in the jobs. Among the tasks whose position is not fixed, it chooses the task for which there is the smallest number of remaining possible positions. In case of tie, the shortest task is chosen. The value-heuristic chooses the smallest value in the position variable.

Our tests have been run using the instances of the Guéret and Prins benchmark [GP99]. It is composed of 80 square problems, i.e. the number of jobs and machines are equal. There are 10 instances for each size ranging from 3x3 tasks to 10x10 tasks. Every runs have been performed on an Intel Xeon 3 Ghz with 512 KB of cache.

The first experiment consists in observing the total runtime and the size of the search tree to solve each instance of the benchmark, using different combinations of propagators. The running time is limited to one hour for each instance. The results are presented in Tables 7.1 and 7.2. Table 7.1 gives the number of solved instances and the average number of nodes in the search tree. The mean is computed over the instances commonly solved whenever the number of solved instances differs (only for problem size 7x7). In table 7.2, the same scheme is used but the mean running time is presented instead of the size of the search tree. The running time is given in seconds.

In the two tables, columns 2 and 3 present the results when only PS is used but nor EF neither NFNL. Columns 4 and 5 presents the results when only PSB is used. In the third setting (columns 6-7), NFNL and EF are activated but not PS, nor PSB. In the columns 8-9 and 10-11, NFNL and EF are used in conjunction respectively with PS and with PSB.

Whenever NFNL and EF are used, the same total number of instances are solved with or without our new propagator. However, the solved instances are not always the same. From the two first settings, it can be concluded that the new propagator is not able to solve hard problems alone. In conjunction with NFNL and EF, Table 7.1 shows that PS and PSB are able to reduce the size of search tree, sometimes substantially, as it is the case for the unique solved instance of size 8x8. Concerning the size 6x6, surprisingly the mean size is greater when using PS and PSB. Looking at the detail for each instance of this size, it appears that only the first instance(GP06-01) has a greater

Table 7.1: Number of solved instances and mean size of the search tree

	PS		PSB		NFNL+EF		PS+NFNL+EF		PSB+NFNL+EF	
Size	Solved	Nodes	Solved	Nodes	Solved	Nodes	Solved	Nodes	Solved	Nodes
3x3	10	39	10	38	10	38	10	39	10	38
4x4	10	128	10	127	10	134	10	127	10	126
5x5	10	451	10	456	10	371	10	369	10	373
6x6	10	3483	10	3896	10	2612	10	3402	10	3816
7x7	3	-	3	-	7	280914	8	208571	8	208582
8x8	0	-	0	-	1	120156	1	12953	1	12929
9x9	0	-	0	-	1	747146	0	-	0	-
10x10	0	-	0	-	0	-	0	-	0	-
Tot	43		43		49		49		49	

Table 7.2: Number of solved instances and mean running time in seconds

	PS		PSB		NFNL+EF		PS+NFNL+EF		PSB+NFNL+EF	
Size	Solved	Time	Solved	Time	Solved	Time	Solved	Time	Solved	Time
3x3	10	0.008	10	0.008	10	0.006	10	0.01	10	0.008
4x4	10	0.075	10	0.047	10	0.054	10	0.069	10	0.068
5x5	10	0.38	10	0.32	10	0.19	10	0.36	10	0.32
6x6	10	3.9	10	3.5	10	1.9	10	4.3	10	3.9
7x7	3	-	3	-	7	338	8	496	8	432
8x8	0	-	0	-	1	106	1	43	1	37
9x9	0	-	0	-	1	1708	0	-	0	-
10x10	0	-	0	-	0	-	0	-	0	-
Tot	43		43		49		49		49	

Table 7.3: Additional Pruning and time spent with PS and PSB(in %)

	Red. S(t)		Red. B(t)		Red. P(t)		Fails		Time	
Size	PS	PSB	PS	PSB	PS	PSB	PS	PSB	PS	PSB
3	7.6	1.5	0.3	0.3	5.7	3.6	0	0	-	-
4	13.6	5.6	7.0	7.4	14.2	12.7	2.1	2.1	184.0	165.7
5	14.1	5.6	4.8	4.9	11.2	9.8	0.7	0.8	192.1	182.2
6	27.3	14.9	9.0	9.2	15.6	14.1	8.0	8.2	241.3	181.0
7	108.7	58.3	21.3	21.8	42.5	42.7	13.9	14.2	333.2	325.3
8	116.9	34.9	17.4	16.7	30.1	26.1	13.3	13.9	281.1	254.0
9	78.9	25.4	13.9	13.2	20.5	18.0	37.7	36.3	291.5	272.0
10	64.6	17.9	9.9	10.3	20.9	19.5	37.4	37.3	155.5	196.5
Mean	54.0	20.5	10.4	10.5	20.1	18.3	14.1	14.1	239.8	225.3

search tree when using the new propagators. For GP06-01, the search tree is ten times bigger when using PS or PSB while it is on average 30% smaller for the nine other instances of size 6x6.

When the running times are considered, Table 7.2 shows that it is always greater when using PS or PSB than without them, except for the solved instance of size 8 where the time is 2 to 3 times smaller, while the search tree size was almost 9 times smaller.

Note that the reported times are much longer than those presented in [BLN01] because we did not use an environment dedicated to scheduling but a general purpose constraint engine. However, implementing our new propagator in a dedicated environment would be beneficial.

The next experiment (Table 7.3) compares the mean runtime to reach the fixpoint when NFNL, EF and PS(B) are activated with the mean runtime when PS(B) is not used. This comparison is performed on the search tree obtained when NFNL, EF and PS(B) are activated with a maximum number of backtracks of 300,000. For each instance, the runtimes to reach the fixpoints are summed along every states in the search tree.

At the same time, the pruning is also compared. As for the runtime, this pruning is computed along the search tree obtained when every propagators are activated. The number of failed states with and without PS(B) activated are counted. Additionally in each state the supplementary reduction performed after adding PS(B) is counted for each type of variables ($S(t)$, $B(t)$ and $P(t)$) and these quantities are summed upon the whole search tree. The reduction is computed as the difference between the size of the domains of the variables in the initial state in a node of the search tree and their size after performing propagation until the fixpoint in the same node. If a failure is detected, the node is not taken into account for the reduction counts.

Table 7.3 presents the results averaged by size. The three first pairs of columns presents the additional pruning of the variables $S(t)$, $B(t)$ and $P(t)$. The next two columns shows the additional failures detected and the last columns reports the ad-

ditional time spent to reach those improvements. Two cells are empty because the running time was to short to compute them accurately.

It can be seen that the results are quite similar between PS and PSB, except in the columns of the starting time variables, since PS may prune inside the domain of $S(t)$ while PSB cannot. However, this difference does not influence the other variables nor the failures. Indeed, no other constraint considers forbidden values inside domains. Concerning the increase of the running time to reach a fixpoint, it is smaller with PSB because less values are removed by PSB. Taking into account the pruning potential and the used time, we can conclude that PSB is more efficient than PS. Furthermore, there are about 14% more failures detected with either version of our propagator. When no failure is detected, the domains of the variables are also substantially reduced.

Looking at the evolution of the results in function of the size of the problem, the amount of reduction of the domains increases until problems of size 7 and then decreases. The time spent follows the same scheme while the number of failures keeps increasing. Because from size 7 the search trees may be not full (because the search is cut) and the explored part is smaller for increasing size, we can suppose that our propagator detect more failures early in the search but reduces more domains at the end or in the middle of the search than in the first steps. Observing the failures for the smallest sizes, it can also be seen that PS and PSB do not reduce further the small search trees of these instances. When size grows (≥ 6) and complexity increases, PS and PSB prove their usefulness.

In conclusion, the experiments show that although the introduction of PS or PSB does not increase the number of solved instances, the addition of such a propagator substantially improves the pruning at the nodes of the search tree, as well as the number of detection of inconsistencies.

8

JUST-IN-TIME SCHEDULING

8.1 Introduction

As presented in Section 2.1, scheduling problems may feature a variety of objective functions. Minimizing makespan and the sum of weighted tardiness are probably the most commonly used and they aim at scheduling activities early. Just-In-Time Scheduling is a class of problems, which has gained importance and whose goal is not to schedule activities as soon as possible but rather at the right moment. The simplest objective capturing this high-level goal consists in having linear earliness and tardiness costs with respect to a fixed due date for each activity or each job.

This chapter studies the Just-In-Time Job-Shop Problem (JITJSP) proposed in [BFS08]. Its definition is the following. Let N be the number of jobs and M be the number of machines. Each job is composed of a sequence of M activities. Each activity A is described by the following information:

- dur(A) : The execution time of activity A.
- m(A) : The machine required by activity A.
- d(A) : The due date of activity A.
- e(A) : The earliness unit cost of activity A.
- t(A) : The tardiness unit cost of activity A.

The constraints impose that the activities of a job must be executed in the given order and that two activities requiring the same machine cannot execute at the same time. The objective is to minimize the sum of the earliness and tardiness costs of all activities. More formally, if $C(A)$ is the completion time of activity A, the earliness cost of activity A is defined as

$$E(A) = max(0, e(A) * (d(A) - C(A)))$$

and the tardiness cost as

$$T(A) = max(0, t(A) * (C(A) - d(A))).$$

The objective is to minimize the sum of $E(A) + T(A)$ for all activities. We assume that $e(A)$ and $t(A)$ are positive for all activities.

Compared to other Just-In-Time Job-Shop problems, this problem requires the costs to be defined on all activities instead of on the last activity of each job only. As discussed in [BFS08], it is more realistic to have earliness costs (e.g., storage cost) not only for the finished products but also for all partial products.

A version of the problem with costs on the last activities only was studied in several papers [BR03, DP03], as well as Just-In-Time Resource Constrained Project Scheduling Problems (JITRCPSP) [VDH01]. However, to the best of our knowledge, the JITJSP has only been studied in [BFS08], in which the authors propose several lower bounds and use a local search procedure to produce upper bounds in order to assess the value of their Lower Bounds. The JITJSP is also tackled in [LG07], where the authors present a Self-Adapting Large Neighborhood Search applied on a large set of problems comprising the JITJSP. Their method improves several upper bounds over [BFS08].

This chapter proposes a constraint-programming (CP) approach to the JITJSP. Its main focus is on the design of a global constraint for the earliness and tardiness costs but it also presents heuristics to guide the search, as well as a Large Neighborhood Search (LNS) to scale to larger instances. We start by presenting the general search algorithm. we then present the filtering algorithm and some of its theoretical results before introducing some hybrid search strategies. Finally, the chapter presents experimental results and concludes.

8.2 Branch-and-Bound for JITJSP

This section describes the CP model and the Branch-and-Bound (BB) strategy to solve the JITJSP. The basic CP model is given in Figure 8.1. It is using the Scheduling/CP module in COMET.

The model contains the input data presented in the first section. In addition, the matrix job contains, for each job, the ordered indexes of its activities. For instance, $job[4][3]$ is the index of the third activity of the fourth job. The declaration of the data is not shown in Figure 8.1. The decision variables are the completion dates of the activities and $C(A)$ denotes the completion date of activity A in the remainder of the paper. In the model, array C declared in line 8 contains the completion dates of the activities. The arrays E and T contain the auxiliary variables for the earliness and tardiness of each activity and the variable $cost$ is the total cost to be minimized (lines 13-14). Those auxiliary variables are linked to the decision variables by the constraints in lines 21, 22, and 24. The other constraints of the problem are the precedences into the jobs (lines 16–18) and the machine requirements (line 20).

Each unary resource (which represents a machine) supports the traditional disjunctive scheduling, edge-finding [CP89, CP94], and not-first-not-last [CP90, Vil04] algorithms. An almost identical model may be used to solve efficiently the classical Job-Shop Problem with makespan minimization. However, the above model is inefficient for the JITJSP, as the sum constraint in line 24 does not propagate information

```
1  range Activities = 1..N*M;
2  range Machines = 1..M;
3  range Jobs = 1..N;
4
5  Scheduler<CP> cp(0,infinity);
6  UnaryResource<CP> r[Machines](cp);
7  Activity<CP> a[i in Activities](cp,dur[i]);
8  var<CP>{int}[] C = all(i in acts)a[i].end();
9  var<CP>{int} cost(cp,0..infinity);
10 var<CP>{int} E[Activities](cp,0..infinity);
11 var<CP>{int} T[Activities](cp,0..infinity);
12
13 minimize<cp>
14    cost
15 subject to {
16    forall(j in Jobs)
17       forall(i in job[j].low()..job[j].up()-1)
18          a[job[j][i]].precedes(a[job[j][i+1]]);
19    forall(i in Activities){
20       a[i].requires(r[m[i]]);
21       cp.post(E[i]==max(0,e[i]*(d[i]-C[i])));
22       cp.post(T[i]==max(0,t[i]*(C[i]-d[i])));
23    }
24    cp.post(cost==sum(i in Activities)(E[i]+T[i]));
25 }
```

Figure 8.1: The COMET Model for the JITJSP.

```
1  Input: S : JITJSP instance
2  Input−Output: UB : Global Variable (Upper Bound)
3  Output: Best found solution
4  solve(S, UB){
5    propagate();
6    solve a machine relaxation of S;
7    Let LB be the value of the solution;
8    if (LB>=UB) fail;
9    if (the solution is machine−feasible){
10     UB := LB;
11     Save current solution;
12   }else{
13     find 2 conflicting activities A and B;
14     try{
15       S.add(A precedes B);
16       solve(S, UB);
17     }or{
18       S.add(B precedes A);
19       solve(S, UB);
20     }
21   }
22 }
```

Figure 8.2: Pseudo-code of the Branch-and-Bound

from the cost variable back to the decision variables (i.e., the completion time of the activities). In order to overcome this shortcoming, this paper introduces a global constraint to perform deductions based on the cost variable and the current state of the schedule.

The search procedure is given by the pseudo-code in Figure 8.2. This procedure is recursive and each call corresponds to a node of the search tree. After constraint propagation, a relaxation of the problem is solved. This relaxation consists in removing the machines of the problem, resulting in a PERT problem with convex cost functions that can be solved in polynomial time [CS03]. The value of this relaxation gives a valid Lower Bound (LB) for the original problem. If this LB is larger or equal to the current Upper Bound (UB), it is useless to explore the subtree rooted at the current node. In contrast, if the LB is smaller than the UB and the solution to the relaxation satisfies all the machine constraints (meaning that no two tasks that require the same machine overlap in time), the search has found a new best solution and it updates the UB. Finally, if the solution of the relaxation is not feasible in the original problem, this means that at least two activities requiring the same machine overlap in time. This constraint violation can be repaired by adding a precedence constraint between the two conflicting activities. As there are two possibilities to order the two activities,

it is necessary to branch and explore the two situations recursively (which is represented in Figure 8.2 by the non-deterministic instruction "try{}or{}" [VM05]). Note that adding a precedence can only increase the total cost.

8.3 A Global Constraint for Earliness/Tardiness

This section introduces a global constraint to reduce the search space using both the UB and the machine relaxation of the problem. The global constraint updates the domains of the variables, detects implied precedence relations, and provides heuristic information for branching.

The machine relaxation (removing the machines and the associated requirements) is a PERT problem with convex cost functions. This problem can be solved with a complexity of $\mathcal{O}(n \max\{n, m\})$ with the algorithm of [CS03] for linear earliness and tardiness costs. In this paper, the algorithmic complexities are expressed in function of the number of activities (n) and the number of precedences between activities (m). Sourd and Chrétienne's algorithm can also be generalized naturally to the case of convex piecewise linear cost functions. This generalization is important in this paper to accommodate the release dates and deadlines of the activities. As noted in [HS07], it suffices to add almost vertical segments to the cost function of an activity at its release date and deadline to model these constraints.

As mentioned, from the optimal solution of the relaxation, it is possible to perform several deductions such as bounds reduction and precedence detection. The bound reductions take the form of unary constraints on the decision variables $C(A)$, while the precedences are binary constraints on pairs of decision variables. We now review these two forms of propagation and the heuristic information provided by the global constraint.

8.3.1 Bound Reduction

The solution of the machine relaxation (i.e., the PERT problem) gives a fixed completion time for each activity A which is denoted by $C^*(A)$. Both kinds of deduction are based on what happens to the cost if the optimal solution of the relaxation is perturbed by the displacement of an activity A earlier or later than $C^*(A)$. The modification of the cost as a function of the completion time of an activity is a convex piecewise linear function whose minimum coincides with the optimal solution of the machine relaxation. Let Δ_A denote the function for activity A giving the increase in cost with respect to the optimal solution of the PERT problem. Δ_A is a convex piecewise linear function of the completion time of an activity and its minimum is $\Delta_A(C^*(A)) = 0$. This is proved in the next section by Theorem 1.

The first type of pruning is the bound reduction of the completion time variable of each activity. This pruning is enforced by the constraint

$$\Delta_A(C(A)) < (UB - LB)$$

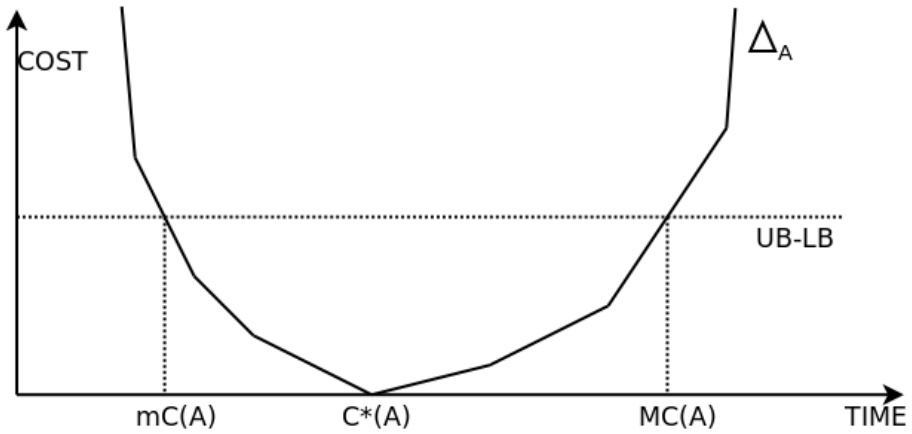


Figure 8.3: Illustration of the Delta Function and the Bound Reduction.

and, since Δ_A is a convex function, this constraint may directly update the bounds of the variable $C(A)$. Figure 8.3 shows an example of a Δ_A function with the new inferred bounds denoted by $mC(A)$ and $MC(A)$ in the figure.

8.3.2 Precedence Detection

The second type of pruning consists in detecting precedences that must hold between two tasks in conflict in the original problem (i.e., two tasks requiring the same machine and overlapping in time in the optimal solution of the PERT relaxation). If two activities A and B are in conflict, either A must precede B or B must precede A. If A is forced to precede B, A and B cannot stay at the minimum of their respective Δ_A (or Δ_B) functions. They must move and their optimal positions minimize the function $\Delta_A(x) + \Delta_B(x + dur(B))$ over x, where x represents the value given to variable $C(A)$. Let us call $Inc(A, B)$ the minimum of this function, i.e.,

$$Inc(A, B) = \min_x(\Delta_A(x) + \Delta_B(x + dur(B))).$$

$Inc(A, B)$ represents the minimum increase of the total cost when A is forced to precede B and $C^*(B) - C^*(A) < dur(B)$. This last condition is true whenever A and B are in conflict. As $C^*(B) - C^*(A) < dur(B)$ holds and the Δ functions are convex, it follows that the increase is minimized when there is no free time between A and B, i.e. if A ends at x and B ends at $x + dur(B)$. The filtering can be written as:

- if $Inc(A, B) > (UB - LB)$, then post (B precedes A).
- if $Inc(B, A) > (UB - LB)$, then post (A precedes B).

If $Inc(A, B)$ is larger than the allowed increase, then the opposite precedence can be posted. As the sum of two convex functions is also a convex function, it is easy

to compute $Inc(A, B)$ and $Inc(B, A)$ and check whether some precedence constraint must be posted.

The computation of the value $Inc(A, B)$ may consider twice a third task C but the result remains a valid lower bound of the real increase. Indeed, A and B must move in opposite directions to solve their conflict. Then C may only influence positively one of the two functions. It may have a negative influence on the other function, leading to a valid lower bound of the real increase.

8.3.3 Branching Heuristics

In addition to the above pruning, the information computed for the filtering can be used to guide the search heuristically. As indicated in Figure 8.2, the branching consists in adding precedences between two conflicting activities. The first-fail principle commands to choose two activities to detect failures earlier in the search tree. In the present problem, this suggests choosing a pair of activities that improves the lower bound the most.

More precisely, the search strategy adopted in the algorithm resolves all the conflicts of one machine before going onto another machine. It chooses the machine with the largest sum of minimum increase ($Inc(A, B)$) for all its conflicts. Among the activities requiring the chosen machine, the heuristic chooses to branch on the two conflicting activities maximizing the minimal increase in cost when they are ordered (as computed for the filtering), i.e.,

$$\underset{A\,and\,B}{\text{argmax}}\{\max(Inc(A, B), Inc(B, A))\}$$

To guide the search towards good solutions (and improve the upper bound), the branch with the smallest increase in cost is visited first

$$\text{argmin}\{Inc(A, B), Inc(B, A)\}.$$

8.4 Slope Computations for the Cost Functions

This section presents the algorithmic and theoretical results behind the computation of the Δ_A functions. It first proves that the cost functions are convex and piecewise linear. It then proposes an approximation of the Δ_A function and shows that it is sound with respect to pruning. Finally, the section describes how to compute these functions.

8.4.1 The Shape of the Slope

Theorem 1 *Starting from the optimal solution of a machine relaxation, the function Δ_A, the evolution of the cost as a function of the completion time of activity A, is convex piecewise linear.*

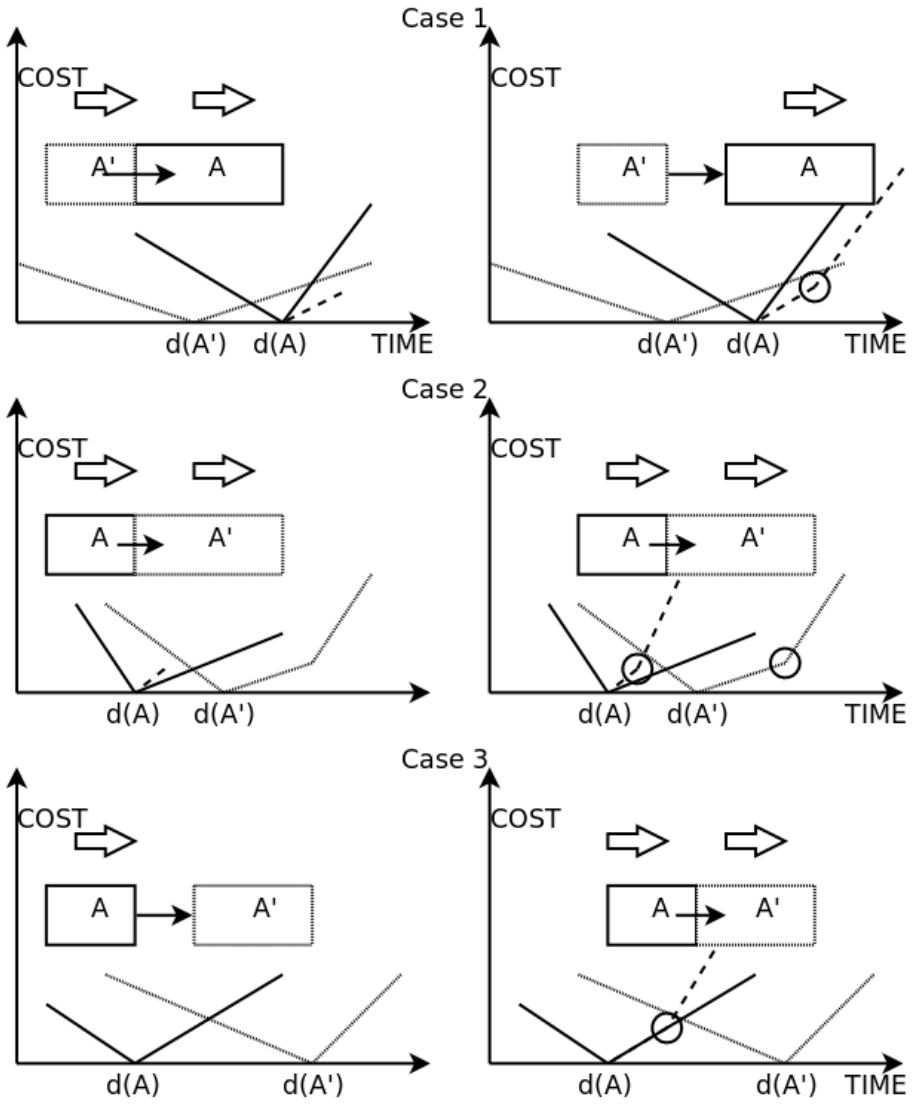


Figure 8.4: Illustration of the three cases in the proof of Theorem 1, before (left) and after (right) a breakpoint. The simple arrows depict precedences and the large arrows show the activities that are moving. Below the activities are the corresponding individual cost functions and the right of the Δ_A cost function (dashed).

Proof (Sketch) From the optimal solution, moving a task to the right or to the left can only increase the cost, as the current position is at the minimum. Let $LS(A)$ and $RS(A)$ be the unit increase cost (or slope) directly to the left and to the right of $C^*(A)$ respectively. In general, slopes of the Δ_A functions are denoted by an uppercase S, while the slopes of the individual cost function of the activities (i.e., their joint earliest/tardiness cost functions with additional segments for capturing the release and deadline constraints) are denoted by a lowercase s. For brevity, we only consider the right part of the Δ_A function as the left part is similar. When activity A is pushed to the right, some other activities must move to satisfy the precedence constraints or should move to reduce the increase in cost. At any point in time, the slope S of the function Δ_A is the sum of all the individual slopes s of the currently moving activities. Some slope s may be negative but their overall sum is positive on the right of $C^*(A)$. Directly on the right of $C^*(A)$, the cost is increased by $RS(A)$. However, the cost will increase further subsequently when breakpoints of other slopes are reached. The breakpoints are of several kinds.

1. An activity A' (or a group of activities) reaches an optimum of its individual cost function and is not forced to move by precedence constraints. This activity is left at its optimum and the slope S of Δ_A is increased by the opposite of the slope s at the left of the optimum of the individual function of A'. This is a positive increment as the slope s on the left of the optimum is negative by definition. This is illustrated in the first part of Figure 8.4 which considers the move of activity A. Before reaching $d(A')$ (left of the figure), A' moves with A. Passed $d(A')$, A continues alone and A' is left at its optimum (right of the figure).

2. A moving activity A' reaches a breakpoint of its own individual cost function (and it does not respect the conditions of case 1): The slope S of Δ_A is increased by the difference between the slopes s on the left and on the right of the breakpoint. This increment is positive as the individual cost functions are convex. This is shown in the middle part of Figure 8.4.[1]

3. The block of moving activities reaches a non-moving activity A'. The slope S of Δ_A is increased by $RS(A')$ which is positive, as this activity was optimally scheduled. This case is illustrated in the third part of Figure 8.4.

Each successive breakpoint falls in one of those three categories. The increase of the slope S of Δ_A at every breakpoint is thus positive making the whole function convex. The slope S is only modified at breakpoints and, between two breakpoints, the function follows linear segments, making the whole function piecewise linear. ■

8.4.2 Approximating the Δ_A Function

The exact outline of the Δ_A function for each activity can be computed with a variation of the PERT algorithm of [CS03]. We could not find an algorithm with a complex-

[1] Note that the individual functions may have more than 2 segments due to the release and deadline constraints.

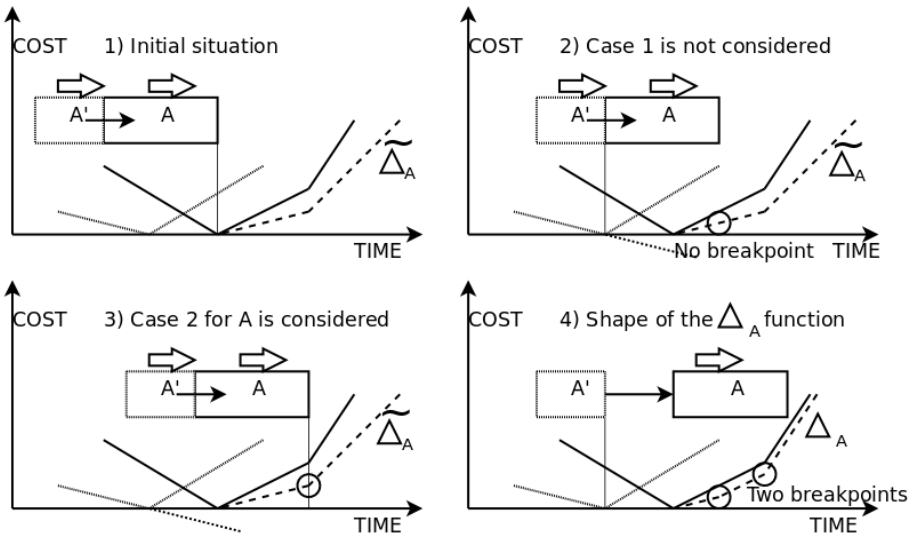


Figure 8.5: Illustration of the relaxation of the cost function. Parts 1-3 show the relaxed function $\widetilde{\Delta_A}$ while Part 4 (lower right) presents Δ_A.

ity better than $\mathcal{O}(n^2 \max\{n, m\})$. For this reason, our implementation uses a lower approximation $\widetilde{\Delta_A}$ of the Δ_A function that only considers the initial slopes ($RS(A)$ and $LS(A)$) and a subset of the breakpoints. Considering only a subset of the breakpoints gives a convex piecewise linear function that is a lower bound of Δ_A. This lower approximation is used instead of Δ_A in the implementation.

The breakpoints used in the lower approximation belong to the second category given in the proof of Theorem 1. For such breakpoints, it is relatively easy to compute the individual breakpoints of each activity and the associated increases in slope. The activities considered for $\widetilde{\Delta_A}$ are the successors of A in the transitive closure of the precedence graph. The breakpoints for all $\widetilde{\Delta_A}$ functions can then be computed in one pass over the precedence graph.

Figure 8.5 illustrates the functions Δ_A and $\widetilde{\Delta_A}$ on a small example with two activities. Part 1 of the figure shows the initial state at $C^*(A)$. Part 2 shows that breakpoints of the first category are not considered for $\widetilde{\Delta_A}$, as if the individual cost function of A' was a linear decreasing function. Part 3 shows a considered breakpoint of the individual cost function of A. Finally, part 4 presents the shape of Δ_A when all breakpoints are considered.

8.4.3 Computing $RS(A)$

It remains to show how to compute $RS(A)$ and $LS(A)$ for every activity A. For brevity, we consider the $RS(A)$ case only. Its computation can be performed in several

ways. The first possibility is to reuse the algorithm for PERT scheduling with the additional constraint that A cannot finish earlier than $C^*(A) + 1$. The difference between the optimal total cost of the variation and the optimal cost of the base version gives $RS(A)$. This mechanism has the disadvantage to run the whole PERT algorithm for every activity giving a time complexity of $\mathcal{O}(n^2 \max\{n, m\})$. A better way is to use an adaptation of the PERT algorithm. Starting from the optimal solution for the machine relaxation, it consists in adding a new fictional activity A' that has only A as successor. The due date of this task is $C^*(A) - dur(A) + 1$ and its earliness cost is set to an arbitrarily large value $e(A') = M$. The idea is to perform the first steps of the PERT algorithm until the step where it is necessary to move the block of A. At this point, the slope of the block containing A and A' is equal to $RS(A) + M$. The time complexity is bounded by $\mathcal{O}(n^2 \max\{n, m\})$. However, in practice, there are very few steps of the algorithm to perform and it is easy to remove A' from the schedule in order to be ready to compute the next RS.

8.4.4 Faster Computation of $RS(A)$

A faster and more elegant way to compute the $RS(A)$ exists for a special case that appears often in practice. Call equality arcs the arcs (A, B) of the precedence graph satisfying $C^*(B) - C^*(A) = dur(B)$, that is the arcs between two activities that are directly chaining up. The equality graph is the restriction of the precedence graph to the equality arcs. Clearly, only the activities that are part of the connected component of A in the equality graph may impact the value of $RS(A)$. In the special case where the equality graph consists of trees (there is no cycle in the underlying undirected graph), the following recurrence relations allow to compute the $RS(A)$ for all activities efficiently:

- $RS(A) = s(A) + \sum_{(A,B)} FS(A, B) + \sum_{(B,A)} BS(A, B)$
- $FS(A, B) = s(B) + \sum_{(B,C)} FS(B, C) + \sum_{(C,B)\neq(A,B)} BS(B, C)$
- $BS(A, B) = \min(0, s(B) + \sum_{(B,C)\neq(B,A)} FS(B, C) + \sum_{(C,B)} BS(B, C))$

where $s(A)$ is the slope of the individual cost function of activity A directly to the right of $C^*(A)$. The summations are performed over every in- or out-arcs of an activity in the equality graph. As the equality graph is a collection of trees, each arc separates a connected component into two disconnect parts. $FS(A, B)$ (forward slope) is the slope induced by the part containing B when A is moved. The same is true for $BS(A, B)$ (backward slope) except that it may be zero as the part containing B is not forced to move when A moves if its slope is positive. The base of the recurrence relations happen when the activity B of $FS(A, B)$ (resp. $BS(A, B)$) is incident only to the edge (A, B) (resp. (B, A)). In such a case, $FS(A, B) = s(B)$ (resp. $BS(A, B) = \min(0, s(B))$). Another special case is when an activity A is not incident to any edge. In this case, $RS(A) = s(A)$. This means that the recurrence starts from the leaves of the tree. There are two values to compute for each arc and one for each node, that is there are $\mathcal{O}(n + m)$ values to compute in total.
Figure 8.6 presents a little example with 4 activities. The $s(A)$ and $RS(A)$ are noted in the respective activities, while the $FS(A, B)$ and $BS(A, B)$ are noted on the arcs.

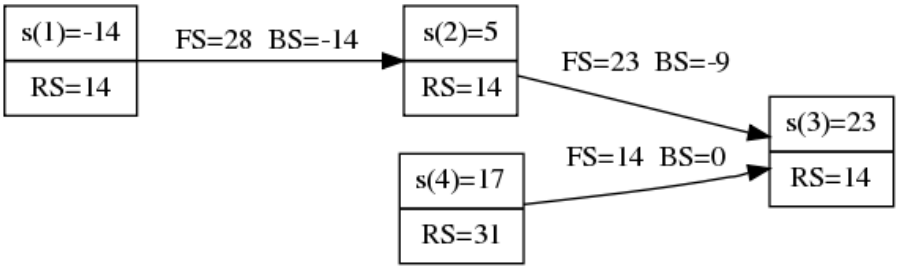


Figure 8.6: Illustration of the computation of the RS(A)

The arc $(4, 3)$ illustrates the case where activity 4 does not move when any of the three others move, as it would incur an additional cost of 17 ($BS(3, 4) = 0$). On the contrary, activities 1 and 2 are moved whenever activity 3 is displaced because it reduces the cost ($BS(2, 1)$ and $BS(3, 2)$ are negative).

The same relations exist for $LS(A)$. In the experiments, the faster computation mechanism is used whenever it is possible. The variation of the PERT algorithm is only used in the cases where the equality graph is not composed of trees.

8.5 Additional Heuristics

As JITJSPs are extremely hard problems, we embedded two additional mechanisms into our search procedure: a simple local search to post-optimize each solution and a Large Neighborhood Search [Sha98, DP03].

8.5.1 Simple Local Search

The local search starts from a feasible solution and tries to improve it greedily by swapping the order of two tasks that execute successively on the same machine and undo the move if it does not improve the value of the solution. This is repeated until a local optimum is obtained. Each time the branch-and-bound finds a new (improving) solution, the local search is run from this solution trying to improve it. The value of the local optimum is then used as a new UB. This makes it possible to obtain good upper bounds early. It is important to note that the addition of this local search preserves the completeness of the Branch-and-Bound search.

8.5.2 Large Neighborhood Search

In addition to the above complete search procedure, we also implemented an incomplete Large Neighborhood Search (LNS). LNS is a local search whose moves explore the solutions of a subproblem using CP. For the JITJSP, our LNS consists of the following steps:

1. Let $n = 20$;
2. Choose $n/10$ machines.
3. Relax the current solution by removing the precedences between activities executed on the chosen machines.
4. Solve the problem with the B&B search limited to 1000 fails.
5. Update the current solution if a better solution has been found.
6. Increase n if the B&B search was completed, decrease it otherwise (with a minimum of 20).
7. Go to step 2 unless the running time is exhausted.

The choice of the machines to relax is performed randomly according to a distribution that reflects the costs incurred by the activities executed on each machine.

8.6 Experimental Validation

The aim of the experiments is to demonstrate the effectiveness of the global constraint and the influence of the two additional mechanisms to help solving the JITJSP. The benchmarks are those introduced in [BFS08] and include 72 instances ranging from 20 to 200 activities. The instances are distributed following four criteria: tightness of the due dates, repartition of the costs, number of jobs, and number of machines. The due dates are either tight or loose. If they are tight, the distance between the due dates of two successive activities of a job are equal to the duration of the second activity. If they are loose, some free time is allocated between two due dates. The earliness and tardiness unit costs are either taken randomly in $[0.1, 1]$ (equal scheme) or in $[0.1, 0.3]$ for the earliness cost and in $[0.1, 1]$ for the tardiness cost (tard scheme). The number of jobs is 10, 15, or 20, and the number of machines is 2, 5, or 10. Only 1 instance of this benchmark was previously closed (optimum known and proved), namely the 10x2-t-l-1 instance (10 jobs, 2 machines, tight due dates, loose cost scheme, number 1).

We ran four versions of our search on the whole collection of benchmarks. The first one, denoted *CPnoG*, is the pure CP approach without the novel global constraint.The second one, called *CP*, is the same CP approach but with the novel global constraint. *CP+ls* adds the local search described in the previous section and *LNS* adds the local search and the Large Neighborhood Search of the previous section. Every run is allocated 600 seconds (10 minutes) and is performed on one core of a Intel Core 2 Quad at 2.40GHz with 4MB of memory. The entire algorithm is implemented in Comet.

8.6.1 Results

The results are presented in Tables 8.1, 8.2 and 8.3 (Pages 116-117).The column *LB* gives the best-known lower bound. These values are taken from [BFS08], except those we improved, which are shown in italic. The column *BF&S* shows the upper bounds from [BFS08] and serves as reference for our results. The column *SA-LNS* presents

Instance	LB	BF&S	SA-LNS	CPnoG	CP	CP+ls	LNS
10x2-tight-equal-1	434	**453**	-	742.15	461.96	461.96	522.9
10x2-tight-equal-2	*448.32*	458	-	**448.32**	**448.32**	**448.32**	484.86
10x5-tight-equal-1	660	826	-	2556.47	935.78	783.43	**764.8**
10x5-tight-equal-2	612	848	-	1686.04	**779.4**	**779.4**	808.64
10x10-tight-equal-1	1126	1439	-	5901.26	1622.26	**1339.64**	1527.28
10x10-tight-equal-2	1535	2006	-	5552.85	1930.65	1930.65	**1902.3**
10x2-loose-equal-1	*224.84*	225	-	384.52	**224.84**	**224.84**	225.81
10x2-loose-equal-2	313	324	-	565.15	**319.37**	**319.37**	347.65
10x5-loose-equal-1	1263	1905	-	3191.61	1995.5	1877.93	**1823.85**
10x5-loose-equal-2	878	1010	-	2732.44	1851.56	1155.89	**999.14**
10x10-loose-equal-1	331	**376**	-	1676.34	620.29	403.87	381.88
10x10-loose-equal-2	246	260	-	4122.75	325.92	274.31	**256.78**
10x2-tight-tard-1	*179.46*	195	-	184.9	**179.46**	**179.46**	193.44
10x2-tight-tard-2	143	**147**	-	259.17	173.67	164.38	164.38
10x5-tight-tard-1	361	405	-	1722.42	444.64	407.4	**398.37**
10x5-tight-tard-2	461	708	-	1364.13	722.76	707.81	**639.16**
10x10-tight-tard-1	574	855	-	2138.78	928.98	806.74	**773.26**
10x10-tight-tard-2	666	**800**	-	1939.05	1094.71	879.5	830.39
10x2-loose-tard-1	416	**416.44**	-	**416.44**	**416.44**	**416.44**	**416.44**
10x2-loose-tard-2	137	138	-	171.11	148.31	**137.94**	147
10x5-loose-tard-1	168	188	-	930.13	243.06	199.91	**182.64**
10x5-loose-tard-2	355	572	-	1643.19	733.69	**513.91**	542.29
10x10-loose-tard-1	356	409	-	1950.79	476.82	402.27	**387.05**
10x10-loose-tard-2	138	152	-	1155.84	152.66	151.97	**144.94**

Table 8.1: Detailed results for the 10-jobs instances

Instance	LB	BF&S	SA-LNS	CPnoG	CP	CP+ls	LNS
15x2-tight-equal-1	3316	3559	**3372.09**	4318.16	4269.09	3641.19	3641.19
15x2-tight-equal-2	1449	1579	**1508.59**	1885.51	1578.2	1534.12	1534.12
15x5-tight-equal-1	1052	1663	1684.17	5131.06	1604.52	1538.09	**1504.04**
15x5-tight-equal-2	1992	2989	**2919**	7719.21	3042.32	2993.5	3096.6
15x10-tight-equal-1	4389	8381	**6848.97**	16798.5	9870.99	9089.61	8189.7
15x10-tight-equal-2	3539	7039	7199.82	20606.2	10072.8	5665.38	**5536.07**
15x2-loose-equal-1	1032	1142	**1048.47**	1389.12	1453.6	1249.68	1249.68
15x2-loose-equal-2	490	**520**	529.24	1064.7	550.86	524.1	560.15
15x5-loose-equal-1	2763	4408	**3572.86**	10489.4	5011.53	3757.93	3745.96
15x5-loose-equal-2	2818	4023	3642.24	9786.51	5449.25	3418.87	**3397.42**
15x10-loose-equal-1	758	1109	1205.42	10851.1	1747.86	1083.02	**1033.06**
15x10-loose-equal-2	1242	2256	1855.01	10533.2	3703.48	1937.27	**1792.67**
15x2-tight-tard-1	786	913	**824.19**	1123.88	1214.39	835.52	835.52
15x2-tight-tard-2	886	956	**905.37**	1219.88	1100.32	947.17	947.17
15x5-tight-tard-1	1014	1538	1553.22	4365.97	1567.86	**1530.96**	1597.9
15x5-tight-tard-2	626	843	**761.25**	4030.46	959.25	785.36	775.01
15x10-tight-tard-1	649	972	**921.46**	3411.5	1458.38	921.67	923.88
15x10-tight-tard-2	955	1656	**1633.14**	8666.48	2341.59	1663.05	1693.04
15x2-loose-tard-1	650	730	**655.93**	847.64	869.72	666.37	666.37
15x2-loose-tard-2	278	**310**	312.17	567.02	370.98	336.48	336.48
15x5-loose-tard-1	1098	1723	**1431.36**	5849.32	3802.18	1528.36	1478.97
15x5-loose-tard-2	314	**374**	386.25	1642.12	585.72	409.6	401.65
15x10-loose-tard-1	258	312	324.03	5200.42	564.65	342.49	**300.11**
15x10-loose-tard-2	476	855	781.8	11036.2	1378.26	**658.9**	717.9

Table 8.2: Detailed results for the 15-jobs instances

Instance	LB	BF&S	SA-LNS	CPnoG	CP	CP+ls	LNS
20x2-tight-equal-1	1901	2008	**1967.4**	2434.94	2148.81	2115.58	2115.58
20x2-tight-equal-2	912	1014	**993.43**	1412.43	1400.2	1104.2	1124.47
20x5-tight-equal-1	2506	**3090**	3266.35	9800.11	4085.52	3349.28	3349.28
20x5-tight-equal-2	5817	7537	**7456.91**	15568.0	10226.2	8112	7883.5
20x10-tight-equal-1	6708	12951	**11929.2**	42488	21405.9	14537.1	14004.9
20x10-tight-equal-2	5705	9435	**7763.34**	34149.5	13363.6	8603.76	8535.88
20x2-loose-equal-1	2546	**2708**	2750.21	3142.91	3410.65	2789.07	2789.07
20x2-loose-equal-2	3013	3318	**3182.94**	4754.77	3760.64	3386.88	3386.88
20x5-loose-equal-1	6697	9697	**8285.88**	27431.9	15069.2	9481.56	9481.56
20x5-loose-equal-2	6017	**8152**	8400.03	25041.6	13138.5	8835.72	8835.72
20x10-loose-equal-1	3538	6732	6742.01	36924.7	10773.3	6206.3	**6101.67**
20x10-loose-equal-2	1344	2516	2023.89	18332.5	4797.62	2006.67	**1963.05**
20x2-tight-tard-1	1515	1913	**1761.69**	2729.81	2395.97	1892.22	1892.22
20x2-tight-tard-2	1375	1594	**1471.25**	2039.81	2121.18	1704.26	1744.25
20x5-tight-tard-1	3244	4147	**3778.88**	9814.55	6420.98	4067.73	4067.73
20x5-tight-tard-2	1633	**1916**	2006.45	9443.06	3497.69	2040.7	2040.7
20x10-tight-tard-1	3003	5968	5622.63	32574.2	9871.52	5172.14	**5125.88**
20x10-tight-tard-2	2740	**3788**	4382.08	19923.2	6229.24	3992.48	3938.51
20x2-loose-tard-1	1194	1271	**1256.02**	2177.05	1545.92	1409.73	1409.73
20x2-loose-tard-2	735	857	**784.54**	1700.04	1170.59	907.6	907.6
20x5-loose-tard-1	2524	**3377**	3421.56	8466.93	5091.8	4015.62	4644.44
20x5-loose-tard-2	3060	5014	**3965.95**	13980.5	6946.88	4539.36	4539.36
20x10-loose-tard-1	2462	**6237**	6877.88	38611.5	20511.3	7462.39	7287
20x10-loose-tard-2	1226	1830	2046.53	12267.0	2776.79	1741.44	**1727.88**

Table 8.3: Detailed results for the 20-jobs instances

the results from [LG07]. This approach has only been tested on the instances with 15 or 20 jobs. The last four columns show the cost of the best solutions found by each version of our algorithm. In the case of *LNS*, this value is an average over 10 runs. The bold values are the best ones for each instance.

The comparison of the values between *CPnoG* and *CP* shows clearly the benefits of using the new global constraint. On the whole, *CP* improves over *BF&S* on 9 instances. *CP+ls* and *LNS* provide significant benefits and improve over *BF&S* on 40 instances each. In addition, *CP+ls* was able to prove optimality on 5 instances thus closing 4 new instances. These new instances with their respective total costs are:

- 10x2-tight-equal-2 : 448.32
- 10x2-loose-equal-1 : 224.84
- 10x2-tight-tard-1 : 179.46
- 10x2-loose-tard-2 : 137.94

SA-LNS provides the best solution for 26 instances out of 48, while our *LNS* approach only provides 10 such best solutions. But is is interesting to analyze the average performance of the algorithms, as summarized in Table 8.4 (Page 118). In this table, we give the gap averaged by size. The gap is defined as $(UB - LB)/LB$ where LB is the best-known lower bound of an instance and UB is the total cost computed by each algorithm for the same instance. This table indicates that, for the smallest instances (20 tasks), *BF&S* gives better solutions. In general, for the instances with 2 or

Size	BF&S	SALNS	CPnoG	CP	CP+ls	LNS
10x2	**2.80**	-	41.48	4.79	3.03	8.79
15x2	10.11	**4.46**	53.41	29.61	10.06	10.98
20x2	12.32	**7.66**	63.30	40.81	18.09	18.73
10x5	33.55	-	277.04	58.67	32.07	**27.09**
15x5	46.60	**36.67**	366.54	90.08	37.49	37.32
20x5	34.50	**28.30**	294.70	102.88	39.92	42.54
10x10	21.48	-	539.69	45.06	21.75	**19.06**
15x10	67.69	58.84	1026.50	152.08	56.67	**51.30**
20x10	84.44	81.06	902.55	253.81	81.92	**78.53**

Table 8.4: Average gap by size (in %). Bold values are the best on their benchmarks

5 machines, *SA-LNS* is the best. However, as size increases, *LNS* becomes the best on average. Again, we see that the pure *CP* search greatly benefits from the introduction of the global constraint. However it is still not effective due to the lack of a good upper bound early in the search which would allow to prune large parts of the search tree. Adding good upper bounds, thanks to the local search, greatly improves the pure *CP* approach. Finally, LNS improves the larger instances but is useless on instances with 2 machines. Indeed, the idea behind our *LNS* is to relax at least 2 machines, which is the complete problem in 2-machines instances. For such instances, *LNS* only serves as a restart and cannot drive the search as it is the case for larger problems.

Additional experiments have shown that the PERT algorithm adapted from [CS03] is the bottleneck of the search. Future work will be devoted to the development of an incremental version of a convex PERT algorithm and to an extension of the approach to other kinds of Just-In-Time Scheduling Problems that feature cumulative and state resources. In particular, it will focus on the adaptation of the global constraint for these problems.

9

CONCLUSION

Our thesis focused on solving scheduling problems from high-level models. Our contributions are twofold, working both on the synthesis of algorithms from the models, and on improved search procedures with new propagators for global constraints. In Section 9.1, we recall the main achievements of the thesis. In Section 9.2, we also outline the main limitations of our work, and we present several possible directions for future research.

9.1 Results

The main contributions of our thesis are a set of high-level modeling abstractions to describe scheduling problems, an engine to simplify and classify problems from the models, and a set of synthesizers to solve the problems based on their classification. The goal of the classification of the problem is to generate an appropriate algorithm, with minimal intervention of the user.

As presented in Chapter 5, the modeling layer allows to represent a large range of problems using a syntax close to other existing modeling languages. The modeling layer is totally independent from the underlying search algorithms, and the user does not have to commit to one algorithm. Moreover the user does not have to write the search procedure, but has simply to choose a synthesizer. Synthesizers correspond to different search paradigms (e.g. Constraint Programming, Local Search).

The classification of a problem is done is several steps (see Chapter 3). First, the model is mapped to an internal form and simplified. The structure of the simplified form is then analyzed to retrieve a large set of characteristics of the problem. Finally, the values of the characteristics define the classes the problem belongs to.

Synthesizers associate search strategies with classes of problems. The right strategy is executed based on the class of the problem. The strategy can be further specialized given the characteristics of the problem. Views of the problem are used to feed the data to the strategy in a neutral way. This has been explained in Chapter 4.

We have also shown that, given the separation of modeling and solving, it is straightforward to combine different search algorithms, and to create loosely coupled hybrid algorithms.

Our contributions are also in algorithms to solve scheduling problems. We presented a Constraint Programming propagator for disjunctive resources that performs a shaving on the positions of activities to reduce the search space (Chapter 7). We also introduced a global constraint for the Just-In-Time objective, that reduces the domain of the activities, and detects implied precedence constraints (Chapter 8). This global constraint has been shown to be very useful to solve hard Just-In-Time Job-Shop Problems.

We developed the AEON system as a proof-of-concept of our contributions. AEON includes a modeling library, the modules to analyze and classify a problem, and a set of synthesizers. The synthesizer dedicated to constraint programming contains the propagators aforementioned. One of the strengths of AEON is that it is extensible. It is easy to add new classes of problems, new synthesizers and new strategies.

In Chapter 6, we have experimentally shown that using AEON does not induce a large overhead for the analysis and classification of problems. For largely studied problems, our implementation of the search procedures is not able to match state-of-the-art algorithms. But the extension mechanism permits to add such algorithms to render AEON competitive on those problems. Another asset of AEON is that it is also able to solve those specific problems even tough they are not presented as such. Finally, AEON can solve a broad range of problems, in particular heterogeneous problems.

9.2 Future Work

AEON, as a prototype implementation, has several limitations that are due to the academic nature of our research. We can cite the lack of different underlying synthesizers (for different search techniques), and a limited coverage of the problems that can be solved by the existing strategies. This problem is also reflected in the average performance of the search algorithms for classical problems. Part of our future work will be devoted to include state-of-the-art algorithms in AEON for different kind of problems.

Of particular interest is the generalization of successful algorithms to classes of problems broader than originally designed for. This can be done by looking at the definition of the classes of problems (as depicted in Appendix C), and by studying how an algorithm for a class of problems depends on the constraints put on this class, and how these constraints can be relaxed.

We focus in the following on the non-technical limitations of our contributions. They open the doors to future research possibilities.

- The modeling library defines an immutable set of abstractions. It is not possible to define new constraints outside the existing ones. This limitation is hard to remove, because adding new abstractions for modeling requires to add their counterpart in the internal representation, and to take them into account in the analysis and classification of problems.

This calls for having a very different system where it is possible to add new concepts (defined by a modeling abstraction, plus a representation in the internal form, a set of analysis functions and classification) in a simple way. This takes the problem one step further, as such a system would allow to create modeling libraries not only for scheduling but for many other types of problems.
We currently have very few clues as how to design such a system. Good starting points may be existing modeling languages that allow extension (like e.g. ZINC [MNR+08]), or Constraint Programming systems that let the user define his own constraints (like this is the case in COMET [VM05]). The main difficulty, however, is that in our case, we loose the important property of orthogonality. More precisely, when we add a modeling concept, it potentially affects the definition of all existing classes of problems, which is not desirable in practice.

- The previous point is (too) ambitious. There are, however, large classes of problems that deserve being studied specifically in the perspective of a scheduling system. We identified problems involving uncertainty, online features, and/or with several (competing) objectives. For such problems, it is necessary to redefine the characteristics of interest, and how they relate with the efficiency of the search algorithms.
  An interest of these classes of problems is that algorithms in the literature to solve them often make use of existing algorithms to solve the underlying deterministic or "single-objective" problems. In a synthesis system, this means that we might embed existing strategies as building blocks of more complex strategies, to solve problems with uncertainty regardless of the other characteristics of the problems.
- The mapping from classes of problems to strategies is currently constructed by hand, from our own knowledge. This is the same for the many parameters of strategies that are experimentally tuned in a trial-error fashion. In the spirit of the automatic generation of search algorithms, the automated learning of the mapping and the automated tuning of the parameters would be big improvements. A lot of research has been carried on the automated tuning of algorithms (see e.g., [BYBS09]), but we are not aware of works on the automation of a mapping between classes of problems and algorithms. This can be considered as a learning problem, where we want to learn a function from the set of characteristics of a problem to an algorithm and its parameters. Notice that it is different from a self-adapting algorithm that learns the parameters to use for an instance while solving this instance.
  With the automation of the mapping and tuning, it would become easier to increase the discrimination power of the classification of problems. As an example, we saw during our experiments that Group-Shop instances closer to a Job-Shop are not solved as well as the other ones with the current strategies. Creating a new class for those problems is not difficult with the extension mechanism for new classes of problems. Designing another more efficient search algorithm for this subclass may take some time. Part of this time may be spared if the system is able to choose the right parameters and building blocks of existing

strategies by itself.

- Another path of future research is in the building of more complex and more powerful hybrid algorithms. Indeed, synthesizers and strategies can be easily combined sequentially. However, there is room for other kinds of hybrids, such as parallelization of algorithms, or combination of master and slave algorithms. In our context, we can take advantage of the classification of the problems to automatically discover relaxed versions of the problem, and to associate hybrids to some classes of problems. The use of the DAG of classes will be instrumental for the discovery of relaxations of a problem. Relaxations are very useful to find good bounds on the value of an objective function, and to guide other search procedures.

A

AEON'S MODELING API

This appendix presents the details of the Abstract Public Interface (API) to model a scheduling problem in AEON. This is composed of a set of classes and function in COMET. For each class, we separate the methods in different groups, which are:

- the constructors,
- the methods that state a constraint or the objective of the problem,
- the methods to access related objects.

Examples of uses of the API can be found in Appendix B.

A.1 Class Schedule<Mod>

This is the central class of the model. It must be created first, and passed to the constructor of most other objects.

Constructors

- `Schedule<Mod>::Schedule<Mod>(int _horizon)`
  Constructor with a fixed horizon.
- `Schedule<Mod>::Schedule<Mod>()`
  Constructor without horizon. The horizon is fixed to `System.getMAXINT()`.

Constraints and Objectives

- `void allPrecedencesAreNoWait()`
  Forces all tasks linked by precedence constraints to be executed directly one after the other.
- `void preemptionIsAllowed()`
  Allows preemption for all the activities of the schedule.

- `void maximizeObj(SchedulingObjective<Mod> obj)`
  Tells that the goal is to maximize the given objective function.
- `void minimizeObj(SchedulingObjective<Mod> obj)`
  Tells that the goal is to minimize the given objective function.
- `void satisfy()`
  Tells that the goal is to find a satisfiable solution. Any call to one of the three methods for objectives overwrites a previous call.

Accessors

- `TimePoint<Mod> end()`
  Returns the end time point of the schedule.
- `TimePoint<Mod> start()`
  Returns the starting date of the schedule.
- `TimePoint<Mod> zero()`
  Returns the reference "origin" time point.
- `int getHorizon()`
  Returns the horizon of the schedule

Miscellaneous

- `void showGraph()`
  Shows the precedence graph (in PNG, layout by "dot").
- `string toString()`
  Returns a textual representation of the problem.

A.2 Activities, Jobs and Precedences

Activities are central in the definition of a scheduling problem. Consequently their API presents methods to state a lot of different constraints, both related to resource requirements and to precedences.

Class Task<Mod>

Task<Mod> is the super class for Activity<mod> and Job<Mod>. This class corresponds to the behavior common to activities and jobs, in particular precedence constraints. It is not intended to be directly instantiated.

Constraints

- `void precedes(Task<Mod> a)`
  Constraints this task to come before task "a".
- `void precedesDirectly(Task<Mod> a)`
  Constraints task "a" to start as soon as this task has ended (no-wait constraint).

- `void follows(Task<Mod> a)`
  Constraints this task to come after task "a".
- `void followsDirectly(Task<Mod> a)`
  Constraints this task to start as soon as task "a" has ended (no-wait constraint).
- `void isReleasedAt(int date)`
  This task cannot start before the given date.
- `void hasDeadline(int date)`
  This task cannot end after the given date.
- `void isPartOf(Job<Mod> j)`
  Tells that this task is part the given job.
- `void preemptionIsAllowed()`
  Allows preemption for the activity, or for all activities enclosed in the job.

Accessors

- `TimePoint<Mod> start()`
  Returns the start of the task.
- `TimePoint<Mod> end()`
  Returns the end of the task.
- `Job<Mod> job()`
  Returns the containing job if it exists. Returns null otherwise.

Class Activity<Mod> (extends Task<Mod>)

Objects of this class represent (single-mode) activities.

Constructors

- `Activity<Mod>::Activity<Mod>(Schedule<Mod> s, int proctime, string name)`
  Creates a new Activity in the schedule with the processing time "proctime" and the given name.
- `Activity<Mod>::Activity<Mod>(Schedule<Mod> s, int minproctime, int maxproctime, string name)`
  Creates a new Activity in the schedule with a processing time to determine between "minproctime" and "maxprotime", and with the given name.

Constraints

- `void requires(Machine<Mod> m)`
  Tells that the activity needs the given machine.
- `void requires(CumulativeResource<Mod> m, int cap)`
  Tells that the activity requires "cap" units of resource "m"
- `void provides(CumulativeResource<Mod> m, int cap)`
  Tells that the activity provides "cap" units of resource "m" during its execution.

- `void consumes(Reservoir<Mod> m, int cap)`
  Tells that the activity consumes some amount of the given reservoir.
- `void produces(Reservoir<Mod> m, int cap)`
  Tells that the activity produces some amount for the given reservoir.
- `void requires(StateResource<Mod> m, int state)`
  Tells that the activity requires the resource "m" to be in the state "state".
- `void requiresOneOf(Machine<Mod>[] m)`
  Tells that this activity needs one of the given machines.
- `void requiresOneOf(set{Machine<Mod>} m)`
  Tells that this activity needs one of the given machines.
- `void requiresOneOf(set{CumulativeResource<Mod>} m, int cap)`
  Tells that this activity needs "cap" units of one of the given resources.
- `void requiresOneOf(CumulativeResource<Mod>[] m, int[] cap)`
  Tells that this activity needs one of the given resources. The request for resource "m[i]" is "cap[i]".
- `void requires(Need<Mod> n)`
  Adds a request (need) of resources to the activity. See later for the creation of needs.y
- `void requiresAlternatively(Need<Mod> n)`
  Offers an alternative request (need) of resources for the activity. See later for the creation of needs.
- `void setType(int i)`
  Sets the type of the activity.
- `void setOptional()`
  States that this activity can be absent in the schedule.

Class MultiModeActivity<Mod> (extends Activity<Mod>)

A Multi-Mode Activity has a different behavior depending on its mode. It has all the features of a normal activity. The resource requests are defined on the current mode (that can be changed). The mode that is specified can also be given as first argument of the methods. For instance "void requires(int mode, CumulativeResource<Mod> m, int cap)".

Constructor

- `MultiModeActivity<Mod>::MultiModeActivity<Mod> (Schedule<Mod> s, int nbMode, string name)`
  Creates a new Multi-mode Activity in the given Schedule, with the given number of modes and the given name. The current mode is the first (whose index is 1).

Constraints

- `void addMode()`
  Add a new mode and switch to this mode for further constraints.
- `void setCurrentMode(int mode)`
  Changes the mode that is currently specified.
- `void setProcTime(int procTime)`
  Tells that the current mode has the given processing time.
- `void setProcTime(int minProcTime, int maxProcTime)`
  Tells that the processing time of the current mode lies in the given interval.
- `void setProcTime(int mode, int minProcTime, int maxProcTime)`
  Sets the interval of possible processing time of the mode "mode".

Accessors

- `int getCurrentMode()`
  Returns the index of the current mode.
- `int getNumberOfModes()`
  Returns the number of modes of this activity.

Class Job<Mod> (extends Task<Mod>)

A job is container for a group of tasks. It can hold both activities and other jobs at the same time. A task can be part of only one job.

Constructor

- `Job<Mod>::Job<Mod>(Schedule<Mod> s, string name)`
  Creates a new job in the given schedule and with the given name.

Constraints

- `void contains(Task<Mod> t)`
  Tells that the task "t" is part the job.
- `void contains(Task<Mod>[] ts)`
  Tells that all the tasks in "ts" are part of the job.
- `void contains(Activity<Mod>[] ts)`
  Tells that the job contains all the activities in table "ts".
- `void contains(set{Activity<Mod>} ts)`
  Tells that the set of activities "ts" is part of the job.
- `void contains(set{Task<Mod>} ts)`
  Tells that the set of tasks "ts" is part of the job

- `void containsInSequence(Task<Mod>[] ts)`
  Tells that the tasks in "ts" are part of the job and they must be executed in the given order.
- `void containsInSequence(Activity<Mod>[] ts)`
  Tells that the job contains the tasks in "ts" and that they must be executed in the specified order.
- `void allPrecedencesAreNoWait()`
  Forces all tasks inside the job that are linked by precedences to be executed without interruption between them.
- `void hasMaxSlack(int mslack)`
  Forces the job to be executed in less time than the given value.
- `void preemptionIsAllowed()`
  Allows preemption for all contained activities.
- `void noOverlap()`
  No two tasks inside the jab can be executed at the same time.

Accessors

- `set{Task<Mod>} tasks()`
  Returns the set of tasks that are part of the job.

Class TimePoint<Mod>

Time points represent the start and the end of the tasks and of the schedule. They can be seen as events. They are not directly instantiated by the user, but they can be queried with the "start()" and "end()" methods of "Task<Mod>" and "Schedule<Mod>".

Constraints

- `void comesAfter(int date)`
  This point is forced to come after the given date (relative to the zero reference).
- `void comesAfter(TimePoint<Mod> t, int delay)`
  This point must come at least "delay" units of time after the given point.
- `void comesAt(TimePoint<Mod> t, int delay)`
  This point must come exactly "delay" units of time after the given one. It will come before if the delay is negative.
- `void comesBefore(TimePoint<Mod> t, int delay)`
  This point must come at least "delay" units of time before the given point.
- `void comesBefore(int date)`
  This point is forced to come before the given date (relative to the zero reference).
- `void comesBetween(int date1, int date2)`
  This point is forced to come between the two given dates (relative to the zero reference).
- `void follows(TimePoint<Mod> t)`
  This point must come after the given one.

- `void precedes(TimePoint<Mod> t)`
  This point precedes the given one.

A.3 Resources and requirements

Resources are represented using four different classes for the four types of resources. These classes provide an alternate way to state that they are required by some activities.

Class CumulativeResource<Mod>

This is the general class for resources. It represents Cumulative resources.

Constructors

- `CumulativeResource<Mod>::CumulativeResource<Mod>`
  `(Schedule<Mod> s, int cap, string name)`
  Creates a resource in the given schedule with the specified capacity and name.
- `CumulativeResource<Mod>::CumulativeResource<Mod>`
  `(Schedule<Mod> s, int minCap, int maxCap, string name)`
  Creates a resource in the given schedule with the specified minimal and maximal capacities and the given name.

Constraints

- `void addBreak(int init, int length)`
  Add a break to this resource. The break makes the resource unavailable starting at the date "init" and for "length" units of time.
- `void addPeriodicBreak(int init, int length, int period)`
  Add a break that is repeated regularly. Breaks begin at "init", "init + period", "init + 2*period", ...
- `void addProfileComponent(int init, int end, int height)`
  Add a profile component to the resource. It changes the capacity of the resource between the dates "init" and "end" to the given capacity "height".
- `void requiredBy(Activity<Mod> a, int cap)`
  Tells that the given activity requires "cap" units of the resource.
- `void setTransitionMatrix(int[, ] matr)`
  Add a matrix of transition times between different kinds of activities.

Class Machine<Mod> (extends CumulativeResource<Mod>)

A machine is a cumulative resource whose capacity is set to one.

Constructor

- `Machine<Mod>::Machine<Mod>(Schedule<Mod> s, string name)`
  Creates a new machine in the schedule with the given name.

Constraints

- `void requiredBy(Activity<Mod> a)`
  Tells that the given activity requires the machine. This is similar to "a.requires(mach)".
- `void requiredBy(Activity<Mod>[] as)`
  Tells that the activities in "as" require the machine.
- `void requiredBy(set{Activity<Mod>} as)`
  Tells that the set of activities requires this machine.

Class Reservoir<Mod> (extends CumulativeResource<Mod>)

A reservoir is resource whose current capacity can be increased or decreased by activities.

Constructor

- `Reservoir<Mod>::Reservoir<Mod>(Schedule<Mod> s, int minCap, int maxCap, int initCap, string name)`
  Creates a reservoir in the schedule. It has the given minimal, maximal and initial capacities and the specified name.

Class StateResource<Mod> (extends CumulativeResource<Mod>)

A state resource is a resource that has different states but no capacity. An activity requires the resource to be in some predefined state.

Constructor

- `StateResource<Mod>::StateResource<Mod>(Schedule<Mod> s, range states, int initState, string name)`
  Creates a State Resource with a set of states, an initial state and a name.

Class Need<Mod>

Need<Mod> represents all the needs of activities for resources. It has different subclasses. They are not intended to be directly created. Rather use the provided functions (they are not methods). A need must be associated to an activity using one of the methods "requires(need)" or "requiresAlternatively(need)".

Functions to create needs

- `Need<Mod> Requires(Machine<Mod> m)`
  Returns the requirement for a machine.
- `Need<Mod> Requires(StateResource<Mod> r, int s)`
  Returns a request for state "s" of the given state resource.
- `Need<Mod> Requires(CumulativeResource<Mod> r, int c)`
  Returns a request of "c" units of the given resource.
- `Need<Mod> Provides(CumulativeResource<Mod> r, int c)`
  Returns a request that provides "c" units of the given resource.
- `Need<Mod> Consumes(Reservoir<Mod> r, int c)`
  Returns a request that consumes "c" units of the given resource.
- `Need<Mod> Produces(Reservoir<Mod> r, int c)`
  Returns a request that produces "c" units of the given resource.
- `ConjunctiveNeeds<Mod> And(Need<Mod> n1, Need<Mod> n2)`
  returns the conjunction of two requirements.
- `AlternativeNeeds<Mod> Or(Need<Mod> n1, Need<Mod> n2)`
  Returns the disjunction of two requirements.

A.4 Objective Functions

The objective function is built using a set of classes representing different basic functions (directly depending on the state of one task), and aggregate functions. For convenience, the objective function can be defined using a set of COMET functions and operators. The complete objective function is passed to the methods "maximizeObj(obj)" and "minimizeObj(obj)" of the "Schedule<Mod>" object.

Class SchedulingObjective<Mod>

This is the general class for objectives. It is not intended to be instantiated directly.

Class TaskObjective<Mod> (extends SchedulingObjective<Mod>)

Represents an objective that depends on the state of a given task. It is not intended to be instantiated.

Class CompletionTime<Mod> (extends TaskObjective<Mod>)

This class represents a function that increases linearly and whose zero is at the zero date.

- `CompletionTime<Mod>::CompletionTime<Mod>`
  `(Schedule<Mod> s, Task<Mod> t)`
  Creates a completion time function for the given task.

Class Earliness<Mod> (extends TaskObjective<Mod>)

This class represents a function that is zero if the task is late and increase linearly if it is done too early.

- `Earliness<Mod>::Earliness<Mod>(Schedule<Mod> s, Task<Mod> t, int date)`
  Creates an earliness cost for the specified task with respect to the given due-date.

Class Lateness<Mod> (extends TaskObjective<Mod>)

This class represents a function that increases linearly and that is zero when the tasks ends exactly at its due-date.

- `Lateness<Mod>::Lateness<Mod>(Schedule<Mod> s, Task<Mod> t, int date)`
  Creates a lateness cost for the given task and due-date.

Class Tardiness<Mod> (extends TaskObjective<Mod>)

This class represents a function that is zero if the task is on-time and increases linearly when it is late.

- `Tardiness<Mod>::Tardiness<Mod>(Schedule<Mod> s, Task<Mod> t, int date)`
  Creates a tardiness cost for the specified task and the given due-date.

Class PiecewiseLinearFunction<Mod> (extends TaskObjective<Mod>)

This class defines piecewise linear functions of the completion time of a task.

- `PiecewiseLinearFunction<Mod>:: PiecewiseLinearFunction<Mod>(Schedule<Mod> s, Task<Mod> t, int[] time, float[] cost)`
  Defines a function by linear interpolation of the pairs time-cost. External segments are considered half-lines.
- `void addPoint(int time, float cost)`
  Add a point at a given point in time and a given cost.

Class UnitCost<Mod> (extends TaskObjective<Mod>)

This class represents a function that is zero when the task is on-time and one when the task is late.

- `UnitCost<Mod>::UnitCost<Mod>(Schedule<Mod> s, Task<Mod> t, int date)`
  Creates a unit cost for the given task and due-date.

Class AbsenceCost<Mod> (extends TaskObjective<Mod>)

This class represents a unit cost for an optional task that is not taken.

- `AbsenceCost<Mod>::AbsenceCost<Mod>(Schedule<Mod> s, Task<Mod> t)`
  Creates a cost if the task is not executed. This has a meaning only for optional activities.

Class AlternativeCost<Mod> (extends TaskObjective<Mod>)

This class represents a unit cost for choosing an alternative for Multi-Mode activities

- `AlternativeCost<Mod>::AlternativeCost<Mod> (Schedule<Mod> s, MultiModeActivity<Mod> act, int mode)`
  Creates a cost if the activity is executed in mode "m".

Class AgregObjective<Mod> (extends SchedulingObjective<Mod>)

Objective that aggregates several objectives. It is an abstract class.

Class MaxObjective<Mod> (extends AgregObjective<Mod>)

This class represents a function that takes the maximum of its component functions.

- `MaxObjective<Mod>::MaxObjective<Mod>(Schedule<Mod> s, SchedulingObjective<Mod>[] objs)`
  Creates a new function that is the max of the given objective functions.
- `MaxObjective<Mod>::MaxObjective<Mod>(Schedule<Mod> s)`
  Creates a new Max objective function.
- `void add(SchedulingObjective<Mod> obj)`
  Adds a function as component of this max-function.

Class SumObjective<Mod> (extends AgregObjective<Mod>)

This class represents a function that returns the sum of its component functions.

- `SumObjective<Mod>::SumObjective<Mod>(Schedule<Mod> s, SchedulingObjective<Mod>[] objs)`
  Creates a new function that is the sum of the given objective functions.

Functions and operators that define an objective

- `SchedulingObjective<Mod> completionTimeOf(Task<Mod> t)`
  Defines the completion time of task t.

- `SchedulingObjective<Mod> earlinessOf(Task<Mod> t, int duedate)`
  Defines the earliness of task t with respect to the due-date.
- `SchedulingObjective<Mod> latenessOf(Task<Mod> t, int duedate)`
  Defines the lateness of task t with respect to the due-date.
- `SchedulingObjective<Mod> makespanOf(Schedule<Mod> s)`
  Defines the makespan of the schedule.
- `SchedulingObjective<Mod> tardinessOf(Task<Mod> t, int duedate)`
  Defines the tardiness of task t with respect to the due-date.
- `SchedulingObjective<Mod> unitCostOf(Task<Mod> t, int duedate)`
  Defines a unit step cost for task t with respect to the due-date.
- `SchedulingObjective<Mod> absenceCostOf(Task<Mod> t)`
  Defines a unit cost for the absence of task "t".
- `SchedulingObjective<Mod> alternativeCostOf (MultiModeActivity<Mod> t, int mode)`
  Defines a unit cost for task t being executed in the given mode.
- `SchedulingObjective<Mod> maxOf (SchedulingObjective<Mod>[] objs)`
  Defines the max of several objective functions.
- `SchedulingObjective<Mod> sumOf (SchedulingObjective<Mod>[] objs)`
  Defines the sum of several objective functions.
- `SchedulingObjective<Mod> operator +(float w, SchedulingObjective<Mod> obj)`
  Adds a constant to an objective function.
- `SchedulingObjective<Mod> operator + (SchedulingObjective<Mod> obj, float w)`
  Adds a constant to an objective function.
- `SchedulingObjective<Mod> operator + (SchedulingObjective<Mod> obj1, SchedulingObjective<Mod> obj2)`
  Adds two objective functions.
- `SchedulingObjective<Mod> operator *(float w, SchedulingObjective<Mod> obj)`
  Multiplies an objective function by a constant factor.
- `SchedulingObjective<Mod> operator * (SchedulingObjective<Mod> obj, float w)`
  Multiplies an objective function by a constant factor.

B

COMPLETE MODEL EXAMPLES

This chapter contains the complete code for running several models with AEON. All the models make use of the `Script` class that is responsible for parsing command-line options. The code of `Script` is given in Section B.3.

B.1 Job-Shop like Problems

We present first a very classical model for the Job-Shop Problem with makespan minimization. It reads files in the ORLIB format for JSPs.

```
1  include "fullSchedule";
2  Script scr("ft10.txt");
3  ifstream input = scr.getInput();
4
5  //File reading and parameters initialization
6  string doc = input.getLine();
7  cout << doc << endl;
8  int nbjobs = input.getInt();
9  int nbmachines = input.getInt();
10 range jobs = 1..nbjobs;
11 range machines = 0..nbmachines-1;
12 range tasks = 1..nbjobs*nbmachines;
13 int proc[tasks] = 0;
14 int mach[tasks] = 0;
15 int job[jobs,machines] = 0;
16 int i = 1;
17 forall(j in jobs) {
18     forall(t in machines) {
19         int m = input.getInt();
```

```
20            int d = input.getInt();
21            proc[i] = d;
22            mach[i] = m;
23            job[j,t] = i;
24            i++;
25        }
26    }
27
28    //Schedule definition
29    Schedule<Mod> s();
30    Job<Mod> J[i in jobs](s,IntToString(i));
31    Machine<Mod> M[i in machines](s,IntToString(i));
32    Activity<Mod> A[i in tasks](s,proc[i],IntToString(i));
33    forall(i in tasks)A[i].requires(M[mach[i]]);
34    forall(i in jobs){
35          J[i].containsInSequence(all(j in machines)A[job[i,j]]);
36    }
37    s.minimizeObj(makespanOf(s));
38
39    //Synthesis
40    scr.resolve(s);
```

Here is an alternative model for the Job-Shop Problem.

```
1   include "fullSchedule";
2   Script scr("ft10.txt");
3   ifstream input = scr.getInput();
4
5   //File reading and parameters initialization
6   string doc = input.getLine();
7   int nbjobs = input.getInt();
8   int nbmachines = input.getInt();
9   range jobs = 1..nbjobs;
10  range machines = 0..nbmachines−1;
11  range tasks = 1..nbjobs∗nbmachines;
12  int proc[tasks] = 0;
13  int mach[tasks] = 0;
14  int job[jobs,machines] = 0;
15  int i = 1;
16  forall(j in jobs) {
17      forall(t in machines) {
18          int m = input.getInt();
19          int d = input.getInt();
20          proc[i] = d;
21          mach[i] = m;
22          job[j,t] = i;
```

```
23          i++;
24      }
25  }
26
27  //Schedule definition
28  Schedule<Mod> s();
29  Reservoir<Mod> M[i in machines](s,0,5,5,IntToString(i));
30  MultiModeActivity<Mod> A[i in tasks](s,2,"Act"+IntToString(i));
31  forall(i in tasks){
32      A[i].setProcTime(1,proc[i],proc[i]);
33      A[i].requires(1,M[mach[i]],3);
34      A[i].setProcTime(2,proc[i],proc[i]);
35      A[i].requires(2,M[mach[i]],4);
36  }
37  forall(i in tasks:i%nbmachines!=0) A[i].precedes(A[i+1]);
38  s.minimizeObj(maxOf(all(i in tasks)completionTimeOf(A[i])));
39
40  //Synthesis
41  scr.resolve(s);
```

The model for the Open-Shop reads problems in the Gueret&Prins format. This model makes use of both jobs and machines. It is also possible to design a model with only machines.

```
1   include "fullSchedule";
2   Script scr("GP07−01.TXT");
3   ifstream input = scr.getInput();
4
5   //File reading and parameters initialization
6   string doc = input.getLine();
7   int nbjobs = input.getInt();
8   int nbmachines = input.getInt();
9   range jobs = 1..nbjobs;
10  range machines = 0..nbmachines−1;
11  range tasks = 1..nbjobs∗nbmachines;
12  int proc[tasks] = 0;
13  int mach[tasks] = 0;
14  int job[jobs,machines] = 0;
15  int i = 1;
16  forall(j in jobs) {
17      forall(t in machines) {
18          int d = input.getInt();
19          proc[i] = d;
20          mach[i] = t;
21          job[j,t] = i;
22          i++;
```

```
23      }
24  }
25
26  //Schedule definition
27  Schedule<Mod> s();
28  Job<Mod> J[i in jobs](s,IntToString(i));
29  Machine<Mod> M[i in machines](s,IntToString(i));
30  Activity<Mod> A[i in tasks](s,proc[i],IntToString(i));
31  forall(i in tasks)A[i].requires(M[mach[i]]);
32  forall(i in jobs)J[i].contains(all(j in machines)A[job[i,j]]);
33  forall(i in jobs)J[i].noOverlap();
34  s.minimizeObj(makespanOf(s));
35
36
37  //Synthesis
38  scr.resolve(s);
```

The Group-Shop problem is an hybrid between a Job-Shop and an Open-Shop. We present here a model different from the one presented in Section 5.3.2. Rather than creating a sub-job for each group, we explicitly state all the precedence constraints between all the pair of activities. The program reads the input data in a file in the format of C. Blum.

```
1   include "fullSchedule";
2   Script scr("ft10_3.gss");
3   ifstream input = scr.getInput();
4
5
6   int nbjobs = input.getInt();
7   int nbmachines = input.getInt();
8   range jobs = 1..nbjobs;
9   range machines = 0..nbmachines−1;
10
11  //Schedule definition
12  Schedule<Mod> s();
13  Job<Mod> J[i in jobs](s,"J"+IntToString(i));
14  Machine<Mod> M[i in machines](s,"M"+IntToString(i));
15  int[][] group = new int[][jobs];
16  int nbt[jobs];
17
18  Activity<Mod>[][] A = new Activity<Mod>[][jobs];
19
20  int i = 1;
21  forall(j in jobs) {
22      nbt[j] = input.getInt();
23      A[j] = new Activity<Mod>[1..nbt[j]];
```

```
24      forall(t in 1..nbt[j]) {
25          int m = input.getInt();
26          int d = input.getInt();
27          A[j][t] = Activity<Mod>(s,d,"A"+IntToString(i));
28          A[j][t].requires(M[m]);
29          J[j].contains(A[j][t]);
30          i++;
31      }
32  }
33
34  int nbJ = 0;
35  forall(j in jobs){
36      group[j] = new int[1..nbt[j]];
37      forall(t in 1..nbt[j]){
38          group[j][t] = input.getInt();
39          nbJ = max(nbJ,group[j][t]);
40      }
41  }
42
43  forall(i in jobs) J[i].noOverlap();
44  forall(j in jobs){
45      forall(t in 1..nbt[j], u in 1..nbt[j]){
46          if(group[j][t]<group[j][u]){
47              A[j][t].precedes(A[j][u]);
48          }
49      }
50  }
51  s.minimizeObj(makespanOf(s));
52
53  //Synthesis
54  scr.resolve(s);
```

The Flexible Job-Shop is an example of problem where there is an alternative between the resources to use by each activity. The program reads files in the FJSPLIB format. In this format, the processing time is given for each alternative but it is equal for all alternatives of an activity. We will see later an example of multi-mode activities.

```
1  include "fullSchedule";
2  Script scr("mt06.fjs");
3  ifstream input = scr.getInput();
4
5  //File reading and parameters initialization
6  string[] sizes = splitLine(input.getLine());
7  int nbjobs = sizes[0].toInt();
8  int nbmachines = sizes[1].toInt();
9
```

```
10  range jobs = 1..nbjobs;
11  range machines = 1..nbmachines;
12  int[][][] proc = new int[][][jobs];
13  int[][][] mach = new int[][][jobs];
14  forall(j in jobs) {
15      int nbtasks = input.getInt();
16      proc[j] = new int[][1..nbtasks];
17      mach[j] = new int[][1..nbtasks];
18      forall(t in 1..nbtasks) {
19          int nbalter = input.getInt();
20          proc[j][t] = new int[1..nbalter];
21          mach[j][t] = new int[1..nbalter];
22          forall(a in 1..nbalter){
23              mach[j][t][a] = input.getInt();
24              proc[j][t][a] = input.getInt();
25          }
26      }
27  }
28
29  //Schedule definition
30  Schedule<Mod> s();
31  Job<Mod> J[i in jobs](s,IntToString(i));
32  Machine<Mod> M[i in machines](s,IntToString(i));
33  Activity<Mod>[][] A = new Activity<Mod>[][jobs];
34  forall(i in jobs){
35      A[i] = new Activity<Mod>[k in proc[i].getRange()]
36                                  (s,proc[i][k][1],IntToString(i)+" "+IntToString(k));
37      J[i].containsInSequence(A[i]);
38      forall(k in proc[i].getRange()){
39          A[i][k].requiresOneOf(all(a in mach[i][k].getRange())M[mach[i][k][a]]);
40      }
41  }
42  s.minimizeObj(makespanOf(s));
43
44
45  //Synthesis
46  scr.resolve(s);
```

Here is the model of the Just-In-Time Job-Shop we used. It reads files in the format of [BFS08].

```
1  include "fullSchedule";
2  Script scr("ET_test1_10x2.txt");
3  ifstream input = scr.getInput();
4
5  //File reading and parameters initialization
```

```
6  int nbjobs = input.getInt();
7  int nbmachines = input.getInt();
8  range jobs = 1..nbjobs;
9  range machines = 0..nbmachines−1;
10 range tasks = 1..nbjobs∗nbmachines;
11 int proc[tasks] = 0;
12 int mach[tasks] = 0;
13 int job[jobs,machines] = 0;
14 int dd[tasks] = 0;
15 float ec[tasks] = 0;
16 float tc[tasks] = 0;
17 int i = 1;
18 forall(j in jobs) {
19     forall(t in machines) {
20         mach[i] = input.getInt();
21         proc[i] = input.getInt();
22         dd[i] = input.getInt();
23         ec[i] = input.getFloat();
24         tc[i] = input.getFloat();
25         job[j,t] = i;
26         i++;
27     }
28 }
29
30 //Schedule definition
31 Schedule<Mod> s();
32 Job<Mod> J[i in jobs](s,IntToString(i));
33 Machine<Mod> M[i in machines](s,IntToString(i));
34 Activity<Mod> A[i in tasks](s,proc[i],IntToString(i));
35 forall(i in tasks)A[i].requires(M[mach[i]]);
36 forall(i in jobs)J[i].containsInSequence(all(j in machines)A[job[i,j]]);
37 Tardiness<Mod> T[i in tasks](s,A[i],dd[i]);
38 Earliness<Mod> E[i in tasks](s,A[i],dd[i]);
39 s.minimizeObj(sumOf(all(i in tasks)(T[i]∗tc[i]+E[i]∗ec[i])));
40
41 //Synthesis
42 scr.resolve(s);
```

As a comparison, the following example is a model of the Just-In-Time Job-Shop without using the modeling facilities of AEON. It directly uses the COMET Scheduling API (augmented with our own global constraint).

```
1 include "fullSchedule";
2
3 int start = System.getCPUTime();
4 Script scr("ET_test1_10x2.txt");
```

```
5  ifstream input = scr.getInput();
6
7  //File reading and parameters initialization
8  int nbjobs = input.getInt();
9  int nbmachines = input.getInt();
10 range jobs = 1..nbjobs;
11 range machines = 0..nbmachines−1;
12 range tasks = 1..nbjobs∗nbmachines;
13 int proc[tasks] = 0;
14 int mach[tasks] = 0;
15 int job[jobs,machines] = 0;
16 int dd[tasks] = 0;
17 int ec[tasks] = 0;
18 int tc[tasks] = 0;
19 int i = 1;
20 forall(j in jobs) {
21     forall(t in machines) {
22         mach[i] = input.getInt();
23         proc[i] = input.getInt();
24         dd[i] = input.getInt();
25         ec[i] = (int)(100∗input.getFloat());
26         tc[i] = (int)(100∗input.getFloat());
27         job[j,t] = i;
28         i++;
29     }
30 }
31
32 //Schedule definition
33 MyScheduler<CP> cp(1000);
34 MyUnaryResource<CP> M[i in machines](cp);
35 Activity<CP> A[i in tasks](cp,proc[i]);
36 var<CP>{int} cost(cp,0..System.getMAXINT());
37
38 //This is a wrapper for the machines needed by the JITObjective.
39 ResourceFacade<CP> rf[m in machines](M[m]);
40 JITObjective jit(cp, cost, A, dd, all(i in tasks)(float)ec[i],
41                                   all(i in tasks)(float)tc[i], rf);
42
43 minimize<cp>
44     cost
45 subject to{
46     forall(i in tasks){
47         A[i].requires(M[mach[i]]);
48         rf[mach[i]].addRequest(new Request<CP>(rf[mach[i]],A[i],1,false,false));
49     }
```

```
50      forall(i in jobs,j in machines:j!=machines.getUp())
51          A[job[i,j]].precedes(A[job[i,j+1]]);
52      cp.post(cost == sum(i in tasks)(max(0,max((A[i].end()−dd[i])∗tc[i],
53                      (dd[i]−A[i].end())∗ec[i]))));
54  // Alternatives to the previous line. They don't all have the same strength.
55  // cp.post(cost == sum(i in tasks)(max((A[i].end()−dd[i])∗tc[i],0)+
56  //     max(0,(dd[i]−A[i].end())∗ec[i])));
57  // cp.post(cost == sum(i in tasks)(max(max((A[i].end()−dd[i])∗tc[i],0),
58  //     max(0,(dd[i]−A[i].end())∗ec[i]))));
59  // cp.post(cost == sum(i in tasks)(max((A[i].end()−dd[i])∗tc[i],
60  //     (dd[i]−A[i].end())∗ec[i])));
61  // The global constraint for the objective
62      cp.post(jit);
63  }using{
64  // The branching heuristic of the global constraint
65      jit.branch();
66  // Placing all the activities when the schedule order is fixed.
67      placeJIT(cp,cost,A,dd,all(i in tasks)(float)ec[i],all(i in tasks)(float)tc[i]);
68      cout << "cost = " << cost.getMin()/100.0 << endl;
69  }
```

B.2 Other Problems

The $1|r_i|\sum w_iC_i$ is a problem with one machine. The goal is to minimize the weighted sum of the completion time of all activities subject to the machine capacity constraint and the release dates of the activities.

```
1   include "fullSchedule";
2   Script scr("test_20_60_0.1wC");
3   ifstream input = scr.getInput();
4
5   //File reading
6   int sz = input.getInt();
7   range acts = 1..sz;
8   int p[acts];
9   int r[acts];
10  int w[acts];
11  forall(i in acts){
12      r[i] = input.getInt();
13      p[i] = input.getInt();
14      w[i] = input.getInt();
15  }
16
17  //Problem definition
```

```
18 Schedule<Mod> s();
19 Activity<Mod> A[i in acts](s,p[i],"A"+IntToString(i));
20 forall(i in acts) A[i].isReleasedAt(r[i]);
21 Machine<Mod> M(s,"M");
22 forall(i in acts) A[i].requires(M);
23 s.minimizeObj(sumOf(all(i in acts)(w[i]*completionTimeOf(A[i]))));
24
25 //Synthesis
26 scr.resolve(s);
```

The $P||C_{max}$ problem has been presented in Section 4.2.1. We show here a model using AEON and an equivalent model using the Scheduling API of COMET. The second model is very inefficient, while the first recognizes the problem and uses a search algorithm for bin-packing problems.

```
1  include "fullSchedule";
2  Script scr("U_1_0010_05_0.txt");
3  ifstream input = scr.getInput();
4
5  range machines = 1..input.getInt();
6  range tasks = 1..input.getInt();
7
8  int proc[i in tasks] = input.getInt();
9
10 //Schedule definition
11 Schedule<Mod> s();
12 Machine<Mod> M[i in machines](s,IntToString(i));
13 Activity<Mod> A[i in tasks](s,proc[i],IntToString(i));
14 forall(i in tasks)A[i].requiresOneOf(M);
15 s.minimizeObj(makespanOf(s));
16
17 //Synthesis
18 scr.resolve(s);
```

```
1  include "fullSchedule";
2  Script scr("U_1_0010_05_0.txt");
3  ifstream input = scr.getInput();
4
5  range machines = 1..input.getInt();
6  range tasks = 1..input.getInt();
7  int proc[i in tasks] = input.getInt();
8
9  //Schedule definition
10 Scheduler<CP> cp(sum(i in tasks)proc[i]);
11 UnaryResource<CP> M[i in machines](cp);
12 AlternativeUnaryResource<CP> R(cp);
```

```
13  forall(i in machines) R.add(M[i]);
14
15  Activity<CP> A[i in tasks](cp,proc[i]);
16  var<CP>{int} m(cp,0..sum(i in tasks)proc[i]);
17  minimize<cp>
18      m
19  subject to{
20      forall(i in tasks) A[i].requires(R);
21      cp.post(m==max(i in tasks) A[i].end());
22  }using{
23      R.assignAlternatives();
24      forall(m in machines) M[m].rank();
25      cp.post(m == m.getMin());
26      cout << m << endl;
27  }
```

To complete the tour of models, we show three variations of the RCPSP problem. First, the RCPSP with Earliness and Tardiness cost on all activities. Then the RCPSP/max, that includes precedence constraints limiting the maximal distance between two activities. Finally, the Multi-mode RCPSP, where each activity has several modes, with different processing time and requirements. This last problem also introduces consumption of reservoirs.

```
1   include "fullSchedule";
2   Script scr("mv1.rcp.full");
3   ifstream input = scr.getInput();
4
5   range jobs = 1..input.getInt();
6   range res = 1..input.getInt();
7   int capa[i in res] = input.getInt();
8   int[][] succ = new int[][jobs];
9   int proc[jobs];
10  int req[jobs,res];
11  forall(i in jobs){
12      proc[i] = input.getInt();
13      forall(j in res){
14          req[i,j] = input.getInt();
15      }
16      int suc = input.getInt();
17      succ[i]=new int[1..suc];
18      forall(j in 1..suc)succ[i][j]=input.getInt();
19  }
20  int dd[jobs] = 0;
21  int ec[jobs] = 0;
22  int tc[jobs] = 0;
23  forall(i in 2..nbjobs-1){
```

```
24      int v = input.getInt()+1;
25      assert(v==i);
26      dd[i] = input.getInt();
27      ec[i] = input.getInt();
28      tc[i] = input.getInt();
29  }
30
31
32  Schedule<Mod> s();
33  Activity<Mod> A[i in jobs](s, proc[i], "Job"+IntToString(i));
34  CumulativeResource<Mod> R[i in res](s,capa[i],"Resource"+IntToString(i));
35  forall(i in jobs){
36      forall(j in succ[i].getRange()){
37          A[i].precedes(A[succ[i][j]]);
38      }
39      forall(j in res){
40          if(req[i,j]!=0)A[i].requires(R[j],req[i,j]);
41      }
42  }
43  s.minimizeObj(sumOf(all(i in jobs)(tardinessOf(A[i],dd[i])*tc[i]+
44                      earlinessOf(A[i],dd[i])*ec[i])));
45
46  //Synthesis
47  Solution<Mod> sol = scr.resolve(s);
```

```
1   include "fullSchedule";
2   Script scr("psp1.sch");
3   ifstream input = scr.getInput();
4
5   int start = System.getCPUTime();
6   int nbacts = input.getInt();
7   int nbres = input.getInt();
8   int nbrvr = input.getInt(); //zero
9   int nbrd = input.getInt(); //zero
10  range acts = 0..nbacts+1;
11  int[][] succ = new int[][acts];
12  int[][] length = new int[][acts];
13  int proc[acts];
14  int req[acts,1..nbres+nbrvr+nbrd];
15  forall(i in acts){
16      input.getInt();//index
17      input.getInt();//nb modes = 1
18      int suc = input.getInt();
19      succ[i]=new int[1..suc];
20      forall(j in 1..suc)succ[i][j]=input.getInt();
```

```
21      length[i]=new int[1..suc];
22      string[] reso = splitLine(input.getLine(),"\t");
23      forall(j in 1..suc)
24          length[i][j]=reso[j-1].substring(1,reso[j-1].length()-2).toInt();
25  }
26  forall(i in acts){
27      input.getInt();//index
28      input.getInt();//nbmodes = 1
29      proc[i] = input.getInt();
30      forall(j in 1..nbres+nbrvr+nbrd){
31          req[i,j] = input.getInt();
32      }
33  }
34  int capa[1..nbres+nbrvr+nbrd];
35  forall(i in capa.getRange()){
36      capa[i] = input.getInt();
37  }
38
39  Schedule<Mod> s();
40  range tasks = acts;
41  Activity<Mod> A[i in tasks](s, proc[i], "Job"+IntToString(i));
42  range resources = 1..nbres+nbrvr+nbrd;
43  CumulativeResource<Mod> R[i in resources](s,capa[i],"Res"+IntToString(i));
44  forall(i in tasks){
45      forall(j in succ[i].getRange()){
46          //Start to Start precedences
47          A[i].start().comesBefore(A[succ[i][j]].start(),-length[i][j]);
48      }
49      forall(j in resources){
50          if(req[i,j]!=0)A[i].requires(R[j],req[i,j]);
51      }
52  }
53  s.minimizeObj(makespanOf(s));
54
55  //Synthesis
56  Solution<Mod> sol = scr.resolve(s);
```

```
1   include "fullSchedule";
2   Script scr("j1010_10.mm");
3   ifstream input = scr.getInput();
4
5   //File reading not shown
6
7   Schedule<Mod> s();
8   range tasks = 1..nbjobs;
```

```
MultiModeActivity<Mod> A[i in tasks](s, modes[i], "Job"+IntToString(i));
range resources = 1..nbres;
range reservoirs = nbres+1..nbres+nbrvr;
range doubly = nbres+nbrvr+1..nbres+nbrvr+nbrd;
CumulativeResource<Mod> R[i in resources](s,capa[i],"Res"+IntToString(i));
Reservoir<Mod> N[i in reservoirs](s,0,capa[i],capa[i],"Rvr"+IntToString(i));
forall(i in tasks){
    forall(j in succ[i].getRange()){
        A[i].precedes(A[succ[i][j]]);
    }
    forall(k in 1..modes[i]){
        A[i].setCurrentMode(k);
        A[i].setProcTime(proc[i][k]);
        forall(j in resources){
            if(req[i,j][k]!=0)A[i].requires(R[j],req[i,j][k]);
        }
        forall(j in reservoirs){
            if(req[i,j][k]!=0)A[i].consumes(N[j],req[i,j][k]);
        }
    }
}
s.minimizeObj(tardinessOf(A[nbjobs],duedate)*tardCost);

//Synthesis
scr.resolve(s);
```

B.3 The `Script` class

The `Script` class is a class to read options from the command line. It allows to feed the file to read data from (-f), the file to write output to (-o), the file to write statistics to (-O), the synthesizer to use (-a), a set of options for synthesizers (-x), and whether to use a visualization (-V). An example use of the line command options is the following:

```
> comet RCPSP.co -a Gx10+LNS -f j12038_7.sm -o res.txt
-x timeLimit=300
```

This command asks to solve the instance j13038_7 of the RCPSP problem using a greedy search repeated 10 times to start a LNS using the best solutions among the ones given by the greedy. The output is written to “res.txt” and we limit the running time to 300 seconds.

```
class Script{
    int start; //the start time in CPUTime
    int start2; //the start time in WCTime
    ifstream input; //the input file
    ScheduleSynthesizer synth; //the synthesizer
```

```
6     string outf; //the output file name
7     string statf; //the statistic file name
8     string fname; //the input file name
9     string algo; //the name of the synthesizer to use
10    string opts; //the set of options
11
12    Script(string name){
13        cout << "This is Aeon! (c) JN Monette, INGI, UCLouvain" << endl;
14        start = System.getCPUTime();
15        start2 = System.getWCTime();
16        cout << SetPrecision(2) << endl;
17        //Input
18        string[] args = System.getArgs();
19        //Reading command line arguments and initializing options.
20        [...]
21
22        input = new ifstream(fname);
23        synth = getSynthesizer(algo);
24        synth.parseParameters(opts);
25        int end2 = System.getCPUTime();
26    }
27
28    ScheduleSynthesizer getSynthesizer(string algo){
29        if(algo.equals("LS")){
30            return ScheduleSynthesizer<LS>();
31        }else if(algo.equals("CP")){
32        //Trying all the possible synthesizers.
33        [...]
34
35        //Building compound synthesizers
36        }else{
37            string[] algos = splitLine(algo,"+");
38            if(algos.getSize()>1){
39                ScheduleSynthesizer tmp[algos.rng()];
40                forall(i in algos.rng()){
41                    tmp[i] = getSynthesizer(algos[i]);
42                }
43                return new ScheduleSynthesizer<Sequence>(tmp);
44            }else{
45                algos = splitLine(algo,"x");
46                if(algos.getSize()==2){
47                    int i = algos[algos.getUp()].toInt();
48                    ScheduleSynthesizer tmp =
49                                              getSynthesizer(algos[algos.getLow()]);
50                    return ScheduleSynthesizer<Repeat>(tmp,i);
```

```
51                  }else{
52                      return ScheduleSynthesizer();
53                  }
54              }
55          }
56      }
57
58      Solution<Mod> resolve(Schedule<Mod> s){
59          Solution<Mod> sol = synth.resolve(s);
60          if(sol!=null){
61              cout << "Sol Value = " << sol.getFloatValue() << endl;
62              if(outf!="")sol.printSolutionToFile(outf);
63          }else{
64              cout << "No Solution found" << endl;
65          }
66          if(statf!=""){
67              //Writing statistics to the stat file
68              [...]
69          }
70          return sol;
71      }
72
73      ifstream getInput(){ return input; }
74
75      [...]
76  }
```

C

FEATURES DESCRIPTION

This appendix presents all the features that are currently defined. In the first section, we describe all the characteristics directly extracted from the problem representation. In the second section, we list all the features derived from these characteristics.

C.1 Problem Characteristics

Activities

`nbActivities` (**integer**):
The total number of activities in the problem.

`nbModes` (**integer**):
The total number of (real) modes of activities in the problem.

`maxProcTime` (**integer**):
The largest processing time of all modes.

`minProcTime` (**integer**):
The smallest processing time of all modes (including option modes).

`nbFixedProcTimeModes` (**integer**):
The number of (real) modes with a fixed processing time.

`nbPreemptiveActivities` (**integer**):
The number of activities that are preemptive.

`nbMultiModeActivities` (**integer**):
The number of activities that have more than one mode.

`nbOptionalActivities` (**integer**):
The number of activities that are optional.

Precedences

`graphForm` (**string**):
The form of the graph. Possible values: `empty`, `chain`, `chains`, `intree`, `intrees`, `outtree`, `outtrees`, `tree`, `trees`, `cycle`, `cycles`, `dag`, `ggraph`

`nbPrecedences` (**integer**):
The number of precedences between activities (in the almost transitive reduction graph).

`nbSimplePrecedences` (**integer**):
The number of precedences that are simple (that is, which are of the form $C(a) + 0 \leq S(b)$).

`nbNoWaitPairs` (**string**):
The number of pairs of precedences that define a no-wait constraint.

`nbDeadlines` (**integer**):
The number of activities having a deadline (not implied by other precedences).

`nbReleaseDates` (**integer**):
The number of activities having a release date (not implied by other precedences).

`releaseDatesAreCommon` (**boolean**):
Whether all activities have the same release date.

`deadlinesAreCommon` (**boolean**):
Whether all activities have the same deadline.

Resources

`nbResources` (**integer**):
The total number of resources.

`nbMachines` (**integer**):
The number of machines.

`nbStateResources` (**integer**):
The number of state resources.

`nbReservoirs` (**integer**):
The number of reservoirs, including cumulative resources and machines.

`maxMaxCapacity` (**integer**):
The largest capacity of capacitated resources.

`minMaxCapacity` (**integer**):
The smallest capacity of capacitated resources.

Resources

`minMinCapacity` (**integer**):
The smallest minimal capacity of capacitated resources.
`maxMinCapacity` (**integer**):
The largest minimal capacity of capacitated resources.
`withTransitionTimes` (**boolean**):
Whether there are transition times defined on machines or state resources.
`maxNumberOfStates` (**integer**):
The largest number of states in any state resource.

Requirements

`reservoirProduction` (**boolean**):
Whether there is production of resources in reservoirs.
`reservoirConsumption` (**boolean**):
Whether there is consumption of resources in reservoirs.
`maxNumberOfRequestedStates` (**integer**):
The largest number of states that are required for a state resources.
`maxDisjunction` (**integer**):
The largest number of alternatives for a mode.
`minDisjunction` (**integer**):
The smallest number of alternatives for a mode.
`maxConjunction` (**integer**):
The largest number of conjunctive requests for an alternative of a mode.
`minConjunction` (**integer**):
The smallest number of conjunctive requests for an alternative of a mode.
`requirementsAreBipartite` (**boolean**):
Tells whether the set of resources can be divided in two sets such that an activity never requires two resources in the same set. This is used to recognize an Open-Shop, for instance.

Objectives

`objectiveType` (**string**):
Tells what the objective is. Possible values: `satisfaction`, `minimization`, `maximization`

Objectives

`objectiveForm` (**string**):
Describes the form of the main objective function. Possible values: `maximum, weightedSum, absent`

`numberOfObjectiveComponentsTypes` (**int**):
The number of different basic functions present in the objective function.

`objectiveComponentType` (**set of strings**):
Describes the basic functions that appear in the objective function. Possible values: `completionTime, lateness, earliness, tardiness, unitCost, alternativeCost, piecewiseLinear`

`objectiveScope` (**string**):
Tells if all activities influence on the objective value. Possible values: `allActivities, someActivities`

`objectiveDueDateForm` (**string**):
Tells whether there are due-dates and if they are common to all activities or there is one per activity or there several per activity. Possible values: `absent, common, onePerActivity, variable`

`objectiveOnlyOnMode` (**boolean**):
Whether the objective is only function of the modes of the activities.

`objectiveOnlyOnTime` (**boolean**):
Whether the objective only depends on the completion time of activities.

`objectiveIsConvex` (**boolean**):
Whether the objective is a convex function of the completion time of the activities.

`objectiveIsIncreasing` (**boolean**):
Whether the objective value always increases when an activity is scheduled later.

`objectiveIsDecreasing` (**boolean**):
Whether the objective value always decreases when an activity is scheduled later.

`costsAreUnit` (**boolean**):
Whether all the costs associated to individual functions are equal to one.

C.2 Features

Features are of three kinds: numeric values, labels and classes. Labels and classes are similar except in their use. We list here all the features we defined. Section 3.3 explains how features are used.

C.2.1 Numeric Values

Numeric values are derived from integer characteristics. Most of them are ratios of activities of a special type, or of precedences of a special type. This allows to test for particular cases simply by checking if the ratio is equal to 1 or to 0.

%MultiModeActivities = nbMultiModeActivities / nbActivities
%OptionalActivities = nbOptionalActivities / nbActivities
%PreemptiveActivities = nbPreemptiveActivities / nbActivities
%Deadlines = nbDeadlines / nbActivities
%ReleaseDates = nbReleaseDates / nbActivities
nbNoWaitPairsX2 = nbNoWaitPairs * 2.000000
%NoWait = nbNoWaitPairsX2 / nbPrecedences
%SimplePrecedences = nbSimplePrecedences / nbPrecedences

C.2.2 Labels and Classes

Each feature is described following this pattern:
Feature name ← *Specialized features*
≙ *Definition of the feature*
Optional textual description
The definition of a feature uses other features previously described and predicates, as defined in Section 3.3.1. The textual description gives explanations about the use of the feature, or about its definition.
For clarity, we divided the features in several categories: labels related to activities, labels for precedences, labels for the resources and requirements, labels of the objective function, and classes of problem.

Activities

FixedProcTime
≙ nbFixedProcTimeModes=nbModes
If all processing times are fixed, it is not necessary to decide them in the search procedure.

EqualProcTime ← FixedProcTime
≙ FixedProcTime⋀maxProcTime=minProcTime
UnitProcTime ← EqualProcTime
≙ EqualProcTime⋀maxProcTime=1
When all processing time are unit, many problems may become polynomially solvable.

Activities

Preemption ← SomePreemption
≙ %PreemptiveActivities=1.0

NoPreemption ← SomePreemption
≙ %PreemptiveActivities=0.0

SomePreemption
≙ ¬(Preemption)⋀¬(NoPreemption)
The two previous labels are for the particular cases where all activities are preemptive, or no activity is preemptive. SomePreemption holds in all other cases.

NoMultiMode
≙ %MultiModeActivities=0
When activities have several possible modes, the search algorithm must fix the mode.

NoOptionalActivities
≙ %OptionalActivities=0

SingleMode ← NoMultiMode, NoOptionalActivities
≙ NoMultiMode⋀NoOptionalActivities
We consider to be in SingleMode when there is no multi-mode activity and no optional activity. This is the simplest setting.

NoDeadlines
≙ %Deadlines=0

NoReleaseDates
≙ %ReleaseDates=0

CommonDeadline
≙ deadlinesAreCommon=true

CommonReleaseDate
≙ releaseDatesAreCommon=true
If all activities have the same release date, the problem can be shifted, such that the release date is equal to 0.

Precedences

NoNoWait
≙ nbNoWaitPairs=0
A no-wait precedence constraint is a pair of arcs between two nodes, that are of opposite length and opposite direction. That is $X+d \leq Y$ and $Y-d \leq X$. They might be replaced by $X + d = Y$.

Precedences

AllNoWait

≜ nbNoWaitPairsX2=nbPrecedences

AllSimpleNoWait ← AllNoWait

≜ AllNoWait⋀%SimplePrecedences=0.5

All precedences constraints have the form $C(A) = S(B)$. This is the case in the no-wait Job-Shop problem.

WithCycles

≜ (graphForm∈cycle,cycles,ggraph)⋁(¬(NoNoWait))

There are cycles in the precedences graph, if the graph is a single cycle, a set of cycles, a graph which contains cycles, or a graph with No-wait constraints (which is a cycle between two activities).

SimplePrecedenceGraph

≜ %SimplePrecedences=0

A simple precedence graph only has precedences of the form $C(A) \leq S(B)$. There exist no delay, and no cycles (if the problem is feasible).

SimpleTemporalConstraints ← NoDeadlines, NoNoWait, NoReleaseDates, SimplePrecedenceGraph

≜ NoReleaseDates⋀NoDeadlines⋀NoNoWait⋀SimplePrecedenceGraph

This features groups often encountered features of simple problems. This means there are no temporal constraints, except simple precedences between activities.

JobShopForm

≜ graphForm=chains

The precedence graph of a Job-Shop is composed of a set of chains of activities.

OpenShopForm

≜ graphForm=empty

The precedence graph of an Open-Shop is empty.

Resources

NoReservoir

≜ reservoirConsumption=false⋀reservoirProduction=false

There are no reservoirs when there is neither consumption nor production of resources. In that case, the reservoirs can be replaced by cumulative resources.

Resources

NoStates
≙ (nbStateResources=0)⋁(maxNumberOfStates≤1)⋁
(maxNumberOfRequestedStates≤1)

Cumulative ← NoStates, NoReservoir
≙ NoStates⋀NoReservoir
A cumulative problem is one with only cumulative (and hence disjunctive) resources.

Disjunctive ← Cumulative
≙ (maxMaxCapacity≤1⋀minMaxCapacity≤1⋀minMinCapacity=0⋀
maxMinCapacity=0)⋀Cumulative
This is a cumulative problem where all resources have a maximum capacity of 1, and there is no minimal capacity.

NoResource ← Disjunctive
≙ nbResources=0
When there is no resources, the problem becomes a PERT one.

AllCapacitiesAreEqual
≙ minMaxCapacity=maxMaxCapacity⋀minMinCapacity=
maxMinCapacity
This feature is used to describe the Cumulative Job-Shop problem.

MaxAlternatives ← SingleMode
≙ SingleMode⋀minDisjunction=nbResources
This feature tells that all activities have the choice between all the resources. This is used to describe the $P||C_{max}$ problem.

FewAlternatives ← SingleMode
≙ SingleMode⋀maxDisjunction≤2
This limits the number of alternatives by activity to 2.

NoAlternative ← FewAlternatives
≙ FewAlternatives⋀maxDisjunction≤1
When there is no alternative, the used resources are fixed. Otherwise, it is necessary to use a search procedure that fixes them.

Objective

Satisfaction
≙ objectiveType=satisfaction

Maximization

Objective

≙ objectiveType=maximization

Minimization

≙ objectiveType=minimization

TimeObjective

≙ objectiveOnlyOnTime=true

A objective that depends only on the time at which activities end.

MonotonicallyIncreasing ← TimeObjective

≙ objectiveIsIncreasing=true⋀TimeObjective

Problems with monotonically increasing objectives require activities to be scheduled early.

MonotonicallyDecreasing ← TimeObjective

≙ objectiveIsDecreasing=true⋀TimeObjective

Convex ← TimeObjective

≙ objectiveIsConvex=true⋀TimeObjective

Problems with a convex objective function can be solved easily once the resources constraints are respected.

Makespan ← Convex, Minimization, MonotonicallyIncreasing

≙ MonotonicallyIncreasing⋀Minimization⋀costsAreUnit=true ⋀objectiveScope=allActivities⋀objectiveComponentsType= completionTime⋀numberOfObjectiveComponentsTypes=1⋀ objectiveForm=maximum⋀Convex

The makespan is the most classical objective. It is to minimize the completion time of the latest activity.

Completion ← Convex, Minimization, MonotonicallyIncreasing

≙ Minimization⋀MonotonicallyIncreasing⋀Convex⋀objectiveForm =weightedSum⋀numberOfObjectiveComponentsTypes=1⋀ objectiveComponentsType=completionTime⋀objectiveScope= allActivities

This feature holds for problems whose objective is the minimization of the weithed sum of the completion times of the activities.

Tardiness ← Convex, Minimization, MonotonicallyIncreasing

≙ objectiveScope=allActivities⋀objectiveComponentsType=tardiness⋀ numberOfObjectiveComponentsTypes=1⋀objectiveForm=weightedSum ⋀Convex⋀MonotonicallyIncreasing⋀Minimization

This feature holds for problems whose objective is the minimization of the weithed sum of the tardiness costs of the activities.

EarlinessTardiness ← Convex, Minimization

Objective

≜ objectiveDepth=2⋀objectiveDueDateForm∈onePerActivity,unique ⋀objectiveComponentsType∋earliness⋀objectiveComponentsType∋ tardiness⋀numberOfObjectiveComponentsTypes=2⋀objectiveForm= weightedSum⋀Convex⋀Minimization

This is the Just-In-Time objective, that is the minimization of the weighted sum of the earliness and tardiness costs for each activity.

Classes

JobShop ← CumulativeJobShop, Disjunctive, FixedProcTime, FlexibleJobShop, JobShopForm, NoAlternative, NoPreemption, SimpleTemporalConstraints

≜ Disjunctive⋀NoAlternative⋀NoPreemption⋀JobShopForm⋀ SimpleTemporalConstraints⋀FixedProcTime⋀withTransitionTimes =false⋀maxConjunction≤1⋀minConjunction≥1

This is the base for different variations that have different objective functions. The Job-Shop is a particular cas of the flexible Job-Shop and of the Cumulative Job-Shop.

JobShopWithMakespan ← CumulativeJobShopWithMakespan, FlexibleJobShopWithMakespan, GroupShopWithMakespan, JobShop, Makespan

≜ Makespan⋀JobShop

JobShopWithTardiness ← JobShop, RCPSP, Tardiness

≜ Tardiness⋀JobShop

JustInTimeJobShop ← JobShop1, EarlinessTardiness

≜ EarlinessTardiness⋀JobShop

OpenShop ← Disjunctive, FixedProcTime, NoAlternative, NoPreemption, OpenShopForm, SimpleTemporalConstraints

≜ FixedProcTime⋀Disjunctive⋀NoAlternative⋀OpenShopForm⋀ NoPreemption⋀SimpleTemporalConstraints⋀maxConjunction≤2⋀ minConjunction≥2⋀requirementsAreBipartite=true

As for the Job-Shop, this is the base that can be further defined with different objective functions.

OpenShopWithMakespan ← Makespan, OpenShop

≜ Makespan⋀OpenShop

GroupShopWithMakespan ← Disjunctive, FixedProcTime, Makespan, NoAlternative, NoPreemption, SimpleTemporalConstraints

Classes

≙ NoPreemption⋀SimpleTemporalConstraints⋀FixedProcTime⋀
withTransitionTimes=false⋀Disjunctive⋀Makespan⋀NoAlternative

FlexibleJobShop ← Disjunctive, FixedProcTime, JobShopForm, NoPreemption, SimpleTemporalConstraints, SingleMode

≙ SingleMode⋀SimpleTemporalConstraints⋀NoPreemption⋀
FixedProcTime⋀Disjunctive⋀JobShopForm

FlexibleJobShopWithMakespan ← FlexibleJobShop, Makespan

≙ Makespan⋀FlexibleJobShop

JobShopMaxSlack ← Disjunctive, FixedProcTime, NoAlternative, NoDeadlines, NoNoWait, NoPreemption, NoReleaseDates

≙ Disjunctive⋀NoPreemption⋀NoAlternative⋀NoNoWait⋀NoDeadlines
⋀NoReleaseDates⋀graphForm=cycles⋀FixedProcTime

JobShopMaxSlackWithMakespan ← JobShopMaxSlack, Makespan

≙ Makespan⋀JobShopMaxSlack

NoWaitJobShop ← AllSimpleNoWait, Disjunctive, FixedProcTime, JobShopForm, NoAlternative, NoDeadlines, NoPreemption, NoReleaseDates

≙ NoPreemption⋀AllSimpleNoWait⋀FixedProcTime⋀NoDeadlines⋀
NoReleaseDates⋀Disjunctive⋀JobShopForm⋀NoAlternative

NoWaitJobShopWithMakespan ← Makespan, NoWaitJobShop

≙ Makespan⋀NoWaitJobShop

CumulativeJobShop ← AllCapacitiesAreEqual, Cumulative, FixedProcTime, JobShopForm, NoAlternative, NoPreemption, SimpleTemporalConstraints

≙ minMinCapacity=0⋀maxMinCapacity=0⋀FixedProcTime⋀
NoAlternative⋀withTransitionTimes=false⋀NoAlternative⋀Cumulative
⋀JobShopForm⋀NoPreemption⋀SimpleTemporalConstraints⋀
AllCapacitiesAreEqual

CumulativeJobShopWithMakespan ← CumulativeJobShop, Makespan

≙ Makespan⋀CumulativeJobShop

MMRCPSP ← FixedProcTime, NoPreemption, NoStates, SimpleTemporalConstraints, Tardiness

≙ minMinCapacity=0⋀maxMinCapacity=0⋀FixedProcTime⋀Tardiness⋀
NoStates⋀NoPreemption⋀SimpleTemporalConstraints

This is the Multi-mode RCPSP.

RCPSP ← Cumulative, MMRCPSP, NoAlternative

≙ Cumulative⋀MMRCPSP⋀NoAlternative

Classes

The RCPSP is a particular case of MMRCPSP without alternatives and only cumulative resources.

OneMachineWithCommonDueDate ← Disjunctive, EarlinessTardiness, FixedProcTime, NoAlternative, NoPreemption, SimpleTemporalConstraints

≙ Disjunctive⋀EarlinessTardiness⋀NoAlternative⋀SimpleTemporalConstraints⋀NoPreemption⋀objectiveDueDateForm=common⋀nbMachines=1⋀FixedProcTime⋀graphForm=empty

In this problem, there is only one machine that is required by all activities. They all need to finish at the same date and there is a penalty for being late ond being early.

1wC ← Completion, Disjunctive, NoAlternative, NoDeadlines, NoPreemption

≙ Disjunctive⋀nbMachines=1⋀NoAlternative⋀NoDeadlines⋀NoPreemption⋀graphForm=empty⋀Completion

In this problem ($1|r_i|\sum w_iC_i$), there is one machine and the objective is to minimize the weighted sum of the completion times. There are release dates but no deadlines.

PCmax ← Disjunctive, Makespan, MaxAlternatives, NoPreemption, SimpleTemporalConstraints

≙ NoPreemption⋀MaxAlternatives⋀graphForm=empty⋀Disjunctive⋀Makespan⋀SimpleTemporalConstraints

This describes the $P||C_{max}$ problem, where activities can be executed on any of the machines. The objective is the makespan.

Trolley ← JobShopForm, Makespan, NoAlternative, NoDeadlines, NoPreemption, NoReleaseDates

≙ maxDisjunction≤1⋀NoDeadlines⋀NoReleaseDates⋀withTransitionTimes=true⋀NoAlternative⋀Makespan⋀JobShopForm⋀NoPreemption⋀reservoirConsumption=false⋀reservoirProduction=false⋀nbStateResources=1⋀maxNumberOfStates>1⋀maxNumberOfRequestedStates>1

The Trolley problem includes cumulative and state resources, and sequence-dependent setup-times.

BIBLIOGRAPHY

[AC91] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. Technical Report 2, ORSA Journal on Computing, 1991. 18, 90

[AGU72] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972. 33, 34

[Bal69] E. Balas. Machine sequencing via disjunctive graphs: an implicit enumeration algorithm. *Operations Research*, 6(17):941–957, 1969. 21

[BDPS07] Sebastien Brand, Gregory J. Duck, Jakob Puchinger, and Peter J. Stuckey. A rule-based system for model transformation. In *ModRef'07: 6th International Workshop on Constraint Modelling and Reformulation*, 2007. 25

[Bea90] J. E. Beasley. OR-library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990. 79

[BFS08] Philippe Baptiste, Marta Flamini, and Francis Sourd. Lagrangian bounds for just-in-time job-shop scheduling. *Comput. Oper. Res.*, 35(3):906–915, 2008. 23, 84, 103, 104, 115, 140

[BJN+96] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1996. 23

[BLN01] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer Academic Publisher, 2001. 16, 98, 100

[BPS03] J. Christopher Beck, Patrick Prosser, and Evgeny Selensky. Vehicle routing and job shop scheduling: What's the difference? In *Proc. of the 13th International Conference on Automated Planning and Scheduling*, pages 267–276, 2003. 15

[BR03] J. Christopher Beck and Philippe Refalo. A hybrid approach to scheduling with earliness and tardiness costs. *Annals OR*, 118(1-4):49–71, 2003. 104

[Bru04] Peter Brucker. *Scheduling Algorithms, 4th Edition*. Springer, 2004. 14, 16, 22

[BS04] C. Blum and M. Sampels. An ant colony optimization algorithm for shop scheduling problems. *Journal of Mathematical Modelling and Algorithms*, 3(3):285–308, 2004. 83

[BYBS09] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. Automated algorithm tuning using f-races: Recent developments. In M. Caserta and S. Voß, editors, *Proceedings of MIC 2009, the 8th Metaheuristics International Conference*, page 10 pages, 2009. 121

[CJ06] Hadrien Cambazard and Narendra Jussien. Identifying and exploiting problem structures using explanation-based constraint programming. *Constraints*, 11(4):295–313, 2006. 27

[CL94] Yves Caseau and François Laburthe. Improved clp scheduling with task intervals. In *Proc. 11th Intl. Conf. on Logic Programming*, 1994. 18, 90

[COS00] A. Cesta, A. Oddi, and Stephen F. Smith. Iterative flattening: A scalable method for solving multi-capacity scheduling problems. In *Proceedings of AAAI 2000*, 2000. 21

[CP89] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, 1989. 18, 90, 104

[CP90] J. Carlier and E. Pinson. A practical use of jackson's preemptive schedule for solving the job-shop problem. *Annals of Operation Research*, 26(1-4):269–287, 1990. 18, 90, 104

[CP94] J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994. 18, 90, 98, 104

[CS03] Philippe Chrétienne and Francis Sourd. Pert scheduling with convex cost functions. *Theor. Comput. Sci.*, 292(1):145–164, 2003. 106, 107, 111, 118

[Dan98] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, August 1998. 23

[DMP89] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 83–93, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc. 18

[DP03] Emilie Danna and Laurent Perron. Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs. In *CP*, pages 817–821, 2003. 104, 114

[DPPH01] Ulrich Dorndorf, Erwin Pesch, and Toàn Phan-Huy. Solving the open shop scheduling problem. *Journal of Scheduling*, 4:157 – 174, 2001. 18, 90

[DT93] M. Dell'Amico and M. Trubian. Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research*, 41:231–252, 1993. 4, 22, 75, 80

[Dyn09] Dynadec, Dynamic Decision Technologies Inc. Comet tutorial, v2.0, 2009. 24

[EMDP08] I. Essafi, Y. Mati, and S. Dauzère-Pèréz. A genetic local search algorithm for minimizing total weighted tardiness in the job-shop scheduling problem. *Computers & Operations Research*, 35(8):2599–2616, 2008. 82

[FHJ+08] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008. 25

[FJMHM05] Alan M. Frisch, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. The rules of constraint modelling. In *Proceedings IJCAI*, pages 109–116, 2005. 25

[Gec06] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`. 63, 98

[GLLR79] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. 5:287–326, 1979. 14

[GP99] C. Guéret and C. Prins. A new lower bound for the open-shop problem. *Annals of Operation Research*, 92:165–183, 1999. 90, 98

[HG95] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *IJCAI-95*, pages 607–613. Morgan Kaufmann, 1995. 19

[HMMP10] Pierre Hansen, Nenad Mladenović, and José A. Moreno-Pérez. Variable neighbourhood search: methods and applications. *Annals of Operations Research*, 175(1):367–407, 2010. 22

[Hoo06] John N. Hooker. *Integrated Methods for Optimization (International Series in Operations Research & Management Science)*. Springer-Verlag New York, 2006. 23

[HS04] Holger Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. 20

[HS07] Yann Hendel and Francis Sourd. An improved earliness-tardiness timing algorithm. *Computers & OR*, 34(10):2931–2938, 2007. 107

[ILO05] ILOG. ILOG SCHEDULER 6.1 reference manual. 2005. 26

[Jac56] J. R. Jackson. An extension of johnson's results on job lot scheduling. *Naval Research Logistics Quarterly*, 3:201–203, 1956. 93

[KH06] Rainer Kolisch and Sonke Hartmann. Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European Journal of Operational Research*, 174(1):23–37, October 2006. 80

[Kre00] S. Kreipl. A large step random walk for minimizing total weighted tardiness in a job shop. *Journal of Scheduling*, 3(3):125–138, 2000. 82

[KS97] R. Kolisch and A. Sprecher. Psplib — a project scheduling problem library. *European Journal of Operational Research*, 96:205–216, 1997. 13, 80

[Lab05] Philippe Laborie. Complete mcs-based search: Application to resource constrained project scheduling. In *IJCAI-05*, pages 181–186, 2005. 19

[LAL92] Peter J. M. van Laarhoven, Emile H. L. Aarts, and Jan Karel Lenstra. Job shop scheduling by simulated annealing. *Operations Research*, 40(1):113–125, 1992. 22

[LCVG94] Claude Le Pape, Philippe Couronné, Didier Vergamini, and Vincent Gosselin. Time-versus-capacity compromises in project scheduling. In *Proc. of the 13th Workshop of the U.K. Planning Special Interest Group*, 1994. 19

[LG07] Philippe Laborie and Daniel Godard. Self-adapting large neighborhood search: Application to single-mode scheduling problems. *In Proceedings MISTA-07, Paris*, pages 276–284, 2007. 19, 26, 80, 84, 104, 117

[LKA04] Joseph Leung, Laurie Kelly, and James H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004. 16

[LON05] S. Q. Liu, H. L. Ong, and K. M. Ng. A fast tabu search algorithm for the group shop scheduling problem. *Adv. Eng. Softw.*, 36(8):533–539, 2005. 83

[MDD07] Jean-Noël Monette, Yves Deville, and Pierre Dupont. A position-based propagator for the open-shop problem. In *CPAIOR07: Proceedings of the 4th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 186–199, Berlin, Heidelberg, 2007. Springer-Verlag. 6

[MDV09a] Jean-Noël Monette, Yves Deville, and Pascal Van Hentenryck. Aeon: Synthesizing scheduling algorithms from high-level models. *Operations Research and Cyber-Infrastructure, Proceedings of the 11th Informs Computing Society Conference*, pages 43–59, 2009. 6

[MDV09b] Jean-Noël Monette, Yves Deville, and Pascal Van Hentenryck. Just-in-time scheduling with constraint programming. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *ICAPS*, pages 241–248. AAAI, 2009. 6

[Mil03] Michela Milano. *Constraint and Integer Programming: Toward a Unified Methodology (Operations Research/Computer Science Interfaces", 27)*. Kluwer Academic Publishers, Norwell, MA, USA, 2003. 23

[MNR+08] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia De La Banda, and Mark Wallace. The design of the zinc modelling language. *Constraints*, 13(3):229–267, 2008. 25, 121

[MS96] Paul Martin and David B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In *In Proc. of the 5th International IPCO Conference*, pages 389–403. Springer, 1996. 90

[NBF+04] Wim Nuijten, T. Bousonville, Filippo Focacci, Daniel Godard, and Claude Le Pape. Towards an industrial manufacturing scheduling problem and test bed. *PMS*, 2004. 13

[NLP98] Wim Nuijten and Claude Le Pape. Constraint-based job shop scheduling with ILOG SCHEDULER. *Journal of Heuristics*, 3(4):271–286, 1998. 90

[NS05] Eugeniusz Nowicki and Czeslaw Smutnicki. An advanced tabu search algorithm for the job shop problem. *J. of Scheduling*, 8(2):145–159, 2005. 79

[Par09] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2009. 5

[PR91] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.*, 33(1):60–100, 1991. 23

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006. 16

[Ref04] Philippe Refalo. Impact-based search strategies for constraint programming. In *CP 2004, Toronto (Canada)*, pages 557–571, 2004. 27

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 362–367, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence. 18

[Sha98] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. *CP*, page 417âĂŞ431, 1998. 19, 114

[SW92] Jorge P. Sousa and Laurence A. Wolsey. A time indexed formulation of non-preemptive single machine scheduling problems. *Math. Program.*, 54(3):353–367, 1992. 23

[Van99] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999. 13, 26

[VBQ03] Vicente Valls, Francisco Ballestín, and Sacramento Quintanilla. A hybrid genetic algorithm for the rcpsp. Technical report, Department of Statistics and Operations Research, University of Valencia, 2003. 80

[VDH01] M. Vanhoucke, E. Demeulemeester, and W. Herroelen. An exact procedure for the resource-constrained weighted earliness-tardiness project scheduling problem. *Annals of Operations Research*, 102:179–196, 2001. 104

[Vil04] Petr Vilím. $O(n \log n)$ filtering algorithms for unary resource constraint. In Jean-Charles Régin and Michel Rueher, editors, *Proceedings of CP-AI-OR 2004*, volume 3011 of *Lecture Notes in Computer Science*, pages 335–347, Nice, France, April 2004. Springer-Verlag. 18, 90, 98, 104

[VM05] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005. 20, 22, 26, 75, 82, 107, 121

[VM07] Pascal Van Hentenryck and Laurent Michel. Synthesis of constraint-based local search algorithms from high-level models. *AAAI'07, Vancouver, British Columbia*, 2007. 26

[WN99] Laurence A. Wolsey and George L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley-Interscience, November 1999. 23

[Wol98] Laurence A. Wolsey. *Integer Programming*. Wiley-Interscience, September 1998. 23

[Wol05] A. Wolf. Better propagation for non-preemptive single-resource constraint problems. In *Proceedings of CSCLP2004*, volume 3419 of *Lecture Notes in Artificial Intelligence*, pages 201–215, 2005. 90

[Zho97] Jianyang Zhou. A permutation-based approach for solving the job-shop problem. *Constraints*, 2(2):185–213, 1997. 90, 91

[ZLRG08] Chao Yong Zhang, PeiGen Li, YunQing Rao, and ZaiLin Guan. A very fast ts/sa algorithm for the job shop scheduling problem. *Computers & Operations Research*, 35(1):282 – 294, 2008. Part Special Issue: Applications of OR in Finance. 79